HARDWARE ARCHITECTURE FOR SEMANTIC COMPARISON

A Dissertation

by

SUNEIL MOHAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2012

Major Subject: Computer Engineering

HARDWARE ARCHITECTURE FOR SEMANTIC COMPARISON

A Dissertation

by

SUNEIL MOHAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2012

Major Subject: Computer Engineering

ABSTRACT

Hardware Architecture for Semantic Comparison. (May 2012)

Suneil Mohan, B.E, Anna University, Chennai, India

Chair of Advisory Committee: Dr. Rabi N. Mahapatra

Semantic Routed Networks provide a superior infrastructure for complex search engines. In a Semantic Routed Network (SRN), the routers are the critical component and they perform semantic comparison as their key computation. As the amount of information available on the Internet grows, the speed and efficiency with which information can be retrieved to the user becomes important. Most current search engines scale to meet the growing demand by deploying large data centers with general purpose computers that consume many megawatts of power. Reducing the power consumption of these data centers while providing better performance, will help reduce the costs of operation significantly.

Performing operations in parallel is a key optimization step for better performance on general purpose CPUs. Current techniques for parallelization include architectures that are multi-core and have multiple thread handling capabilities. These coarse grained approaches have considerable resource management overhead and provide only sub-linear speedup.

This dissertation proposes techniques towards a highly parallel, power efficient architecture that performs semantic comparisons as its core activity. Hardware-centric parallel algorithms have been developed to populate the required data structures followed by computation of semantic similarity. The performance of the proposed design is further enhanced using a pipelined architecture. The proposed algorithms were also implemented on two contemporary platforms such as the Nvidia CUDA and an FPGA for performance comparison. In order to validate the designs, a semantic benchmark was also been created. It has been shown that a dedicated semantic comparator delivers significantly better performance compared to other platforms.

Results show that the proposed hardware semantic comparison architecture delivers a speedup performance of up to 10^5 while reducing power consumption by 80% compared to traditional computing platforms. Future research directions including better power optimization, architecting the complete semantic router and using the semantic benchmark for SRN research are also discussed.

DEDICATION

To my parents

ACKNOWLEDGMENTS

Completing my PhD and writing this dissertation would not have been possible without the guidance, help and support of many individuals who in their unique ways helped me through my time in graduate school. It is a pleasure to be able to convey my gratitude to you all. Forgive me, if due to an extremely unfortunate oversight, I miss someone.

First and foremost, my utmost gratitude and thanks are due to my advisor Dr. Rabi Mahapatra. Without his valuable guidance, support and advice, none of this would have been possible. He has been my advisor, mentor and role model throughout my time here at Texas A & M.

I gratefully acknowledge the contributions of my advisory committee members Dr. Hank Walker, Dr. Riccardo Bettati and Dr. Deepa Kundur for their valuable feedback and guidance at critical points that enabled the successful completion of my research and my PhD.

I would not have been here if it had not been for my family. Thank to my parents for their support, prayers and advice in immeasurable quantities. To my extended family who believed that I could 'do it' even when I had serious doubts about the whole grad school endeavor, thank you for believing in me. I appreciate you listening to me rant and supporting me in so many ways over the years.

Many thanks go to my present and former colleagues at the Embedded Systems and Codesign Group including Aalap, Nikhil, Suman, Ron, Jason & Jagannath. You have helped me in everything from debugging code to proof reading papers to sharing in my joys and sorrows through graduate school.

I've been fortunate to have a close group of friends who have been there for me in more ways than I can individually list: Thank you Parikshith, Swetha, Kymberleigh, Swati, Vinodh, Lilian, Emily, Amy, Ram, Siva, Ayan, Nur, Suresh & Arun . You've been there for me when I needed someone to talk to and given me much needed support and encouragement. I cannot thank you all enough. A special thank you to Parikshith for being an amazing house-mate who was always there to support me, listen to me rant, be a role model and not drive me up the wall. As I heard horror stories about other people's weird house-mates, I was always grateful that we had a calm setup. To the members of the Phorum, I thank you for your everlasting support and encouragement through some of the toughest times.

The staff of the Department of Computer Science and Engineering deserve a special thank you. Thank you Dave, Jeremy, Tony and the rest of CSG for helping me with computers and related technical support. Thank you Adrienne and Lindsay for answering patiently the many questions that I've posed to you over the years. Thank you Tina, Marilyn, Sybil, Lisa and the rest of the Advising, Accounting and Administrative staff of the department for helping me with various issues. Thank you Bruce for handling the logistics including packages, keys and other facility management issues that I've approached you with. Thank you all for answering the many hundreds of questions that I must have asked each of you personally over time.

Thank you Dr. Mahapatra, Dr. Hurley, Dr. Walker, Dr. Bettati and Dr. Sarin for having me as your Teaching Assistant over the many semesters. It was an incredible experience that I would not trade for another. Thank you also to my students from the various classes. You guys taught me a whole lot and introduced me to the American undergraduate culture in a way that would not have been possible otherwise.

Finally, I would like to thank the many more people who have contributed towards my ability to reach this stage in my education. I would like to thank you and apologize for not being able to name you individually as I would have liked to; you did play an important role in my ability to complete my PhD.

NOMENCLATURE

API	Application Programming Interface
BF	Bloom Filter
CAM	Content Addressable Memory
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFF	D Flip Flop
FPGA	Field Programmable Gate Array
NIH	National Institute of Health
$P_{false+ve}$	Probability of False Positive
RAM	Random Access Memory
SoC	System On Chip
SIMD	Single Instruction Multiple Data
SRN	Semantic Routed Network
TF-IDF	Term Frequency - Inverse Document Frequency
UMLS	Unified Medical Language System

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

Searching for information has become an important activity on the Internet and other information systems. At present the world wide consumption of the *web search* as a service is estimated to be 13 billion search queries per month and growing at 38% annually [1].

Users increasingly expect that search engines provide intelligent and relevant matches. For example, when they enter a search term such as *healthy lifestyle*; they expect that articles and web-links that cover a broad range of topics related to a *healthy lifestyle* such as *balanced diet*, *exercise* and *time management* be made available to them.

As the amount of information available on the web grows, there is increased demand for relevant and meaningful search results. The first search engines on the *web* used to perform simple keyword comparisons. This is no longer sufficient. Users are no longer satisfied with search engines returning simple keyword match documents. Indeed, returning simple keyword search results these days would result in millions and millions of search results, most of which would have no relevance to the query.

With the increase in the pervasiveness of the web, massively increasing usage and demands placed on search engines, it is not surprising that implementing a fast meaningful search service is challenging. Current search engines require a lot of processing power because of the volume of data involved. Hence, search engines such as Google and Yahoo are forced to deploy a large number of servers in data centers to service the enormous web traffic [2].

This dissertation follows the style of *IEEE Transactions on Parallel and Distributed Systems*.

These data centers consume a significant amount of power (Megawatts) which has become a key factor in data center planning and management [3] [4] [5]. Therefore there is a pressing need to deliver a technique that can provide high performance while balancing power usage.

Traditional search engines use common information retrieval techniques such as TF-IDF [6] to rate documents and objects on the web. These techniques rely on keyword comparisons to create a ranking on the relative importance of the object. Unfortunately, the reliance on keywords means that if the document does not contain the keyword, it would not be ranked at all. This leads to situations where the search for *healthy lifestyle* would probably not return a single result (object/document) that did not have the words *healthy lifestyle* as part of its description or in the text. In order to deliver better search results, search engines are increasingly turning to the area of semantics to understand both input queries as well as search results, so that they can match and return relevant results [7].

Semantic search techniques on the other hand, decomposes concepts and ideas into their basic concepts and compares these concepts against the known data store [8] [9]. This allows the locating and identification of related articles that may not have the same keywords in the document. Hence semantic search while performing a similar service does not rely on the same search techniques as existing search engines.

A Semantic Routed Network (SRN) proposed initially in [10] provides a superior infrastructure enabling semantic information retrival. A SRN allows search engines to operate with smaller number of servers by enabling automatic reorganization of their search indices based on content. This content reorganization and subsequent query routing requires the use of semantic routers [11]. The critical component of the SRN which determines its performance is a semantic router.

Semantic routers perform semantic comparison using a cosine similarity (dot product) computation scheme that can be elegantly parallelized [12]. By creating a power efficient hardware co-processor that performs dot product we can enable realization of a semantic router. Cosine similarity is also the key-computation in traditional search paradigms such as TF-IDF, hence creating a power efficient hardware co-processor can help in both areas. In this dissertation we will use the terms cosine similarity and dot product interchangeably.

Performing operations in parallel is a key optimization step for better performance on general purpose CPUs. Current techniques for parallelization include architectures that are multi-core [13] and have multiple thread handling capabilities [14] [15]. These coarse grained approaches have considerable resource management overhead and provide only sub-linear speedup. In addition, these approached require new programming models with parallel compilers & libraries as supporting infrastructure. For implementing the dot product required for semantic comparison, a fine-grained approach to parallelization is more suitable. This involves a hardware/software co-design strategy involving hardware centric algorithms and mapping of the algorithm to architecture.

In this dissertation we propose techniques towards a highly parallel, power efficient architecture that performs semantic comparisons as its core activity. Hardware-centric parallel algorithms have been developed to populate the required data structures followed by computation of semantic similarity. The performance of the proposed design is further enhanced using a pipelined architecture. The proposed algorithms were also implemented on two contemporary platforms - the Nvidia CUDA [16] and an FPGA [17] for performance comparison. In order to validate the designs, a semantic benchmark was also been created. It has been shown that a dedicated semantic comparator delivers significantly better performance compared to other platforms.

Results show that the proposed low power architecture presented consumes 82% less power and demonstrates a speed-up in the order of 10^5 compared to a contemporary hardware design, and in the order of 10^3 compared to software approach for large number of basis vectors. Future research directions including better power optimization, architecting the complete semantic router and using the semantic benchmark for SRN research are also discussed.

The key contributions of this dissertation can be summarized as follows:

1. Design of algorithms and data structures that are suitable for semantic comparison in a semantic router.

2. Design of power efficient parallel hardware architecture to perform semantic comparison.

3. Design of pipelined architecture for the enhancement of throughput.

4. Implementation on reference platforms such as the Nvidia CUDA and an FPGA for performance comparison.

5. Development of a semantic benchmark to create semantic traffic for validation.

The rest of this dissertation is laid out as follows. In Section- 2 we present background material and related work. We begin with a discussion on what is semantic routing and how it works. This is followed by a description of Semantic Routed Networks and the application of Semantic Routed Networks to Search. Once we have established how an SRN works and how it can be applied to Search, we discuss the methods of Vector based semantic comparison and the challenges involved. One of the key computation steps in semantic comparison is the computation of cosine similarity. This contains a challenge because of the size of the data-structure used. We present the challenge and our approach to solving this. We then proceed to explanation of a space-efficient data structure known as a Bloom Filter which we use extensively in the architecture and close out this section with an overview of the Fowler / Noll / Vo (FNV) hash algorithm which is also used.

Section 3 introduces our architecture at a high level. We present details on what semantic comparison involves, the descriptors that are used as semantic metadata and an architectural overview of the computation and the requirements and stages of computation. This section also talks about the basic steps involved in the processing and our fine-grained parallelization scheme. The semantic comparison process consists of two major steps and several sub-steps. The next two sections presents each of the major steps in detail.

Section 4 presents the first part of the computation - the architecture required to create and populate the data structures that we use in our later computational stage. We present an analysis of our approach and provide timing and power figures for the design. We explore design alternatives in creating a set of hash-values in a power and time efficient manner.

Section 5 presents the second part of the computation. In this chapter, we present the details of the architecture and highlight key optimizations and design decisions that allows us impressive speedup while remaining power efficient. We also present overall results comparing, our design and that of doing the same computation on a traditional server class processor, timing analysis and overall power figures.

In Section 6 we present performance improvements and alternative implementations. Pipelining the architecture gives us noticeable speedup without excessive power overheads. Hence we present the design of the pipelined version of the architecture and power and timing figures. We then present two alternative implementations of our basic architecture - first on an Nvidia CUDA platform and then on an FPGA. We provide performance comparisons between these platforms and the base ASIC design proposed in the earlier sections.

It is not possible to evaluate our design without a benchmark. Since, to the best of our knowledge, we are the first to explore semantic comparison, there does not exist any datasets that we can use to create the appropriate kinds of traffic. In Section 7 we present the details of a Semantic Benchmark that we created. The details of its composition and features is discussed.

Conclusions and a discussion on potential directions of future research are provided in Section 8. A overview of the various areas that were discussed in the dissertation are provided.

2. BACKGROUND AND RELATED WORK

2.1 Semantic Routing

Semantic routing, the routing of information based on the semantics, is done by humans everyday. It is just not called Semantic Routing in daily usage. The concept of semantic routing can be illustrated using the following scenario: Alice, a first year graduate student of Microbiology wants certain details about the latest Influenza outbreak. She asks her friend Bob who is a post-doctoral researcher if he has specific information. Bob feels that his supervisor Mark is more knowledgeable on the topic, so Bob forwards Alice's request to Mark. Mark responds to Alice's questions and offers to help her. In this scenario, each person is a "semantic router", who determines the next hop based on whom they know to be the best (knowledgeable) person to handle this query. The query here is the 'request for information on the Influenza outbreak' and the best identified resource was Mark who responds to Alice with the offer to help.

2.2 Semantic Routed Networks

A Semantic Routed Network (SRN) routes information on the network using the concept of semantic routing. In this section we provide a brief overview of the working and organization of the SRN, the methods used in the organization of the SRN and Search as an application.

2.2.1 Overview

The abstract model of the SRN consists of two types of nodes: router nodes and resource object nodes as shown in Fig. 2.1. Resource nodes are the objects or resources of a network, whereas the router nodes are semantic routers [11]. Semantic routers attach themselves (with an entry in their routing tables) to both other semantic routers as well as resources. Each router maintains a listing (routing table) of all the resources and other routers attached to itself.

When a query (request for a resource) is injected into the network, the semantic routers compare semantically the query to the entries in their routing table and route the message to the appropriate (or best match) location. The final destination (most likely the resource) would then respond to the originator of the query with the required data. An SRN would be implemented as an overlay network built on top of existing IP networks or a web-service infrastructure. Therefore the response to a query is likely to be a URI containing information about access mechanisms of the resource or relevant web-service response, depending on the type of semantic object/ resource. If SOAP [18] messaging is used then semantic routers may implement a SOAP processor as well.

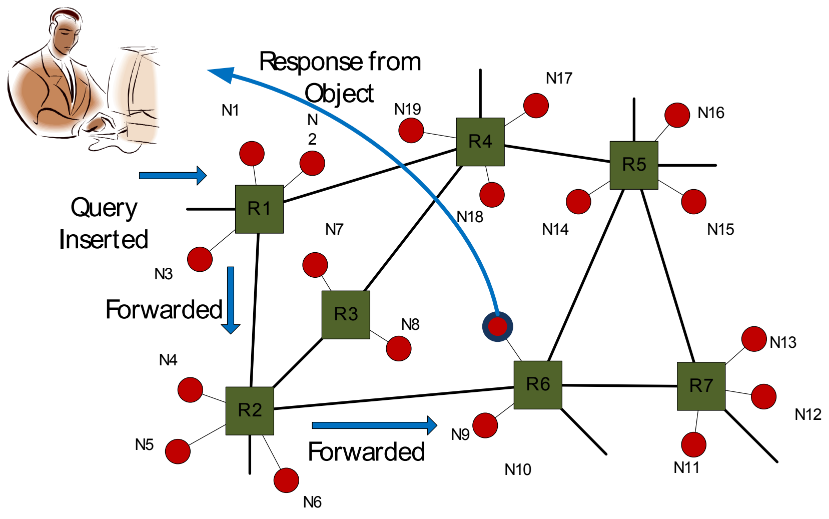


Fig. 2.1. Abstract Model of a Semantic Routed Network

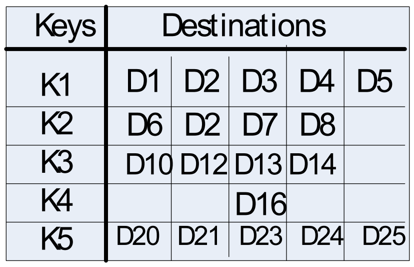
The semantic objects may be data or services having various granularities, formats, structures, service access/delivery mechanisms. These semantic objects are annotated using *metadata* that describes the resource. This metadata can be based on generic or upper ontologies, like Wordnet [19], Gene Ontology [20] or more specialized ontologies / standards like the Unified Medical Language System (ULMS) [21]. By “description” we mean a flat unstructured set of topics which describes a resource. The elementary metadata items that from the description are referred to as a Tags. Compound topics may be formed by collecting multiple Tags. These Tags can be harvested from the structured metadata that describes the resource. Further details of the semantic objects and the comparison scheme are given in Section . 3.1

2.2.2 Methods and Techniques Involved in SRN

The SRN is implemented as a Small World Network. The Small World Network topology offers better congestion behavior and requires less number of routers to interconnect a given number of resources compared to over hierarchical topology [11]. A small world network topology is characterized by two factors: (a) Small expected path length (hop distances) and (b) Large clustering coefficient. A clustering coefficient is the probability that two nodes are connected if they have a common peer [22].

The SRN has several built-in mechanisms to ensure that the network topology is maintained as well as the routing tables of the routers contain the most updated information. Keeping the routing table updates is modeled on current networking technology i.e. BGP (Border Gateway Protocol) [23] where routers periodically exchange messages with each other giving information about resources and other routers connected to them. Once these messages are received, routing table optimization algorithms are run to update the routing table. Fig. 2.2 shows a sample of the semantic routing table layout. Each *Key* corresponds to a semantic interest of the router, and each *Destination* (per key) shows the nodes that the router knows is semantically similar to that key.

Mapping this back to our example in Section 2.1 , the keys correspond to the various interests of a person, and the destinations the people/resources connected



Keys	Destinations				
K1	D1	D2	D3	D4	D5
K2	D6	D2	D7	D8	
K3	D10	D12	D13	D14	
K4			D16		
K5	D20	D21	D23	D24	D25

Fig. 2.2. Semantic Routing Table

with that interest. Bob's interests include Influenza (key) and one of the resources he knew with experience in this area was Mark (destination).

We will now briefly describe the major algorithms that help maintain the semantic routing table.

Node Clustering Algorithm

In order to locate nodes that contain descriptions that are semantically similar to the entries in the routing tables, a semantic router, periodically sends out messages (queries) with a similarity threshold value into the network. Nodes (both resource nodes and other semantic routers) whose descriptions are semantically similar to the query (with a similarity metric greater than the threshold value) respond back to the originating router, which keeps a log of the responses. In order to keep its routing table populated, a router may periodically send out queries with lower threshold values in order to expand the range of resources that is it aware of. Resource nodes that join a network , send out similar messages advertising themselves for routers to become aware of them.

Routing Table Optimization

There are three primary methods involved in maintaining the semantic routing table of a router. Fig. 2.3 shows the three methods.

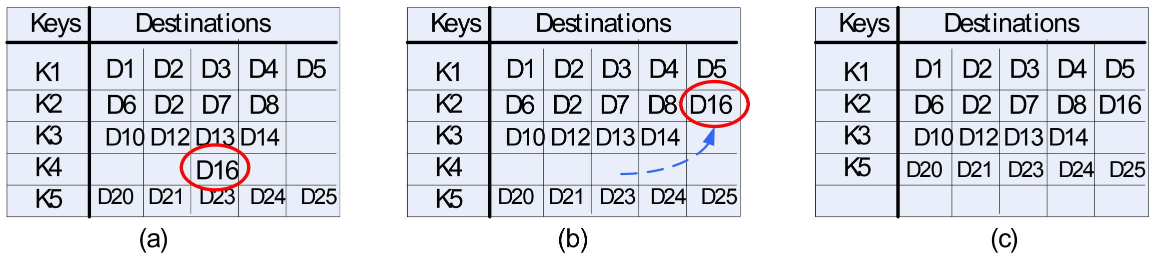
1. Routing table entry eviction: Since the routing tables are of limited size, the routers strive to keep the quality of the entries high by evicting entries that are less similar and replacing them with entries that are semantically closer to the appropriate interest group. Fig. 2.3(a) shows D16 being considered for eviction.

2. Reallocation of destinations: A single key of a routing table can contain more than one destination for the key. If an entry is to be evicted (see previous method), an attempt is made to re-allocate it. This involves looking at the other keys in the table to see if it can be used for another key based on the relevance of the entry. In Fig. 2.3(b) the destination D16 is being re-allocated from key K4 to a location that is alongside K2.

3. Reallocation of keys: Occasionally, a router may find that an entire row of the table needs to be re-allocated or changed. This could be due to either an entire set of resources going off-line or identification of a new more appropriate interest. If the change is due to the destinations going offline, the entries are dropped and the table is purged of that Key. In Fig. 2.3 the key K4 is now dropped since there are no new destinations and the table is compacted. If the change is due to a new key, re-allocation of the destination entries is attempted using the previous step before dropping the entry.

2.2.3 Application of SRN to Search Engines

Semantic Routed Networks (SRN) can be used to enable the automatic reorganization of the search-index of a distributed search engine [10]. This would allow for



(a)

Keys	Destinations				
K1	D1	D2	D3	D4	D5
K2	D6	D2	D7	D8	
K3	D10	D12	D13	D14	
K4			D16		
K5	D20	D21	D23	D24	D25

(b)

Keys	Destinations				
K1	D1	D2	D3	D4	D5
K2	D6	D2	D7	D8	D16
K3	D10	D12	D13	D14	
K4					
K5	D20	D21	D23	D24	D25

(c)

Keys	Destinations				
K1	D1	D2	D3	D4	D5
K2	D6	D2	D7	D8	D16
K3	D10	D12	D13	D14	
K5	D20	D21	D23	D24	D25

Fig. 2.3. Routing Table Optimization

the reduction in the number of servers deployed by the search engine. Search engines such as Google and Bing are scrambling to provide more meaningful and relevant results without an increase in the response time (real-time search). This leads to an increased computational load per query.

Fig. 2.4 illustrates the architecture of a typical internet search engine core [3]. It is assumed that every document indexed by a search engine is assigned a unique *Document ID*. A search engine consists of two core components (A) Index Servers and (B) Doc Servers. Users send raw queries q (at rate Q) to the front-end server. A query processor multicasts these queries to N_s Index shards (rate reduced to Q/N_s) which constitute the index server. For a given query q, index shards return a sorted list results to the document server (a list of ID and the corresponding rank). Index shards are replicated for capacity (often across geographical locations for fault-tolerance). For example, multiple instances of the index servers form a pool as shown in Fig. 2.4. Given a set of *Document IDs* for a given query, a document processor returns the relevant (URL,snippet) which is returned to the user as a result by the front-end service.

The index is typically generated (using statistical measures like TF-IDF [6] or latent semantic dimensions [24]) with rows representing words/dimensions and several columns representing related *Document IDs* and their corresponding weights (Wi,j). The cosine similarity score is computed across the n-dimensions of every index shard. The resulting similarity scores are sorted and further processed by the document servers to return to the user.

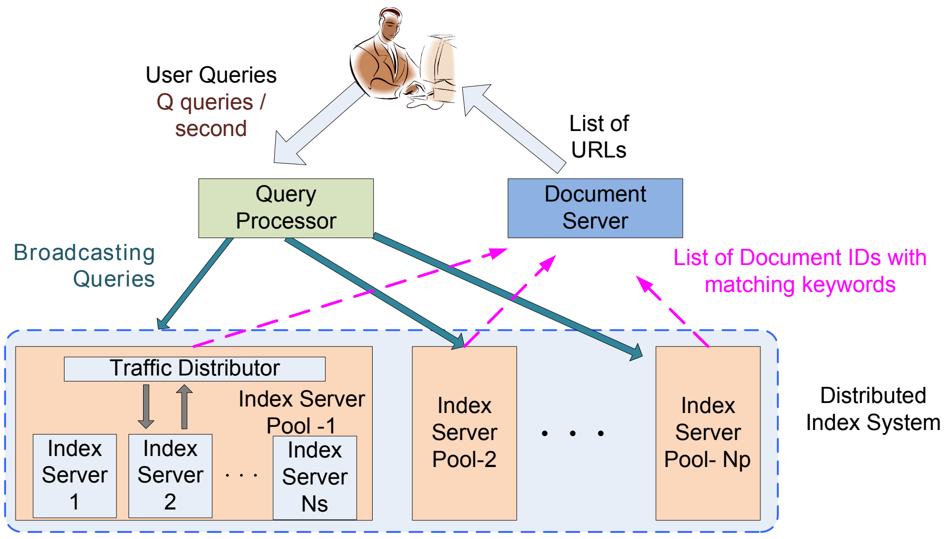


Fig. 2.4. Architecture of a Typical Search Engine

An energy efficient alternative to query broadcasting is to systematically distribute objects to the pools/shards based on the meaning of the objects, so that documents having similar content (or belonging to a similar topic) are in the same shard as shown in Fig. 2.5. Such index distribution would need a specialized network to deliver the queries to a specific pool(s)/shard(s). This arrangement will avoid unnecessary query traffic (query rate reduced to Q/Np as in Fig. 2.5) to all pools thereby allowing smaller number of servers ($n_s \approx Ns/Np = Ns/1000 < Ns$) in each pool and lower power consumption. This specialized network can be elegantly implemented using a Semantic Routed Network.

This semantic routed network (SRN) consisting of multiple semantic routers [10] can be used for:

1. To selectively forward/route a (query) message to an index shard based on the meaning of the query; and

2. To automatically re-distribute index entries based on meaning of the documents/objects.

Achieving true semantic search requires a way to "represent meaning in computers" as compared to naive term-by-term comparison [6] which exists today. This question was addressed in [12] and uses Vectors to perform the semantic comparison.

New methods to represent "composite" meaning in computers have been designed and proven to be superior to TF-IDF in [12]. This enables conjunction, disambiguation & representation of "complex concepts" their synonyms and hyponyms using a Tensor-based transformation model [24]. The use of this model enables the creation of a Semantic Routed Network [10] (SRN) in the index shards of a search engine.

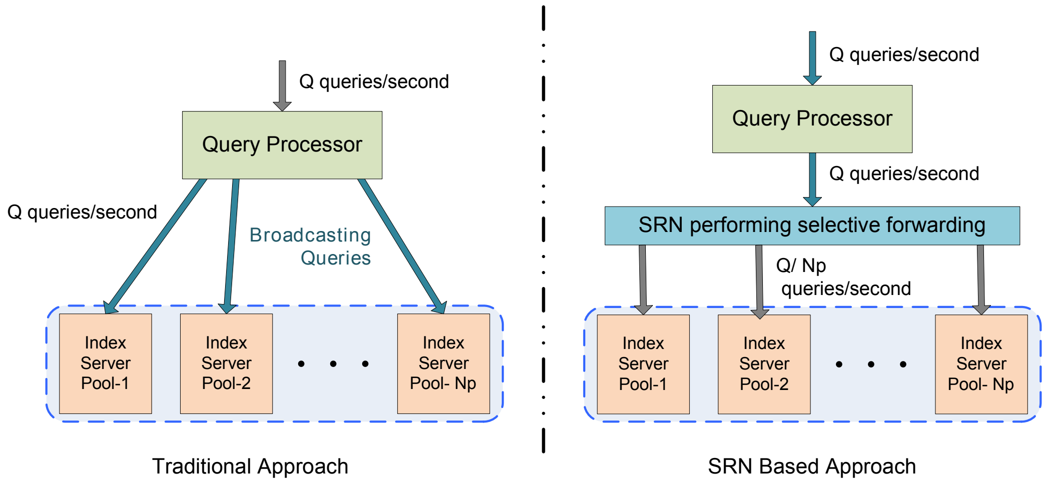


Fig. 2.5. Search Engine with SRN Integrated

A SRN has been proven to enable the automatic reorganization of the search-index of a distributed search engine based on the principles of a Small World Network [8]. Using this model, an incoming query q can be injected into the SRN. The SRM network then routes the query until resolved by reaching the appropriate Index Server. This implements selective unicast instead of the inefficient multicast mechanism discussed previously. Fig. 2.5 shows the reorganization of the Index shards using the principles of SRN. Results in [11] [8] show through simulation that convergence to a Small World Network is possible, with bounded overhead. It is proven that query resolution is guaranteed within a maximum of 3 hops from injection on average (for a fully re-organized index).

These techniques address users need for “semantically” meaningful and relevant results without an increase in the response time (real-time search). Although an SRN has been proven to theoretically converge (in simulation), our experiments in the next Section emphasize the need for hardware in the index shards to meet the cost of a significantly larger computational load per query resolution using a Tensor model and SRN-based index server pool.

2.3 Vector Based Semantic Comparison

Contemporary search engines use vector models for automatic text retrieval [6]. Each of the comparisons being performed by the computers in Fig. 2.4 performs this computation. Techniques that we described in [12] extend these vector models for more efficient semantic comparisons using tensors. Algebraically, a tensor is represented as a sum of scalar (w_i) weighted basis vectors (v_i) as shown in Fig. 2.6. The figure shows text fragments from two documents and their corresponding tensor representation (Tensors D^1 & D^2). Each basis vector denotes elementary meaning which typically is a term or a phrase (character strings) from a controlled vocabulary/dictionary or selectively picked from a text object (e.g. a sentence in Fig. 2.6) and assigned weights (scalar coefficients) depending on the model used [6].

The similarity between the meanings represented by the two vectors is given by their cosine (dot) product. The dot product of two vectors/tensors is given by the sum of products of weights of the basis vectors (having non-zero weights) that are common in both vectors. In a computer, these tensors can be represented by a table of character strings and their corresponding coefficient weights. Computing dot-product of large vectors is challenging. In the next section, we shall explain the challenge.

Text doc D^1: "The soccer player looked at the ball. Then he kicked it."	Text doc D^2: "The soccer player kicked the ball."
$TensorD^1 = w^A_{soccerplayer}\overrightarrow{v}_{soccerplayer} + w^A_{ball}\overrightarrow{v}_{ball} + w^A_{look}\overrightarrow{v}_{look} + w^A_{kick}\overrightarrow{v}_{kick}$	$TensorD^2 = w^B_{soccerplayer}\overrightarrow{v}_{soccerplayer} + w^B_{kick}\overrightarrow{v}_{kick} + w^B_{ball}\overrightarrow{v}_{ball}$

$$Semantic\ Similarity < D^1, D^2 >= D^1 \cdot D^2 = w^A_{soccerplayer}w^B_{soccerplayer} + w^A_{ball}w^B_{ball} + w^A_{kick}w^B_{kick}$$

Fig. 2.6. Tensor Model of Semantic Comparison

2.4 Challenge in Dot Product Computation

Dot product is the sum of product of corresponding non-zero coefficients of two vectors. The key challenge is to pair-up the corresponding vectors to enable the appropriate multiplications. This pairing up requires that each component (basis) vector (from one of the vectors being processed) is checked for a corresponding entry in the other vector. If a corresponding entry is found, then the coefficients of the two are multiplied. If a corresponding entry is not found, this is the same as multiplying that particular coefficient by zero - a superfluous operation. This is a relatively simple task when dealing with small finite vector models (such as in Latent Semantic Indexing [24]) since few superfluous multiplication operations does not increase the computational cost extensively. However, for vector models that deal with infinite dimensional vector space - the superfluous multiplications quickly become computationally expensive and hence need to be eliminated.

If the number of basis vectors in each vector being compared are denoted as n_1 and n_2, then using the traditional computing approach, the search operation involved in dot-product takes $n_1 \cdot log(n_2)$ or $n_1 \cdot n_2$ computations depending on whether or not a binary search tree is created. Note that creating a binary search tree also involves sort operation on one of the vectors - which is also of the order of $n \cdot log(n)$ where

n is the size of the vector being sorted. Hence in the case of a binary search tree based approach, the order of complexity of the total computation is $n_1 \cdot log(n_2) + n_1 \cdot log(n_1)$ (assuming that n_1 is the number of rows in the table getting sorted)

In order to perform this search elegantly (with less complexity) Bloom Filters are used. Bloom filters allow one to perform this search (with a small possibility of false positives) with much less computational complexity $O(1)$. An effective representation of composite meanings is necessary to ensure "meaningful" semantic comparison. To enable this, semantic comparison operates on an infinite dimensional vector space leading to large tensor sizes ($\approx 10^3$).

We briefly describe the basic properties of the Bloom Filter that we used in the next section.

2.5 Bloom Filter Basics

A Bloom filter (BF) is a compact representation of a set [25]. The BF consists of a large single dimensional array of m bits and a set of k hash functions and is a good candidate for performing membership tests on a set. A Bloom filter has the property that while it may present a false positive (indicate that the element is present, while it is actually not present), the BF will never have a false negative (i.e. indicate that the element is not present, when it is actually present). There are two operations that can be performed on a BF - Insertion and Membership testing. We will now briefly describe each of these operations.

2.5.1 Element Insertion

To insert an element (such as a number or a text string) in this set, we hash this element to generate k different values using the k different hash functions. We use these values as bit-indices to decide which bits in the array should be set to 1 as shown in Fig. 2.7.

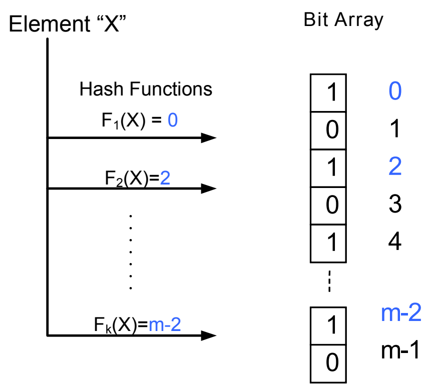


Fig. 2.7. Bloom Filter : Insertion

2.5.2 Membership Testing

To test whether an arbitrary element is in the BF, we similarly generate k bit indices and check whether all of those bits are 1 or not. All bits being 1 indicates that the element is in the set (Fig. 2.8).

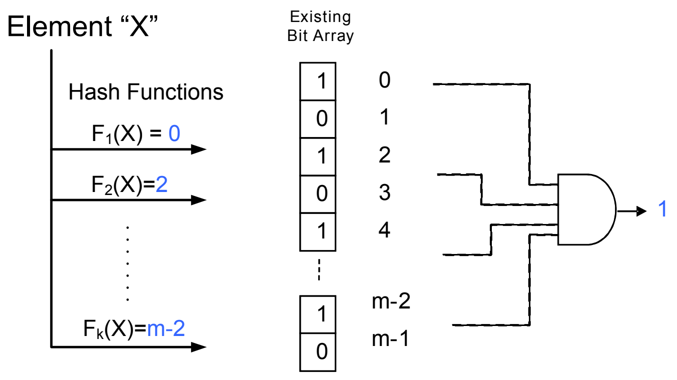


Fig. 2.8. Bloom Filter : Membership Testing

Because testing for presence is performed by looking at the bit-position values, it is possible that a bit that was set due to the insertion of one element, may be used in the testing process for testing for the presence of another element. This gives rise to the false positive rate of the Bloom Filter. However, since there is no possibility that an element that was inserted would not have set the appropriate bit-positions, Bloom Filters will not present a false negative.

The probability of a false positive $P_{false+ve}$is given by:

$$P_{false+ve} = \left(1 - \left[1 - \frac{1}{n}\right]^{km}\right)^{k} \tag{2.1}$$

which can be simplified as

$$P_{false+ve} = \left(1 - e^{-\frac{kn}{m}}\right) \tag{2.2}$$

The probability of false positives, can be minimized by choosing large m, and optimum $k(\approx 0.7 * m/n)$, where m is the number of bits in the BF, k is the number of independent hash functions and n is number of elements in the BF (Fig. 2.8). For example, a basic BF with $m = 10240$ bits, $k = 7$ can keep 10^3 elements with a small $P_{false+ve} \approx 8 \times 10^{-3}$, which will have a negligible effect on similarity comparison.

The basic bloom filter does not provide for deletion of an element or for the insertion of duplicate elements. However, there exists, other variations of bloom filters such as Counting Bloom Filters [26] proposed by Li et al. In [27], Bonomi et al. proposed a *d-left hash* based design that allows for a more compact (space efficient) design of counting filters.

In [28] the Chazelle et al. implemented *Bloomier Filters*, an extension to Bloom filters that allows the implementation of an associative array. The basic bloom filter approach cannot adapt to varying number of elements stored without re-creating the entire filter. This problem was addressed by [29] who proposed Scalable Bloom filters. Scalable Bloom filters are an adaptation of the classic bloom filter design, but in this case can dynamically adapt to the number of elements being stored without affecting the false positive probability.

2.6 FNV Algorithm

2.6.1 Overview

The Fowler / Noll / Vo (FNV) hash algorithm [30] is used as the basic Hash Function in our architecture. It was chosen because of its ease of parallelization compared to more popular algorithms such as MD5 [31] or SHA2 [32] . The algorithm for the computation of the FNV hash is presented in Algorithm 1. The implementation details of this algorithm will be presented in the next chapter.

2.6.2 Algorithm & Relevant Parameters

Algorithm 1
FNV Hash Algorithm (Type 1a)

```
procedure FNV-1A($data$)
    $hash \leftarrow$ offset_basis
    for all $octets\_of\_data$ do
        $hash \leftarrow hash \otimes octet\_of\_data$
        $hash \leftarrow hash \times FNV_{prime}$
    end for
end procedure
```

The parameters of the FNV Hash of size n are:

- *hash* is an n-bit unsigned integer.
- The multiplication operation (denoted by $\times$) is performed modulo 2^n.
- The XOR operation (denoted by $\otimes$) is performed on the lower octet (8 bits) of *hash*.
- FNV_{prime} is dependent on n.
- offset_basis is dependent on n.

2.6.3 Standard Values of Key Parameters

Values for FNV_{prime} and offset_basis for different sizes of n are available at [30]. In this dissertation, the value of FNV_{prime} and offset_basis for a size of $n = 64$ bits is taken. Some representative values for FNV_{prime} are shown in Table- 2.1 and representative values for offset_basis are shown in Table- 2.2

Table 2.1
Values of FNV_{prime} for Different Sizes of n

Size of n (bits)	Value of FNV_{prime}
32	$2^{24} + 2^8 + 0x93$
64	$\mathbf{2^{40} + 2^8 + 0xb3}$
128	$2^{88} + 2^8 + 0x3b$
256	$2^{168} + 2^8 + 0x63$

Table 2.2
Values of offset_basis for Different Sizes of n

Size of n (bits)	Value of offset_basis
32	2166136261
64	**14695981039346656037**
128	144066263297769815596495629667062367629

The value of offset_basis is the FNV-0 hash of the following string (shown in C style notation).

``‘‘chongo <Landon Curt Noll> /\\../\\’’``

This text segment is the original email signature line of one of the creators of the FNV algorithm [30]

3. OVERVIEW OF SEMANTIC COMPARISON

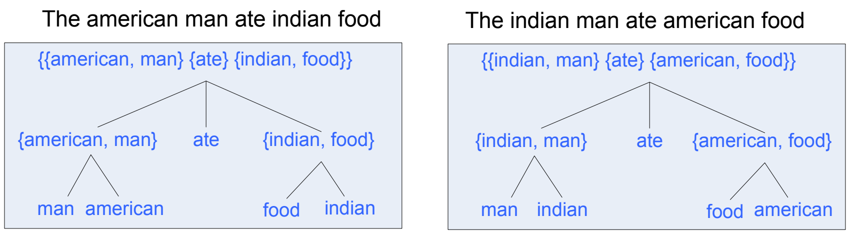
Semantic comparison as performed by a semantic router compares two objects. As explained previously in Section 2.2.1, These objects may be data or services having various granularities, formats, structures, service access/delivery mechanisms. These semantic objects are annotated using *metadata* that describes the resource. This metadata can be based on generic or upper ontologies, like Wordnet, Gene Ontology or more specialized ontologies / standards like the Unified Medical Language System (ULMS).

The metadata describing these objects, is used to create a semantic descriptor - a data structure that contains the semantic description of the object, by applying techniques such as TF-IDF onto the metadata. By comparing the semantic descriptors of two objects, we can compute the semantic similarity between the two.

3.1 Semantic Descriptor

A semantic descriptor can be generated in several ways. The most commonly used technique TF-IDF [6] stores a statistical product term (frequency $\times$ inverse document frequency) for all terms in a document. In contrast [33] proposes and evaluates a weighted Concept (ontology) Tree based descriptor taking compositions into account. A concept tree is a hierarchical acyclic directed $n-ary$ tree where the leaf nodes represent terms whereas the tree itself describes their inter-relationships within a document [34] [35]. Each term is assigned a weight which describes its relative importance. Fig. 3.1 shows two sample trees which represent two distinct "concepts" but use the same keywords (shown as leaf nodes). The intermediate notations (ex. American, man) are notional and are shown for convenience.

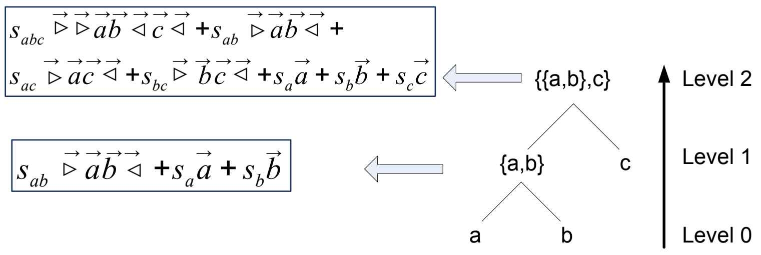
Associated weights for each term (coefficient values describe their relative importance) are not shown for simplicity. Concept tree building has been discussed in detail in [8] [12] and applied to test cases in [36]. It is important to note that



The american man ate indian food
{{american, man} {ate} {indian, food}}
{american, man}
ate
{indian, food}
man
american
food
indian
The indian man ate american food
{{indian, man} {ate} {american, food}}
{indian, man}
ate
{american, food}
man
indian
food
american

Fig. 3.1. Concept Trees of Two Sentences

conventional term based weighting cannot differentiate between the two statements (or their trees). A Tensor model, on the other hand, is used to decompose such concept trees into a flat data structure consisting of polyadic concept terms and their normalized coefficients without loss of the semantics contained in the original tree structure.

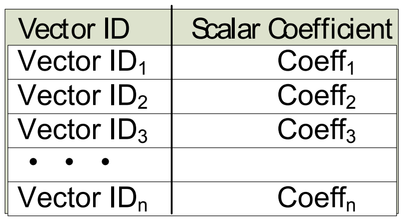


$s_{abc} \triangleright \triangleright \vec{a}\vec{b} \triangleleft \vec{c} \triangleleft + s_{ab} \triangleright \vec{a}\vec{b} \triangleleft +$
$s_{ac} \triangleright \vec{a}\vec{c} \triangleleft + s_{bc} \triangleright \vec{b}\vec{c} \triangleleft + s_a \vec{a} + s_b \vec{b} + s_c \vec{c}$
$s_{ab} \triangleright \vec{a}\vec{b} \triangleleft + s_a \vec{a} + s_b \vec{b}$
{{a,b},c}
{a,b}
c
a
b
Level 2
Level 1
Level 0

Fig. 3.2. Conversion of Concept Tree into Tensor Representation

Fig. 3.2 shows the bottom up expansion of a 3-level, 2 child concept tree. Level 0 shows leaf nodes “a” and “b”. Their composition at Level 1a,b is defined by $\triangleright ab \triangleleft$, a and b ($\triangleright$&$\triangleleft$ are delimiters; s_{ab}, s_a, s_b are normalized weights). The final tensor representation of this concept tree is obtained at Level 2 consisting of weighted polyadic combinations of terms $\triangleright ab \triangleleft c$, $\triangleright ab \triangleleft$, $\triangleright ac \triangleleft$, $\triangleright bc \triangleleft$, a, b, c (called basis vectors) and their normalized weights. The nuances of this conversion including the process of determination of weights is detailed in [37].

The semantic descriptor created by the tensor conversion process comprises of the tensor representation as well as the associated scalar coefficient value as shown in Fig. 3.3.



Vector ID	Scalar Coefficient
Vector ID_1	$Coeff_1$
Vector ID_2	$Coeff_2$
Vector ID_3	$Coeff_3$
• • •	
Vector ID_n	$Coeff_n$

Fig. 3.3. Tabular Form of Semantic Descriptor

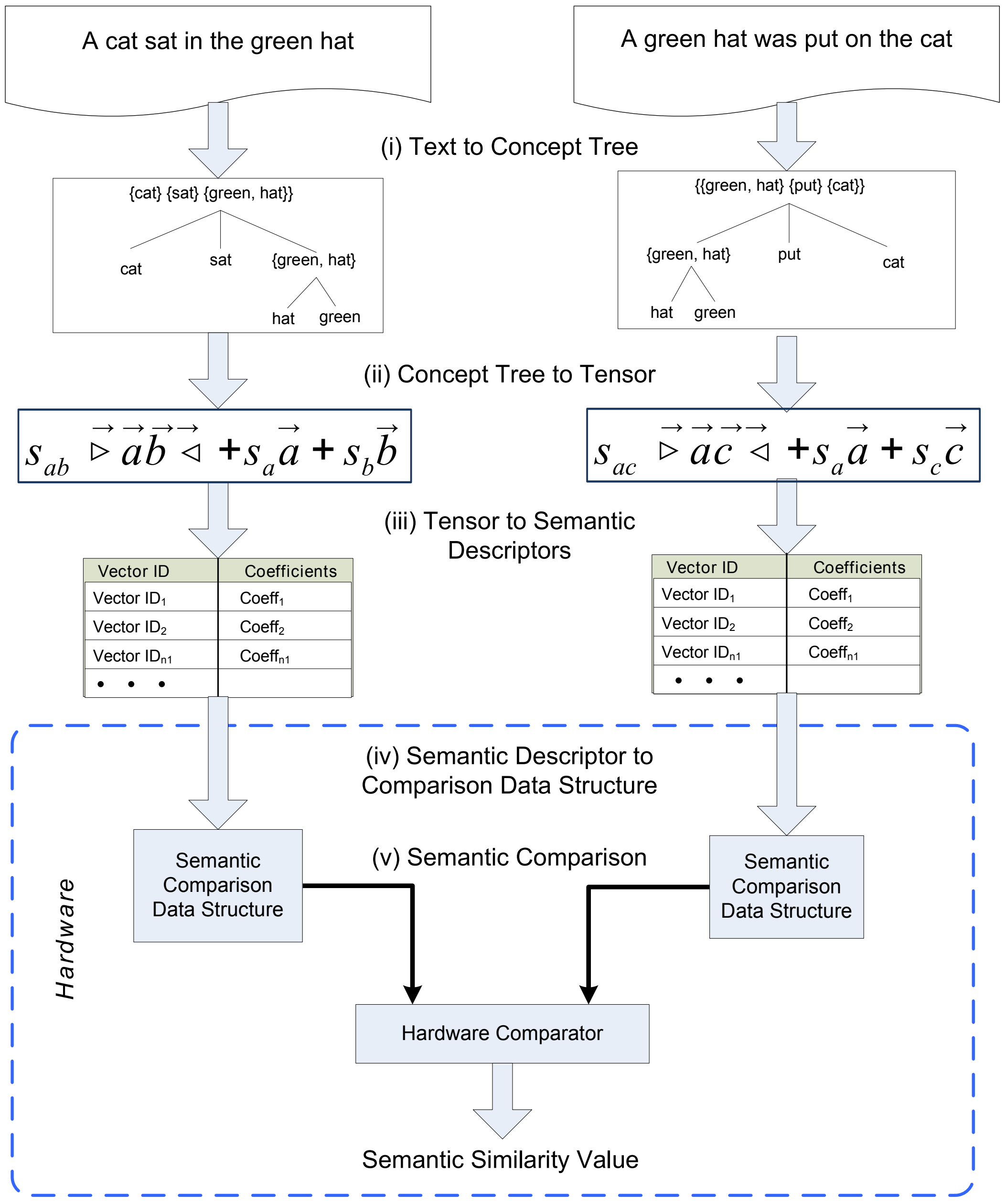
3.2 Semantic Comparison

The basic comparison occurs between two descriptors D1 and D2 as follows. The task is to compute the dot-product between these two descriptors to get a numerical value for the semantic similarity. Fig. 3.4 shows the sequence of steps performed to obtain the similarity value.

The semantic comparison process begins by converting the semantic descriptors discussed above into appropriate data-structures that can be used in the computation stage. Once the conversion is complete, the computation of semantic similarity is performed using the method of cosine similarity (dot-product). For the rest of the dissertation, we will use the terms tensors and vectors interchangeably.

The major steps of computing semantic similarity are as follows

1. Population of Data Structure.
    (a) Computation of FNV Hash.
    (b) Population of Bloom filters.
2. Computation of Cosine Similarity.



A cat sat in the green hat
A green hat was put on the cat
(i) Text to Concept Tree
{cat} {sat} {green, hat}}
cat
sat
{green, hat}
hat
green
{{green, hat} {put} {cat}}
{green, hat}
put
cat
hat
green
(ii) Concept Tree to Tensor
$s_{ab} \triangleright \vec{a}\vec{b} \triangleleft + s_a\vec{a} + s_b\vec{b}$
$s_{ac} \triangleright \vec{a}\vec{c} \triangleleft + s_a\vec{a} + s_c\vec{c}$
(iii) Tensor to Semantic Descriptors
Vector ID
Coefficients
Vector ID1
Coeff1
Vector ID2
Coeff2
Vector IDn1
Coeffn1
(iv) Semantic Descriptor to Comparison Data Structure
Semantic Comparison Data Structure
(v) Semantic Comparison
Semantic Comparison Data Structure
Hardware
Hardware Comparator
Semantic Similarity Value

Fig. 3.4. Sequence of Steps for Semantic Comparison

(a) Identification of Common basis vectors.

(b) Extraction of Coefficients.

(c) Multiplication and Summation of Coefficients.

In the following sections we will discuss each of these steps in detail.

In this dissertation, we focus on Steps (iv) & Step (v) from Fig. 3.4. We presume that Step (i) (conversion of query/object to a concept tree) through Step (iii) (conversion from concept tree to descriptor) has been completed. The SRN will routes packets of data, each packet containing a semantic descriptor.

3.3 Architectural Overview of Semantic Comparator

The proposed semantic processor core, has five basic stages as shown in Fig. 3.5. Stage (A) generates vector IDs and k hash values for each basis vector; Stage (B) populates the Bloom filter of descriptor D2 using the k hash values; Stage (C) identifies the common basis vectors using BF membership testing; Stage (D) extracts the matching pairs of scalar coefficients using Content Addressable Memory (CAM) modules; and Stage (E) multiplies the corresponding pairs of scalar coefficients and calculates the sum to obtain the dot product.

Each of these parallel processing instances, which can be considered as a hardware thread, is executed by each horizontal slice of each circuit stage (A to E), as shown in Fig. 3.5. There are n slices in stage A to C, and b slices in stage-D and p slices in stage-E. Stage-E also consolidates all the processing.

3.3.1 Working

The five stages of the semantic comparator execute the steps listed in Section 3.2. The input descriptors to be compared (D1,D2) are first converted into the respective semantic comparison data structures. This is done by hashing the input

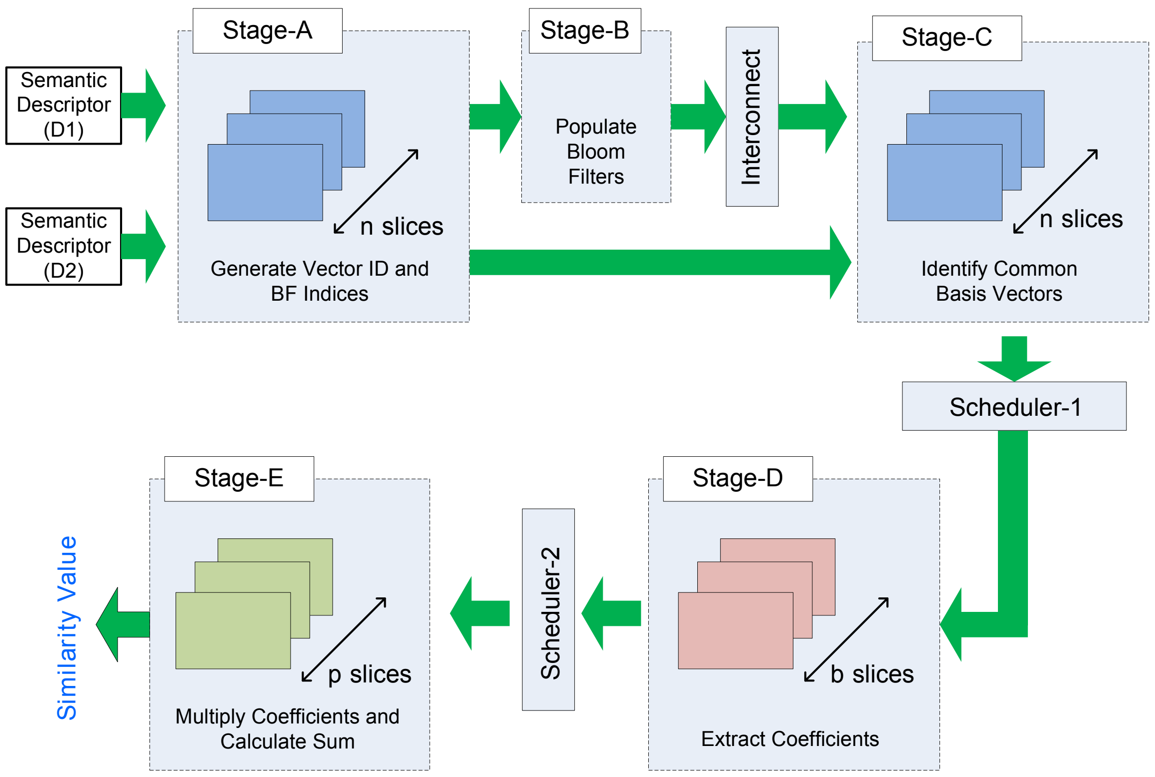


Fig. 3.5. Overall Stages of Processing

vector IDs through a common hash function (FNV hash), and then obtaining the Bloom Filter Indices and storing them. Once both of the descriptors have been converted into the semantic comparison data structure, the actual processing of the semantic comparison begins. To begin this step, the bloom filters are consolidated into a single bloom filter and distributed to the n slices of Stage-A. Each slice of Stage-A gets one of the appropriate rows from the second data-structure. Once the Bloom filter has been consolidated, Stage-C locates the common basis vectors. This identifies which of the basis vectors in the two descriptors being compared are the same. The order of computation for Stage-C because we use n parallel stages and k indices for the bloom filter is $O(k)$. Once the common vectors have been identified, these vectors are passed to Stage-D where the CAM blocks retrieve the corresponding scalar coefficient values. The pairs of scalar coefficient values that need to be multiplied are then sent to Stage-E where a bank of multipliers multiply the values and feed it to an accumulator. This stage (Stage-E) is where the actual computation of dot-product (Cosine similarity) occurs. The output of Stage-E is the similarity value between the two stages. The working of each of the stages A-E is discussed in detail in the next two sections.

4. CREATION OF REQUIRED DATA STRUCTURES

In this section, we present the first of the two major steps (Generation of Data structures, Stages A,B in Fig. 4.1). In the next section (Section. 5) we will describe the architectural details of steps C-E.

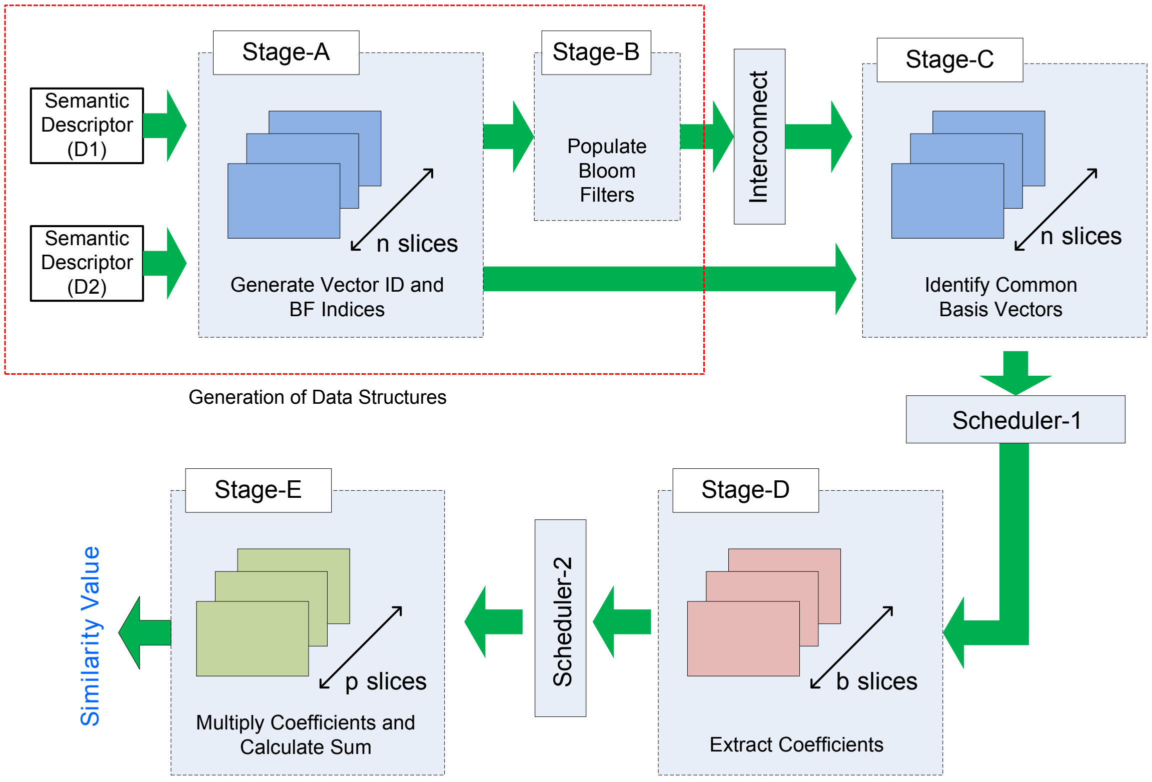


Fig. 4.1. Stages Involved in Generating Data Structures

4.1 Description of the Data Structures

The abstract data structure required for semantic comparison has two components: (1) a Co-efficient table and (2) a large m ($\approx 128K$) bit long Bloom Filter (BF) using $k(=7)$ hash functions [25]. The layout of the data structure is shown in Fig. 4.2 Each row of the coefficient table consists of three columns: (1) Vector ID (e.g. "ID1" in Fig. 4.2); (2) 16 bit fixed point scalar coefficient of the corresponding basis vector (e.g. w_i); and (3) Set of BF indices ($x_1 : 0 \leq x_i \leq m$). The combination of the coefficient table and the bloom filter represents the vector/tensor representing the meaning of an object (text/non text document).

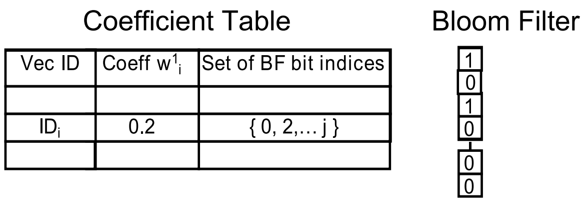


Fig. 4.2. Components of Data Structure

A 64 or 128 bit hash of a basis vector term is generated and inserted as the Vector ID in the coefficient table as shown in Fig. 4.3. To generate a set of k BF indices, each Vector ID (or the basis vector character string) is further hashed by k hash functions and the resultant indices are stored in the third column while the corresponding bit locations are set in the BF.

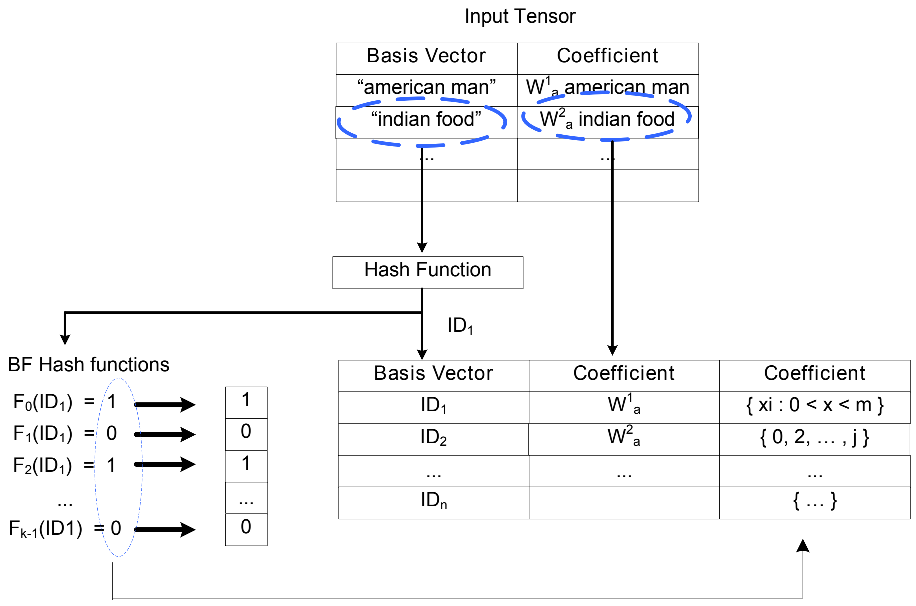


Fig. 4.3. Populating the Data Structure

4.2 Algorithm to Populate the Data Structures

In Fig. 4.4, we present the block schematic of the architecture where the BF indices are generated. We use the FNV hash algorithm as the primary hash for the BF index generation operation. The 64 bit output of the FNV hash module is duplicated. The first copy (f_1) is directly truncated to 17 bits using XOR folding [30]. The second copy (f_2) is rotated by 33 bits and then truncated to 17 bits using XOR folding. For each BF index BF_i (where $0 \leq i \leq k$); a copy of f_1 is rotated by i bits and then XOR'd with f_2. (For example, the second BF index (index-1 in Fig. 4.4) is obtained by first rotating f_1 by one-bit position and then XOR-ing this with f_2.) This enables the creation of k different index values in parallel.

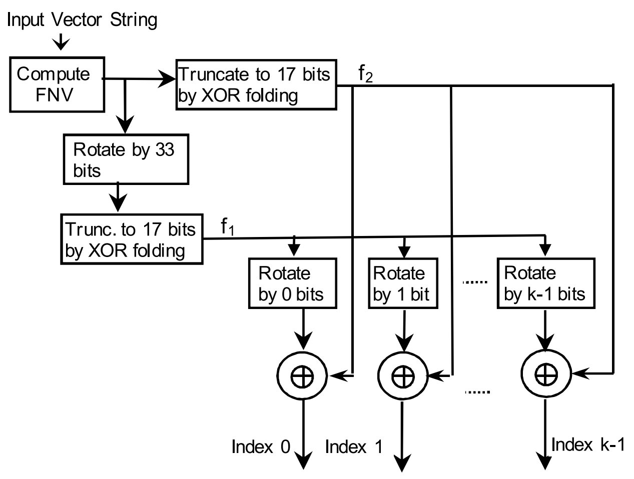


Fig. 4.4. Bloom Filter Index Generation

These bloom indices allow parallel setting of the corresponding bit position in a memory element, hence creating a Bloom Filter for each row. The n individual BFs (one per row) are then consolidated into a single BF using cascaded OR-Gates and distributed to multiple rows.

4.2.1 Consolidation of Bloom Filters

Since we are going to be using n copies of the circuit for Stage-C, we need to consolidate the bloom filters created in Stage-B and then distribute them to the n copies of Stage-C. The distribution is done using the Interconnect shown in Fig. 4.1. The interconnect is a simple tree network similar to the commonly accepted clock-tree implementations.

The n individual BFs (one per row) are consolidated into a single BF using cascaded OR-Gates. Fig. 4.5 shows the completely parallel architecture of doing this. It is also possible to do this in multiple-cycles by using flip-flops at intermediate stages. The use of multi-cycle allows for decreased chances of hold/setup time violations due to the longest path length.

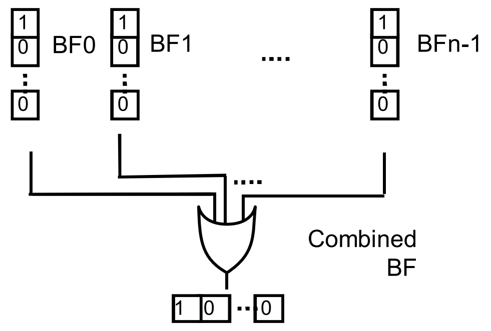


Fig. 4.5. Bloom Filter Consolidation

4.3 Results

4.3.1 Analysis of Hash functions Used for BF Index Generation

A Bloom Filter requires the use of k hash functions to generate k index values. The standard method of doing this is to deploy k separate hash functions for the purpose. However, deploying k hash functions in hardware is not efficient. A more efficient manner to obtain a number of hash values is to combine two hash functions

$h1(x)$ and $h2(x)$ in the form $g_i(x) = h_1(x) + ih_2(x)$. In [38], the authors show that applying this technique to Bloom Filters allows you to efficiently implement a BF without any loss in the false positive probability. In order to locate the most power efficient manner generating hash functions, we looked at several different ways of combining $h1(x)$ and $h2(x)$ In Table 4.1 we present some of the different ways we could combine the hash functions to get the values. In the table, $\otimes$ represents bitwise XOR operation, $*$ represents a multiplication operation and $+$ represents addition.

Table 4.1
Some Possible Variations of Hash Functions

Method	Combination	Power
1	$A + rot(B, i)$	557.103 μ W
2	$A + i * B$	88.136 μ W
3	$A + 2^i * B$	637.910 μ W
4	$A \otimes i * B$	58.269 μ W
5	$\mathbf{A \otimes rot(B, i)}$	**61.258** μ W
6	$A \otimes 2^i * B$	92. 796 μ W

The multiplication operation (denoted by $*$) can be implemented using a *shift* operation in Verilog. However, *shifting* bits, introduces Zeros into the bit positions vacated during the shift operation. Since performing an XOR operation with 0 retains its value ($1 \otimes 0 = 1$ and $0 \otimes 0 = 0$), this would lead to those bit positions being deterministic (The location of the zeros are known). This leads to a sub-optimal value for the Hash function. Hence we did not choose any of the multiplication options. Thus, the most optimal choice of the options was Method-5 from the table above i.e. $A \otimes rot(B, i)$. This gave us the lowest power draw while retaining the randomness due to the rotation.

During the population of the data structure, the computation of the FNV hash can be done either completely in parallel or in multiple cycles. In table 4.2 we present the power and timing differences between completely parallel computation and sequential computation. The module that generates the k hash indices is made of purely combinational logic - hence it is always a single cycle module.

4.3.2 Power Draw by Stages-A & B

The proposed design implemented in Verilog and simulated using ModelSim from Mentor Graphics. To obtain power results, synthesis was performed on the verified design using Design Compiler from Synopsys using components from the DesignWare IP Library and the TSMC 90nm technology library. The memory power data was obtained using the CACTI power model [39]. The feasibility of the design is established by the synthesized power estimate, i.e. if power is within physical limits and the circuit is synthesizable, then it is feasible.

Table 4.2
Power Draw by One Instance of the FNV module

Module	Power	Num Clock Cycles
FNV - Parallel Implementation	25.3 mW	1
FNV - Sequential Implementation	5.97 mW	6

As can be seen from the data. A trade-off must be made between power and timing, depending on the performance goals. We took the sequential approach in order to keep the power draw low. However, if the power budget allows, the single-cycle approach can give a performance improvement.

Table 4.3 shows the power draw for the two stages (A,B). Stage-A power draw comprises of the FNV module and Address-generation modules. (See Fig. 4.4) and Stage-B comprises of the Bloom Filter Consolidation and distribution network (See Fig. 4.5).

Table 4.3
Overall Power Draw by Stages A & B

Module	Total Power draw
Stage-A	6.22 W
Stage-B	1.66 W
Total	7.88 W

4.3.3 Timing Calculations

To generate the FNV hash and the k BF indices for a single basis-vector, takes a time of $O(1) = (t_{FNV}) + O(k)$. For total n ($< 10^3$) basis vectors in each table, the order of the entire data structure generation computation is $O(n_1 \cdot k)$. As computation of each basis vector is independent, each of these can be computed in parallel using $r(\approx n_1)$ circuits within $O(n_1 \cdot k/r)$ time. For $r << n$, this is $O(k)$ with k generally < 20 [25]. Consolidation of the BFs into a single BF is a constant time operation because it is not dependent on n. The analysis of timing across all the stages will be discussed in detail at the end of the next chapter.

Since the two input tables have different processing requirements, the time taken for this stage (t_1) can be denoted as a sum of computation time for Table-1 (t_{1a}) and computation time for Table-2 (t_{1b})

$$t_1 = t_{1a} + t_{1b} \tag{4.1}$$

$$t_{1a} = t_{FNV} + W + O + D \tag{4.2}$$

$$t_{1b} = t_{FNV} + W \tag{4.3}$$

Here, t_{FNV} is the number of cycles taken to process each input string into its corresponding hash value (using the FNV hash algorithm), W is the time taken to write the BF indices. (common to both tables) As explained previously, n individual copies of the BFs need to be consolidated into a single BF prior to distribution, hence O is the time taken to OR together the individual bloom filters. (It should be obvious that only Table-1 needs to undergo this step). D is the time taken to load the multiple copies of the BFs to the corresponding slices in Stage-C (common to both tables). In our design $W = k$, and $O = D = 1$. Assuming a 40 character input string for each basis vector being processed, this stage would take 96 clock cycles to process the data in parallel. The limiting factor is the number of octets that need to be processed by the FNV algorithm block to get the initial hash value in order to generate f_1 and f_2. Once these are generated, generating the k index values and the population and distribution can be carried out quickly. While we have attempted to parallelize the FNV computation, the requirement to feedback the current value of the hash for ever octet of the input data is a part of sequential computation that cannot be completely parallelized..

5. COMPUTATION OF COSINE SIMILARITY

5.1 Introduction

Computation of cosine similarity, requires performing the dot-product operation between the two data structures being compared. As mentioned previously in section 2.4, identifying the common basis vectors is computationally expensive. If we can reduce the computational complexity, we can obtain power and energy savings. In this section, we present an overview of the parallel algorithm and the details of its implementation. We will look at stages C to E of Fig. 5.1 as well as the schedulers.

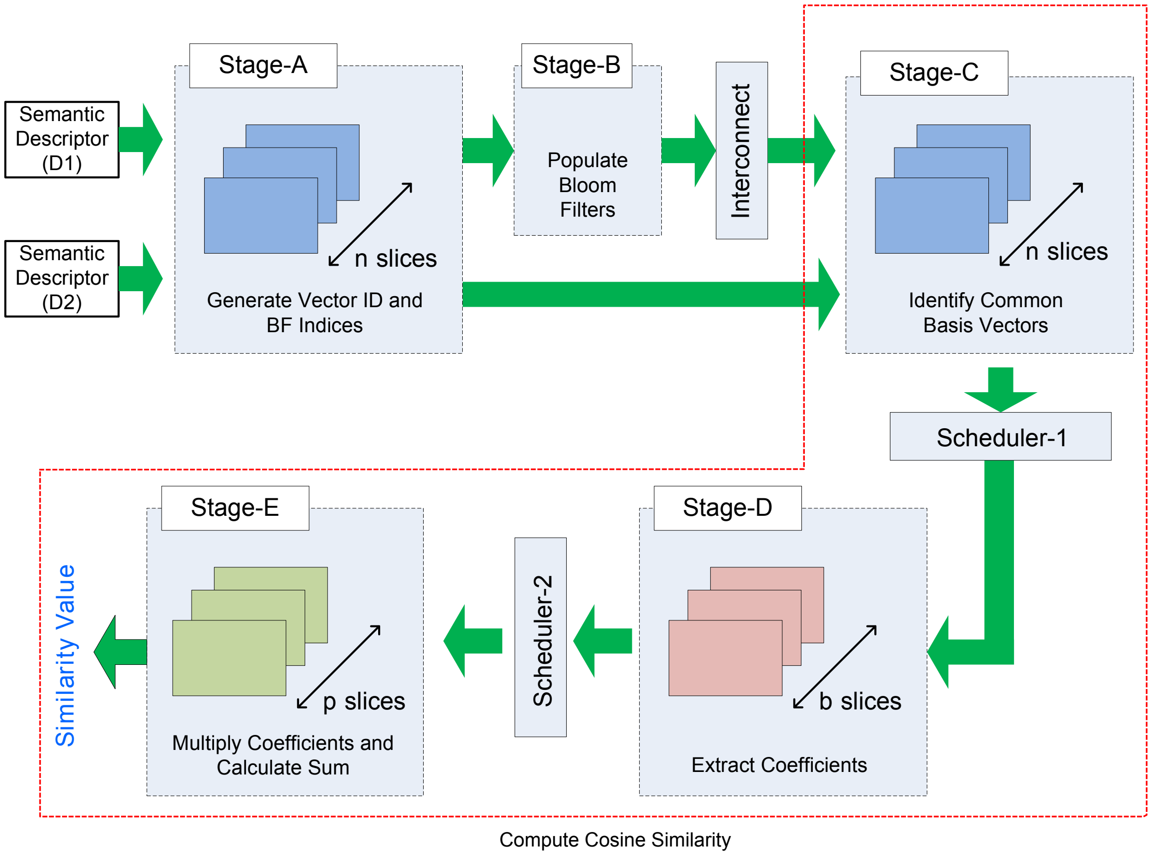


Fig. 5.1. Stages Involved in Cosine Similarity Computation

5.2 Overview of the Parallel Algorithm

The process to perform the computation of cosine similarity can be broken down into the following steps:

1. Identification of Common Basis Vectors.
2. Extraction of Coefficients.
3. Multiplication and Summation of Coefficients.

In Fig. 5.2 we present the graphical version of the steps involved. Two data structures (D1 and D2) are taken as input to generate the cosine similarity value ($D1 \cdot D2$) as follows:

(a) Identify the common basis vectors from the first coefficient table (Component 1,Fig 5.2) by verifying which vector IDs are in the second BF (BF2) by using the set of BF bit indices in the first table.

(b) If a vector is present in the BF2 then we use that common vector ID as the key, and extract the coefficient value from the coefficient lookup table of the second data structure (Coefficient Table of D2). This is carried by a content addressable memory (CAM) lookup mechanism with vector ID as the key.

(c) Multiply the pair of coefficients for each identified common basis vectors (from both tables), and

(d) Add all the products to get the similarity metric.

5.3 Overview of Hardware Implementation

In [12] we showed that in a semantic routed network, c (the number of common vectors) is $\frac{1}{1000} \times n = 0.01\%$ of n. (n is the number of rows in a coefficient table

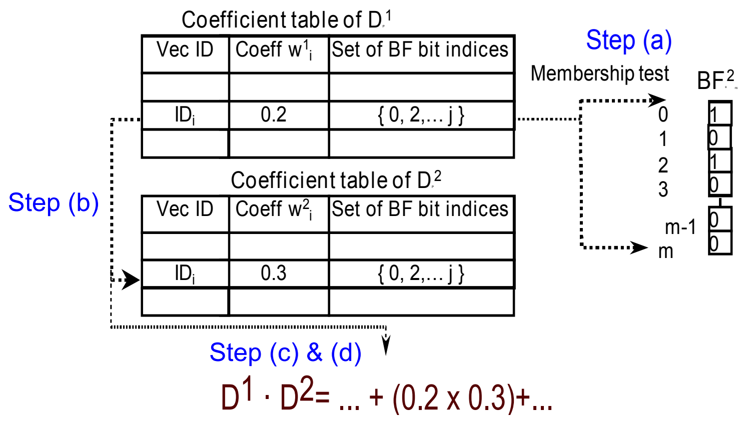


Fig. 5.2. Steps of Computing Cosine Similarity

i.e. number of basis vectors). The expected number of CAM lookups that will take place is given by $(c+(n-c)\cdot P_{false+ve})$, where $P_{false+ve}$ ($\approx 10^{-9}$) is the probability of false positive for the Bloom filter being used ($m \approx 10^5, k \approx 7$) [25]. Therefore only a small number of lookups (in CAM) (0.01% of n) will occur. Hence we use a small number of slices in stage D ($b << n$). This reduces the design requirement from n-CAM blocks to b-CAM blocks and saves power. Further, as multiplier units are expensive in terms of power and area requirements, we use an even smaller number ($p < b, p << n$) of slices in the multiplication stage E.

5.4 Common Vector Identification

Fig. 5.3 shows the design of the block that identifies the common basis vectors amongst the two data structures. BF Indices (e_{1j}) for rows r_1 and r_2 are shown in the figure. There are n such rows present to enable parallel comparisons. Each row has a copy of BF_2 (shown by b1 , b2) allowing for parallel testing. Each entry e_{ij} in a row r_i of the table is fed to a common address bus by means of a tri-state buffer.

A decoder connected to a *modulo-k* counter enables the tri-state buffers. By clocking the counter for k cycles, the k bloom filter index values can be sequentially

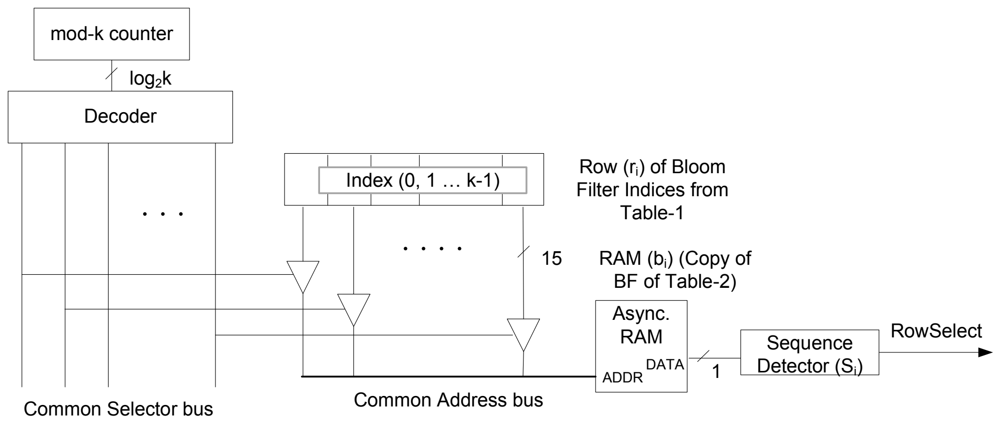


Fig. 5.3. Identification of Common Basis Vectors

fed to the bank b_i of RAM containing BF_2. The DATA port of the RAM bank b_i feeds into a sequence detector s_i. The sequence detector sets the RowSelect line to high when a sequence of k 1s has been observed (indicating that all BF indices e_{ij} of row r_i are present). This is done as part of the Bloom filter (BF) membership testing operation, where we check to see if all selected k bit positions are 1 or not.

Use of asynchronous RAM along with tri-state buffers ensures a fewer number of lines to route. This results in a lower power and area solution. The use of Bloom filters, allows us to identify the common rows in the k cycles. This is much quicker than the $O(n_1 \cdot logn_2)$ or $O(n_1 \cdot n_2)$ computation required for the corresponding software approach.

5.5 Extraction of Coefficients

Once the common basis vector rows are identified, the next step is to extract the scalar coefficients. The architecture of the block that performs the extraction of scalar coefficients from one row is shown in 5.4. The RowSelect signal from the previous stage is used to drive the SearchEnable input to the CAM. When the match is found by the CAM, the CAM asserts the MatchFound signal and sends the address of the match to the RAM block.

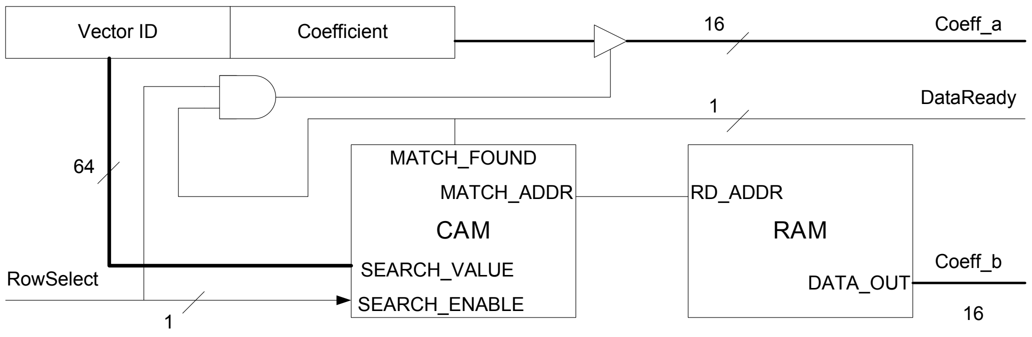


Fig. 5.4. Extraction of Coefficients

The RAM block outputs the corresponding coefficient value (Coeff_b). The match found signal and the RowSelect signal are used to gate the values of the first coefficient (Coeff_a) to the next stage of the process. The MatchFound signal is also used as the DataReady signal for the next stage of the computation process. A set of Coeff_a, Coeff_b and DataReady signals are obtained for every row that is part of the common basis vector set.

5.6 Multiplication and Summation

Once the corresponding pairs of coefficients are obtained, the final step is to multiply each pair of scalar coefficients with each other and calculate the sum of all the product terms.

In an ideal case, with n rows of data, n multipliers and an $n-$input adder would guarantee minimum latency. However, since multipliers are expensive both in terms of power and area, we chose to make a trade-off and use a lower number of multipliers. Fixed point representation allows us to use pipelined multipliers, which are also more power and area efficient than floating point units. The primary challenge in using a smaller number of pipelined multipliers is in scheduling the processing so that a new multiply operation can be started on every multiplier in each cycle. Fig. 5.5 shows the block diagram of the multiplication and summation stage. Fig 5.5 also shows the location of Scheduler-2. We discuss the scheduling logic in the next section.

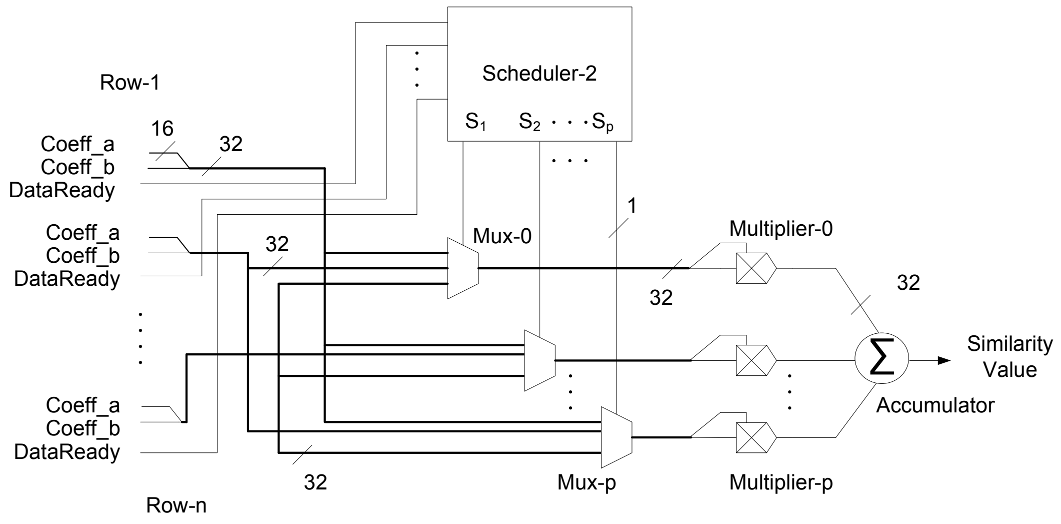


Fig. 5.5. Multiplication and Summation.

If c pairs of coefficients need to be multiplied, p is the total number of multipliers available, L is the latency of the pipelined multipliers, and A is the latency of the adder, then the overall latency of this stage is $(c/p) + L + A$ cycles.

The outputs of the multipliers are fed to a single cycle accumulator that can handle p inputs. Thus at the end of $(c/p) + L + 1$ cycles, the value that is output by the accumulator is the similarity value.

5.7 Scheduling Logic

To facilitate smooth flow of data between the unequal number of slices in different stages and enable maximum utilization of slices, special interconnects are used between the stages. The Interconnect between stage B and C (see Fig. 5.1) distributes the consolidated Bloom filter to n slices in stage C. This is a simple distribution network similar to a Clock-Tree.

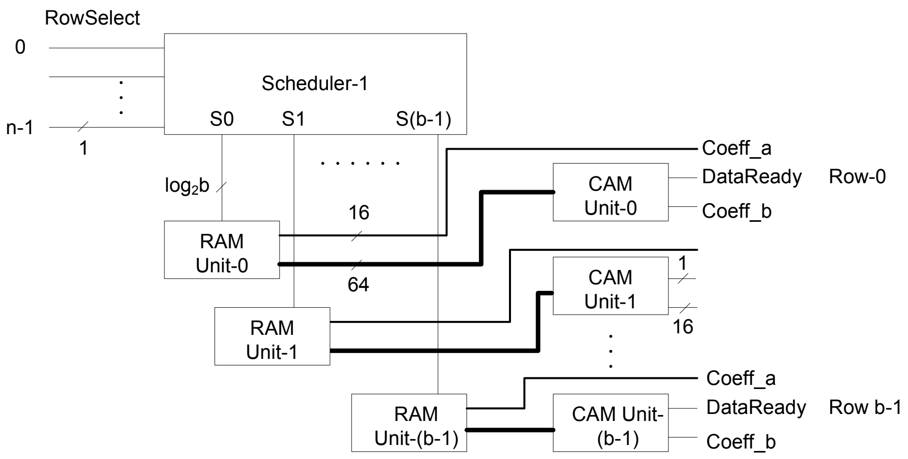


Fig. 5.6. Location of Scheduler-1 between Stages C & D

Scheduler-1 between stages C and D schedules the parallel extraction of coefficient values. This interconnect operates on n one-bit signals from Stage-C (RowSelect) that indicates that the corresponding row is a potential candidate match. RAM units shown in 5.4 contain the coefficient values of Table 1 (b-RAM copies are present in total). The b-CAM units on the other hand store copies of the coefficient values of Table 2. Using b -CAM units instead of n allows us to economize on both power and area both of which are consumed in large quantities by CAM units [40]. Since RowSelect bus consists of n one-bit signals from Stage-C that indicates which rows of Table 1 had a BF Membership test match. The interconnect logic schedules data reads from the RAM units (corresponding coefficients of Table1) in a staggered manner because Stage D has b slices whereas Stage C has n slices ($b << n$).

In a similar manner, Scheduler-2 (shown in Fig. 5.5) between stages D and E schedules the multiplication of the scalar coefficients. Since both interconnects-2 &3 share the same working principle, hence in this section we will explain the working of one of them.

5.7.1 Design of the Scheduler

We designed a scheduler that enabled the dispatch of p rows of scalar coefficient pairs (Coeff_a, Coeff_b) every cycle. Fig. 5.6 & Fig. 5.5 shows the position of the schedulers within the data path. The n DataReady lines from the previous stage are inputs to the scheduler. Every cycle, the scheduler routes p rows to p multipliers via p multiplexers (Mux_0 to Mux_p in Fig. 5.5) until all the valid rows have been processed.

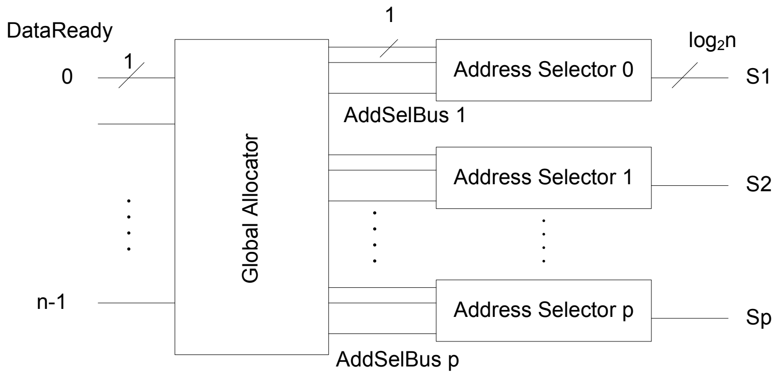


Fig. 5.7. Internal Structure of the Scheduler

The outputs of the multipliers are fed to a single cycle accumulator that can handle p inputs. Thus at the end of $(c/p) + L + 1$ cycles, the value that is output by the accumulator is the similarity value.

Internally, the scheduler consists of two main components: a “Global Allocator” and a set of p “Address Selector” blocks (shown in Fig 5.7). The global allocator allocates a multiplier to every scalar coefficient pair that needs to be multiplied as determined from their DataReady lines. Each Address Selector then schedules the allocated lines sequentially by setting the appropriate selection line (S_1-S_p) of the multiplexer (shown in Fig 5.6) with the address of the line being scheduled. The global allocator is connected to the Address Selectors through the AddSelBus buses.

5.7.2 Overview of Scheduler operation

The operation of the scheduler logic circuit is best explained using the timing diagram as shown in Fig. 5.8. For this subsection, we'll consider Scheduler-1 that sits between Stages C & D. Suppose $X_0, X_1, \cdots, X_{b-1}, X_b, \cdots X_q$ are the index numbers of rows (high $RowSelect$ signals) that have possible common basis vectors, then the interconnect logic schedules data reading from the RAM units in a staggered manner.

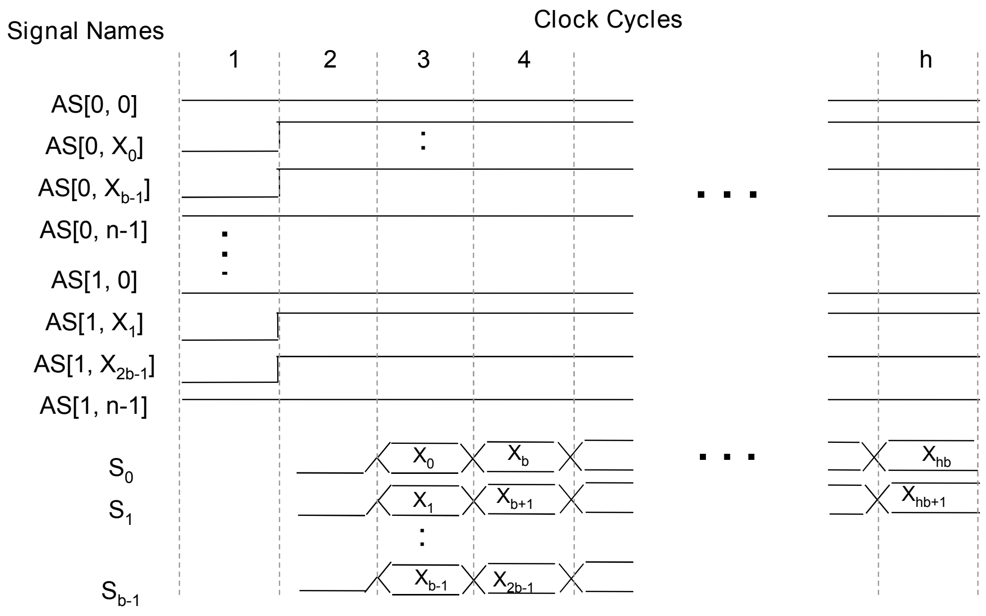


Fig. 5.8. Timing Diagram of the Scheduler

The address locations: $X_0, X_1, \cdots X_{b-1}$ are read in the first cycle; locations: $X_b, X_{b+1}, \cdots X_{2b-1}$, are read in the next cycle and so on.

All signals are 1-bit unless indicated otherwise. The allocator circuit groups and allocates the $RowSelect$ signals to all b address selectors in a single cycle (Cycle-1, in Fig. 5.8). The allocator outputs b bundles of AS signals, with each bundle containing n signal lines. The i^{th} bundle of the j^{th} signal is denoted as $AS[i, j]$. In the first bundle, the signals $AS[0, X_0], \cdots AS[0, X_b], \cdots AS[0, X_{h.b}]$ are driven high by the allocator circuit, where $h = \frac{q}{b}$ (q = number of rows having high $RowSelect$ signals, b = number of CAM units). In the second bundle, the signal lines $AS[1, X_1], \cdots AS[1, X_{b+1}], \cdots AS[1, X_{h.b+1}]$ are driven high, and so on. The i^{th} Address Selector schedules its output address lines (S_i bus) depending on $AS[i, X_i] \cdots AS[i, X_{h.b+i}]$ signals in its incoming bundle. This address bus S_i drives the i^{th} RAM unit in stage D.

5.7.3 Design & Operation of Global Allocator

Fig 5.9 shows the internal details of the global allocator. It consists of a one-hot priority coding circuit [41], a 2^{nd} level detector, a static allocator and n demultiplexers. The output of the one-hot priority coder discloses the position of the first valid (1) line from the n DataReady lines.

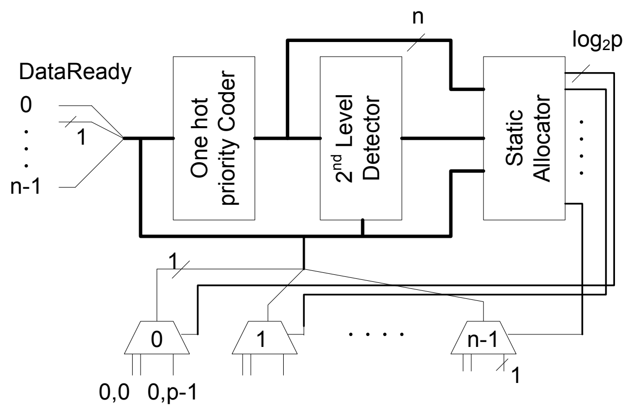


Fig. 5.9. Details of Global Allocator

The 2^{nd} *level detector* uses the output of the *One-hot priority coder* and the n DataReady lines to identify the position of all valid DataReady lines starting from the second valid position. The outputs of the priority coder and the 2^{nd} level detector are used by the static-allocator to allocate the lines that need to be serviced amongst the p address selectors. The 2^{nd} level detect signal is generated using the combinational logic $L2_i = \bar{L1}_i \cdot L0_i$ where $L1_i$ is the output of the one-hot priority coder and $L0_i$ is the DataReady bit of line i. The multiplier allocation is determined as $a_i = (a_{i-1} + L2) mod\, p)$ where a_i is the allocated multiplier for line i.

5.7.4 Design & Operation of Address Selector

The Address Selector needs to pick one allocated line ($DataReady_i$=1) from AddSelBus_i in every cycle so that it can pass the corresponding coefficient pairs to its multiplier using the corresponding line address. The challenge is to identify the next available line without requiring an $O(n)$ lookup. This requires a logic block that can identify the first un-serviced pair in every cycle from its input AddSelBus. This task is addressed by the *Address Selector* block. (Fig 5.10).

The Address Selector circuit consists of a set of BitMask cells that are connected to a one-hot priority coder. The AddSelBus is connected to the priority coder through the BitMask cells. The output of the priority coder circuit is fed to an encoder as well as the Mask input of the respective BitMask cell. The encoder generates the address of the line to be serviced. (S_i)(as shown in Fig. 5.6)

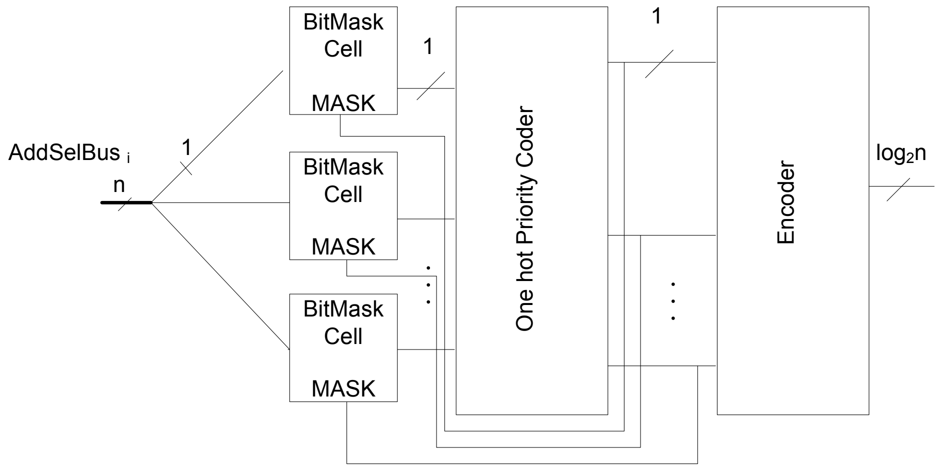


Fig. 5.10. Details of Address Selector

The one-hot priority coder of the Address Selector indicates the position of the first high input from among its input lines (*AddSelBus_i*) on its n output lines. This represents the first pair of coefficients to be fed to the multiplier. Blocking of this bit from being passed back to the priority coder in future cycles will ensure that the scheduled rows are no longer in consideration for future scheduling. The BitMask cell provides this blocking feature when the Mask signal is asserted. The one-hot priority coder along with the BitMask cell addresses the challenge of finding the next valid line to be serviced in a single cycle without requiring an $O(n)$ lookup.

The BitMask Cell as shown in Fig. 5.10 consists of a D flip-flop (DFF) with a AND gate in a feedback loop. The cell has two states-*Pass* and *Block* as shown in Fig. 5.11 . When Reset is asserted, the DFF output is preset to 1 placing the cell in the *Pass* state. In *Pass* state, the cell transmits the input to the output every cycle. The clear input of the DFF is connected to the Mask line. When this is asserted, the output of the DFF goes low from the next clock cycle and hence blocking the transmission of the input for future cycles (*Block* state).

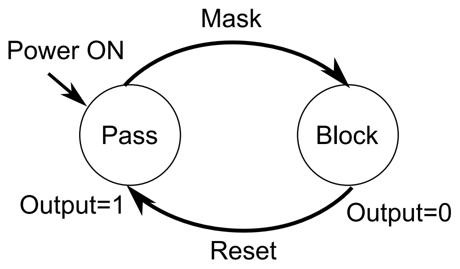


Fig. 5.11. State Diagram of BitMask Cell

5.8 Timing Analysis

The computation time t_2 for the computation of the dot product can be denoted as:

$$t_2 = \left(\left\lceil \frac{n}{r} \right\rceil * k \right) + E + \left\lceil \frac{G}{b} \right\rceil + S + \left\lceil \frac{|n * c|}{p} \right\rceil + L + A \tag{5.1}$$

where

$$G = (c + (n - c) \cdot P_{false+ve}) \tag{5.2}$$

Here, n = number of basis vectors in the smallest vector/tensor, r = number of parallel bloom filter test circuits, k = Number of Bloom Filter Indices, E = Number of cycles needed to extract the coefficients (using CAM), b = Number of CAM blocks, S = Number of cycles to schedule the multiplications, c = percentage similarity between the two tables being compared, p = Number of multipliers available, L = Latency of each multiplier, A= Latency of the adder and $\lceil\rceil$ denotes the ceiling function. In our design, $E = S = A = 1$ and $L = 5$. Here the bottleneck is the multiplier stage as only a few multipliers can be used because they are expensive in terms of power and silicon area. The BF will help us identify the $c + (n - c) \cdot P_{false+ve}$ suspected matching basis vectors (where $P_{false+ve}$ is the probability of BF false positives). In the CAM lookup stage, the c suspects are confirmed and $(n - c) \cdot P_{false+ve}$ false positive suspects are rejected. Hence, there are only $n \cdot c$ multiplications that need to be carried out by the p multipliers in the last stage.

5.8.1 Overall Timing Analysis

The computation time T for both major stages is the sum of times t_1 (Equation 4.1) and t_2 (Equation 5.1.)

$$T = t_1 + t_2 \tag{5.3}$$

hence:

$$\begin{aligned}T = {} & 2 * (t_{FNV} + W) + O + D \\ & + \left(\left\lceil \frac{n}{r} \right\rceil * k\right) + E + \left\lceil \frac{(c + (n - c) \cdot P_{false+ve}}{b} \right\rceil \\ & + S + \left\lceil \frac{|n * c|}{p} \right\rceil + L + A\end{aligned} \tag{5.4}$$

5.9 Power Figures

As mentioned previously in Section 4.3, the proposed design was implemented in Verilog and simulated using ModelSim from Mentor Graphics. To obtain power results, synthesis was performed on the verified design using Design Compiler from Synopsys using components from the DesignWare IP Library and the TSMC 90nm technology library. The memory power data was obtained using the CACTI power model [39]. The CAM power data was obtained from the work on low power CAM designs done by Ng et al. in [42].

Table 5.1
Power Figures for Computation of Cosine Similarity

Module	Num instances	Power
Identification of Common Vectors	n (1024)	0.11 W
Extraction of Coefficients	b (16)	1.976 W
Multiplication & Summation of Coeffs	p (8)	7.77 mW
Interconnects	2	0.55 W

5.10 Overall Results

5.10.1 Comparison with Server Class Processor

To obtain timing on a representative server class processor, an Intel Xeon processor was used to executed code that performed dot product computation. The dot product code identifies common basis vectors using a Binary Search Tree. Once the common basis vectors are identified, the corresponding coefficients are multiplied to obtain the similarity value. Thus this search is of the order of $O(n_1 log(n_2))$. In Algorithm. 2 we present the pseudocode for the software implementation of dot-product (cosine similarity) computation.

Algorithm 2
Pseudocode for optimum software based comparison

```
function ComputeDotProduct
                                        ▷ Timing Measurement begins here
                                                        ▷ Build the tree
    for $i \leftarrow 0, NumberofBasisVectorsinTable - 1$ do
        rbtree.insert($table1\_coefficient[i]$)
    end for
                                                       ▷ Search the tree
    for $i \leftarrow 0, NumberofBasisVectorsinTable - 2$ do
        $rb\_treeptr \leftarrow rb\_tree.search(table2\_coefficient[i])$
        if $rb\_treeptr != NULL$ then
            $dot\_product \leftarrow dot\_product + (table2\_coefficient[i] + prt.value)$
        end if
    end for
    return $dot\_product$
                                          ▷ Timing Measurement ends here
end function
```

The order of speedup (Xeon vs. our design) is the same when fixed point or floating point representation is used. The speedup is $\approx 4K$ in case of floating point compared to $\approx 3.5K$ if fixed point representation is used for $n = 1024$, $c = 100\%$ on Xeon. However fixed point representation will require pre- and post-processing of every 16 bit coefficient value, which is between 0 and 1. This will add significant overheads when implemented at the user programming level, hence to obtain best results it is left to the discretion of the compiler by choosing float representation.

The balanced binary tree was implemented using the STL Map container as provided by the GCC Compiler. The GCC STL-Map implements the binary search tree using a highly optimized red-black tree implementation.

We did not consider the alternative software approach for comparison, which is software implementation of the Bloom Filter. This is because this approach would involve multiple hash computations for each basis vector and would be computationally more expensive and hence suboptimal. Hence this approach is not suitable for comparison.

In all cases, the program memory was locked into RAM using the Linux *mlock* command for the duration of the process. This ensure that the memory allocated to the program is not swapped out during the course of execution. We further ensured that our code ran at the highest priority possible (for a userspace program) and did not get swapped between processors (using the *taskset* command). Hence our execution time does not get influenced by other tasks on the system.

We use clock cycles in lieu of execution time to compare designs in a clock speed neutral manner. We did this to ensure fair comparison with reference hardware designs [43] [44] [45] which have been implemented and evaluated at different clock speeds. Details of this comparison is presented in Section 5.10.3.

For all simulation experiments below, number of basis vectors $n = r = 1024$, expected number of common basis vectors $c = 102$ (10% of n, a very conservative value), number of multipliers $p = 16$, BF length $m = 10240$, number of BF hash functions $k = 7$ (optimum), BF false positive probability $P_{false+ve} \approx 8.3 \times 10^{-3}$ (for $n = 1024$), and $E = S = A = 1$ unless different values are implied. The expected values of execution/processing times are the simulation results. The performance evaluation results of our design and the comparison against available hardware designs are presented in Table- 5.2.

In Table 5.2 we present the superior performance of our design in terms of speed (clock cycles to perform semantic comparison for a pair of vectors) and circuit power draw (for 90nm technology, 3Ghz) compared to an Intel Xeon processor (a representative server class processor). The Intel Xeon (3Ghz version), used in this example, has a maximum instruction per cycle (IPC) figure of 4. All other high performance sequential processors have IPC of very similar order. Hence the performance cannot be significantly improved any further on a traditional server class processor. We discuss the limitations of multi-core/GPU based systems in the next section.

Table 5.2
Superior Performance of Hardware Design

Comparison	Execution time (in cycles)		Power Draw
	c=10%	c=100%	
Proposed Hardware	131	303	10.52 Watts
Intel Xeon	390,986	557,592	40-80 Watts/core
Comments	Speedup of 2984	Speedup of 1840	82% less Power

The addition of Interconnect-2 to the design presented in [46] allowed a significant reduction the number of CAM blocks required. This along with a lower number of multipliers, contributed towards a reduction in power consumption from the then reported figure of 109W [46] to 10.528 Watts in this design, though additional computational stages were added.

5.10.2 Power Consumption per Functional Block

In table 5.3 we present the break up of the power consumed by the major functional blocks

As can be seen from the above table, the major power draw is from the Bloom Filter Generation and consolidation stages (Stages A&B). This is primarily because of the large number of OR-gates involved in the consolidation of the n independent Bloom Filters into a single BF and then distributing them to the n different rows of Stage-C (Identification of Common Basis Vectors).

Table 5.3
Power Consumed by Each Functional Block

Stage	Functional Block	Power Draw
A	Generate BF indicies	6.230 W
B	Bloom Filter Consolidation	1.66 W
C	Identification of Common Vectors	0.11 W
D	Extraction of Coefficients	1.976 W
E	Multiplication & Summation of Coeffs	7.77 mW
-	Schedulers	0.55 W
	Total	10.528 W

5.10.3 Comparison with Other Hardware Designs

From our literature search, we were believe we are the first to investigate the concept of hardware based *semantic comparison*. Hence direct comparison against other semantic comparators is not possible since they do not exist. However, hardware to compute cosine similarity (which is a part of our computation) has been investigated by [43] [44] [45]. Fig 5.12 shows the comparison of speedup of our design against that of the other available designs (compared to the execution time of an equivalent efficient software code). We show comparison with both c=100% and c=10% by converting their reported execution times to clock cycles.

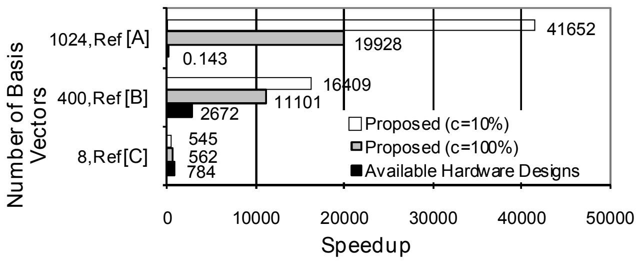


Fig. 5.12. Comparison of Speedup Against Other Hardware Designs. [A]=Ref. [45], [B]=Ref. [43], [C]=Ref. [44]

Table 5.4
Speedup Comparison with Other Hardware Designs

Number of Basis Vectors	Compared Against	Improvement in Speedup (times)	
		c=10%	c=100%
8	Ref. [43]	0.696	0.717
400	Ref. [44]	6.14	4.15
1024	Ref. [45]	291,275	139,357

The other hardware designs do not take into consideration the number of common basis vectors. The proposed design performs consistently better due to fine grained parallelism in the hardware for large meaning vectors (number of basis vectors = 400, 1024). Such parallelism has not been exploited by other hardware based designs [43] [44] which carry out the computations sequentially. [45] uses a parallel execution scheme, however, their design leads to an execution time which scales exponentially with the number of vectors being processed. In comparison, our design takes a much lower number of cycles and scales linearly with a very small slope within the given range. In addition, none of these approaches perform true semantic comparison.

In Table- 5.4 we present a comparison of our hardware with those presented in [43] [44] [45] and show the factors of speedup by which our design performs better. For large meaning vectors (number of basis vectors = 1024) our design gives a speedup increase of 291,275 times for c=10% and 139,357 times for c=100% compared to the hardware in [45]. The other hardware designs do not address power issues and hence it is difficult to compare power consumption.

5.10.4 Overall Execution Time

Our design can handle a larger number of input rows ($n > 1024$) by splitting the rows into multiple partitions of 1024 rows each. For example if the input row size were 2048, it would be split into two partitions of 1024 rows each and then processed. Fig. 5.13 shows the comparison of number of cycles required by the proposed hardware and Xeon for different values of n (number of basis vectors) to show their relative scaling behavior with n.

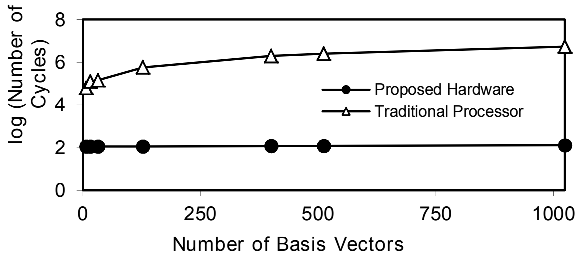


Fig. 5.13. Variation in Number of Clock Cycles between Proposed Hardware and Traditional Processor

5.10.5 Variation of Number of Basis Vectors

Fig. 5.14 shows that the execution clock cycles varies linearly (with a small slope) with number of basis vectors n (scaling behavior) as there are limited number of multiplier units. This and all following experiments are carried out with the parameters of $n = 1024, k = 7, b = 16 \& p = 8$ unless specified otherwise.

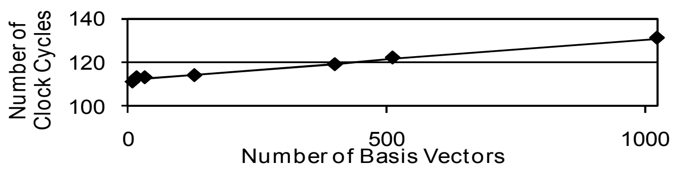


Fig. 5.14. Variation in Execution Time due to Number of Basis Vectors

5.10.6 Variation in Speedup with Similarity

Fig. 5.15 shows the variation in Speedup with change in percentage similarity among the two Tables. Smaller value of c leads to lesser number of multiplications.

The variation is bounded and gives a speedup of at-least 19,969 for c=100% (worst-case).

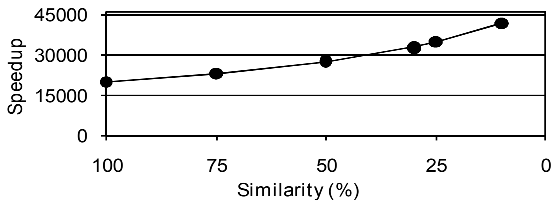


Fig. 5.15. Variation of Speedup with Variation in Similarity

5.10.7 Variation in Timing due to Number of CAM Units

Fig. 5.16 shows variation number of Clock cycles needed by the proposed hardware with varying number of CAM's but fixed number of rows n = 1024, multipliers p = 8. The three starred points indicate the reach of a steady state in number of cycles. Increasing the number of CAM units beyond this point does not yield any improvement in performance.

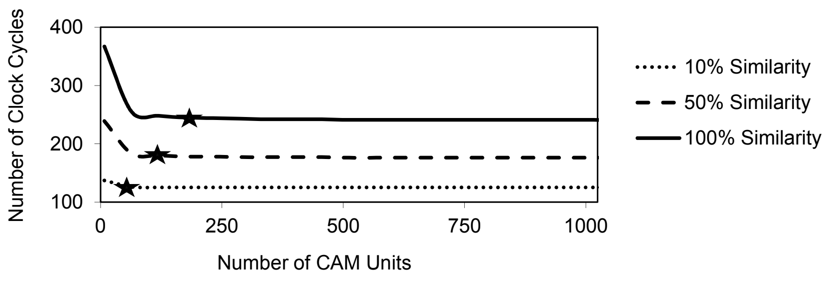


Fig. 5.16. Variation in Execution Time due to Number of CAM Units

5.10.8 Variation in Timing due to Number of Multiplier Units

Fig. 5.17 shows variation number of Clock cycles needed by the proposed hardware by varying the number of multiplier units with fixed number of rows n = 1024, number of CAM units $b = 16$. As in the previous case, a steady state (indicated by the star) is reached beyond which there is no reduction in the number of clock cycles required to compute the similarity value.

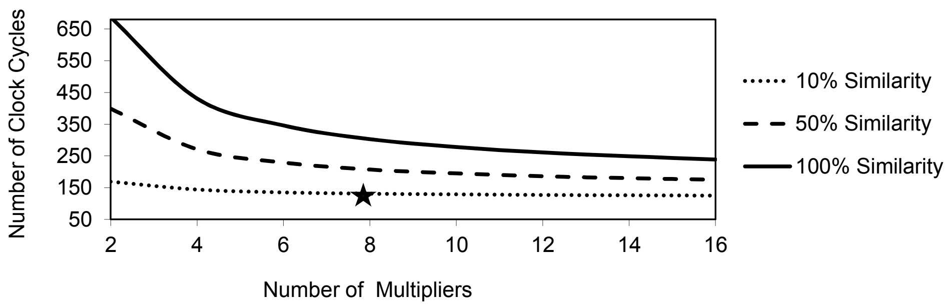


Fig. 5.17. Variation in Execution Time due to Number of Multiplier Units

The steady state value of CAM and Multiplier Units (16 CAM and 8 multipliers) for 10% similarity and $n = 1024$ rows can be seen in Fig. 5.16 & Fig. 5.17. This justifies our choice for choosing these values in our synthesized design.

As can be seen from Fig. 5.16 & Fig. 5.17, there is a mininum number of clock cycles required for the computation of the semantic similarity. The figures appear to saturate at a certain number of cycles because of the number of parallel units in either stage-C or Stage-D. If the value of number of CAM units (in Stage-C)(b) and number of multipliers (in Stage-D) (p) were set to the maximum of 1024 (full parallelism), then number of clock cycles required to compute the similarity value would be 132 (for 10% common basis vectors). This number is because there are (atleast) 100 rows that need to extracted and matched. Stages C-E would consume 17 cycles for this and Stage-A,B would consume 96 cycles. This can be verified by using equation 5.4. The mimium number of cycles is dependent on those steps which cannot be parallized efficiently.

6. PERFORMANCE ENHANCEMENTS AND ALTERNATIVE IMPLEMENTATIONS

In this section, we will look at performance enhancements and two alternative implementations of the proposed architecture. The parallel algorithms are hardware agnostic and hence by implementing these in different hardware platforms we can study the relative merits and demerits of the respective platform for the proposed application. In section 6.1, we present the pipelined implementation and results. In section 6.2 we present the implementation details on an Nvidia CUDA enabled graphics card. The algorithm was also implemented on a Xilinx Virtex-5 FPGA, in section 6.3 we present the details and power figures from this.

6.1 Pipelining

By implementing pipelining into the design, performance enhancements can be obtained. Pipelining in our case allows the processing of multiple comparisons, allowing a new one to start before the previous comparison completes. This reduces the latency between successive semantic comparisons.

6.1.1 Implementation

In Fig. 6.1 we present the overall architecture of the pipelined version of our design. We extended the design of the basic architecture (presented in Section 3.3) by adding pipeline registers and associated control logic. In the original design, the steps of computation were already demarcated by the presence of schedulers and the interconnect logic between the various processing steps. Hence for ease of implementation, the pipeline registers were primarily inserted into the data-path by modifying the two schedulers (Scheduler 1 and Scheduler 2). This allows each processing block to complete processing, hand over the intermediate results to the next stage and immediately begin processing the next set of input values.

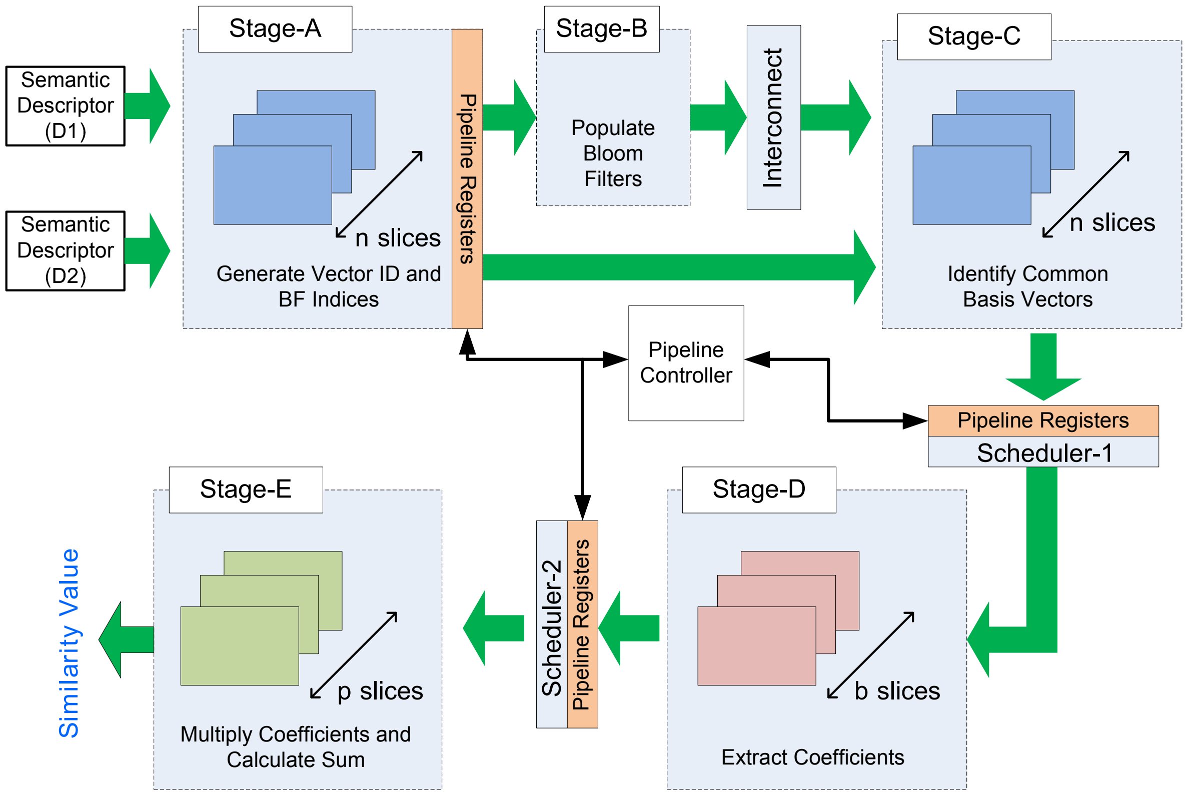


Fig. 6.1. Overall Architecture with Pipeline Enabled

A block diagram of the pipeline control logic is given in Fig. 6.2. The logic control block keeps track of how many inputs were given to the specific processing block and computes the number of cycles required to generate the outputs. It then waits for the required number of cycles before releasing the next set of inputs to that processing block.

To enable pipelining, registers to hold the intermediate values were inserted into the datapath (Shown in Fig. 6.1 as *Pipeline Registers.*) There are three optimal locations for these. Before the inputs to the schedulers (for stages B,C) and right after Stage-A. By inserting the registers into the data path at these locations, they

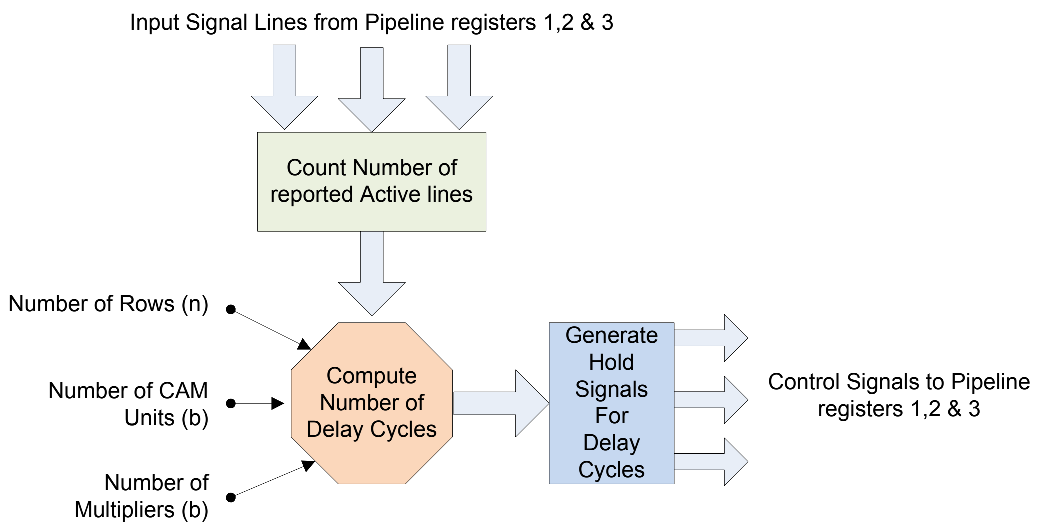

Input Signal Lines from Pipeline registers 1,2 & 3
Count Number of reported Active lines
Number of Rows (n)
Number of CAM Units (b)
Number of Multipliers (b)
Compute Number of Delay Cycles
Generate Hold Signals For Delay Cycles
Control Signals to Pipeline registers 1,2 & 3

Fig. 6.2. Block Diagram of Pipeline Controller

buffer the results of the previous stage, while the previous stage begins computation of the next set of data. The control logic controls the flow of data by turning ON/OFF the appropriate gates that lead to these registers.

6.1.2 Operation

The pipelined version of the semantic comparator works in the same manner (algorithm wise) to the non-pipelined version. It begins by taking in the two tables to be compared, generates the BF indices and stores the data into the tables. The common basis vectors for the tables being compared are then identified. Once the common basis vectors are identified, the corresponding scalar coefficients are extracted and the sum of products (cosine product) of these scalars is computed. This final computed value is the semantic similarity between the tables being compared.

Pipelining is enabled by the registers and control logic that have been inserted into the existing datapath. The output of each stage, is tapped by the control logic block to count the number of outputs generated and being given to the next stage as input. For example, if Stage-C (Identification of common basis vectors) identifies 20 common basis vectors, then there will be 20 values heading to Stage-D (extraction of coefficient values). Since the logic block knows that there are 16 CAM block available, it can compute, that to test all 20 values, Stage-D will take 2 cycles. (16 in the first cycle, and 4 in the next cycle). Hence the logic block, delays the entry of the next set of values into Stage-C and therefore into Stage-D accordingly so that in three cycles from now, Stage-D will have another set of values to test for.

6.1.3 Results

We present the power and timing results from the pipelined implementation in this section. Pipelining does have an power overhead due to the added pipeline registers plus the associated control logic. The trade-off of the increased power is the reduced latency between successive outputs of data, leading to speedup.

Table 6.1
Comparative Results between Pipelined and Non-Pipelined Designs

Design	Execution Time (cycles)	Latency	Power Draw
Non-pipeline design	131	131	10.52 Watts
Pipelined design	139	8	13.563 Watts
Comments	-	16x Speedup	$\sim$ 30% power overhead

The pipelined version takes an extra 8 cycles to process the data. The eight cycles is introduced due to the pipeline registers involved. The pipeline registers at each of the interconnects takes 2 cycles of delay, and the remaining two cycles are used for synchronization at the input and output of the pipeline. Other performance figures such as scaling for varied number of rows remains the same as the non-pipelined version.

6.2 CUDA Implementation

An implementation of the developed algorithms for semantic comparison was performed on an NVIDIA-CUDA compatible GPGPU [47]. In this section, we present the details of the implementation and results from some experiments to study the performance in comparison to the ASIC design presented in the previous sections.

6.2.1 Introduction to CUDA

Modern GPUs released by hardware manufacturers NVIDIA and ATI (acquired by AMD in 2006) have numerous (> 100) SIMD (Single Instruction Multiple Data) processing cores. These cores have the ability to process several parallel streams of data at any given point in time. The parallel processing capability of these cores has attracted the interest from many diverse fields that benefit from parallel processing such as Information Retrieval [48], Video/Audio Encoding [49] [50], Astronomy [51], Medical Sciences [52] & Bio-Informatics [53].

In response to the interest from research community and industry, these manufactures have released APIs' and programing models that allow these cards to perform non-graphical computation as well. NVIDIA's implementation of this approach is termed Compute Unified Device Architecture (CUDA) [16] [49]. (ATI/AMD's similar implementation for its range of multi-core graphics cards is marketed under the name AMD Firestream [54])

The CUDA SDK consists of a collection of APIs (high and low level), compiler & device drivers that allow for custom code to be written, compiled and executed on the graphics card. The software that runs on the CUDA cards, is written in a version of the C programming language called "C for CUDA" comprising of restrictions and extensions defined by NVIDA to ensure compatibility with the CUDA architecture. The CUDA architecture has become popular in the community, with wrappers for several other languages being developed including Perl [55], Python [56], Fortran [57], Java [58] and also platforms such as Mathematica and MATLAB [59].

6.2.2 Overview of CUDA Architecture

A CUDA enabled GPU consists of multiple light-weight SIMD processing cores. The number of cores ranges from as low as 8 on the *QuadroFX 370 LP* [60] all the way to 512 on the *M2090 GPU Computing Module* [61]. PCI-E is used as the interface between the GPGPU and the Host PC's motherboard.

The Host PC initially loads a self contained code segment (called a kernel) into the GPGPU memory. Each SIMD processor then executes this kernel, generally over different data-elements in parallel. The GPGPU can process thousands of threads simultaneously. Using scheduling mechanisms, the number of logical threads and thread blocks (groups of logical threads) surpasses the number of physical execution units.

Fig. 6.3 shows the logical subdivision of CUDA functionality into Grids, Kernels and Threads [16]. The kernel, launches a Grid of Thread blocks. The hardware scheduler on the GPU schedules thread blocks onto the individual cores. CUDA Threads are extremely light-weight with very little creation overhead and can perform very fast context-switching.

Fig. 6.4 shows the memory model of the CUDA architecture [49]. There are three major groups of memory accessing. Threads - which can access registers (per thread) and local memory (off-chip); Blocks - which share memory between threads and Device. The Device contains three forms of memory: Global Memory (shared between kernels), Constant memory (read only, stores invariants) and Texture memory (which is limited and distributed, but can cache parts of the Global Memory).

Processing of data/executing a program (Programming Model) on the CUDA architecture, consists of the following steps:

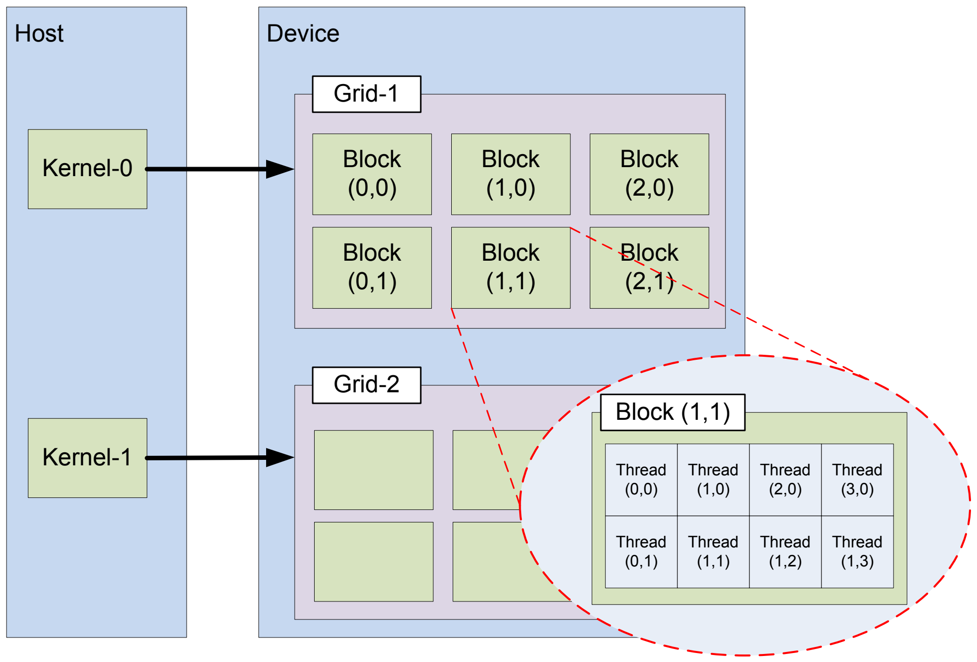


Fig. 6.3. CUDA Logical Layout

1. Split a task into subtasks
2. Divide input data into chunks that fit global memory
3. Load a data chunk from global memory into shared memory
4. Each data chunk is processed by a thread block
5. Copy results from shared memory back to global memory

We map the hardware algorithm to perform the semantic comparison into the same series of steps. In the next section, we discuss the phases involved in semantic comparison and how they have been mapped to the programming model of CUDA.

6.2.3 Phases Involved in Semantic Comparison on CUDA

The implementation of our semantic comparator algorithm in the CUDA architecture consists of four phases. The four phases map the parallel algorithm into a

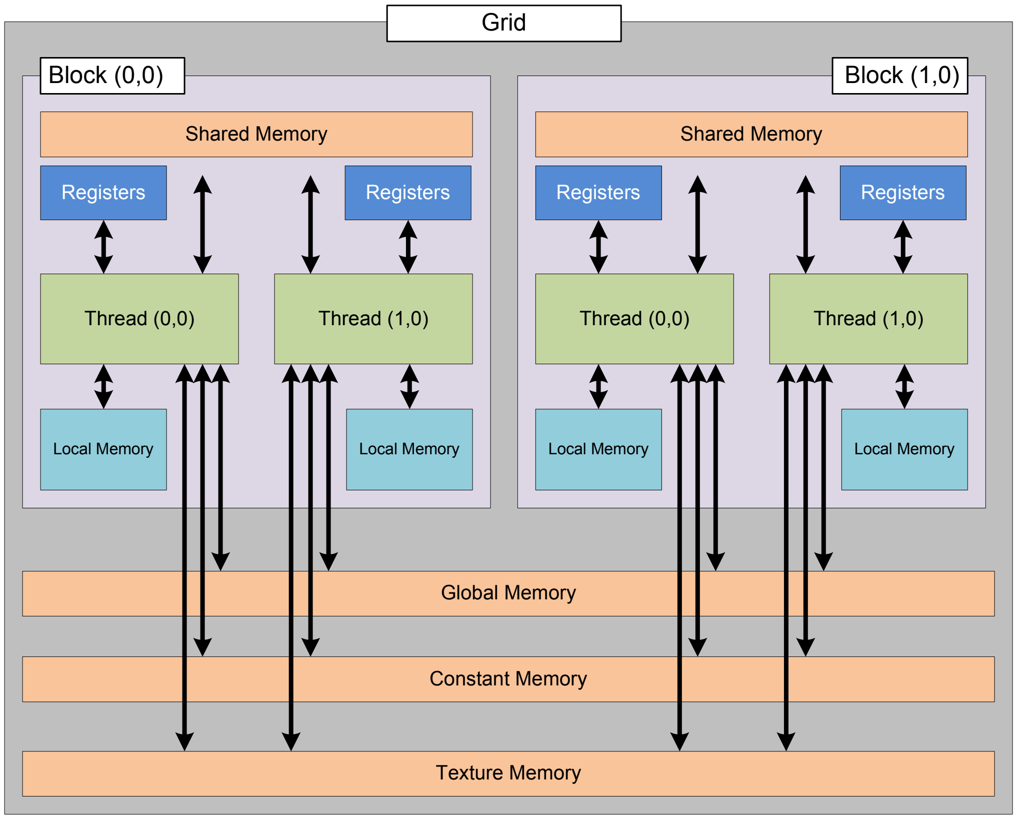


Fig. 6.4. CUDA Memory Model

form suitable for execution on the CUDA enabled graphics card. The four phases involve:

(A) copying the data structures (tables) into the CUDA memory from the Host-PC

(B) performing the computation to generate the vector IDs

(C) using Bloom Filters to identify the common basis vectors and

(D) extracting the common-coefficients to perform the dot-product computation.

Several optimizations were performed to enable efficient use of the CUDA resources. These include:

1. Maximize independent parallelism - We minimized the inter-core communication as much as possible.
2. Data structure is flattened to increase coalesced memory accesses - The layout of the data-structures (written in C) were modified slightly to obtain a flatter structure. This allowed us to maximize the available PCIe bandwidth (76.8 Gb/s for the NVIDIA C870 [62])
3. Limit the number of blocks and increase the number of threads. This allows us to increase the reuse of shared memory.
4. Since shared memory is inaccessible after the end of kernel execution, we transfer the data from the kernel to the Global memory as the last step of the kernel.
5. Partitioning the computation to keep all stream cores busy. Using multiple threads we were able to keep multiple thread blocks in constant use.
6. Monitoring per-processor resource utilization. By ensuring low utilization per thread block allows us to have multiple active blocks per multi-processor.

In the next section, we discuss each of the processing stages of the computation in detail.

6.2.4 Phase A: Host-PC to CUDA Global Memory Copy

In the first phase of the computation, Coefficient Table-1 & 2 are copied to CUDA Global Memory. The data structure is internally flattened to ensure coalesced memory accesses. The flattening of the data structure is performed basically as a serialization of the data structure. By ensuring that the data can be read into the CUDA processor in a continuous stream, we accelerate the copy. This process is shown in Fig. 6.5. The transformation into a coalesced memory layout ensures maximum usage of available of PCIe bandwidth.

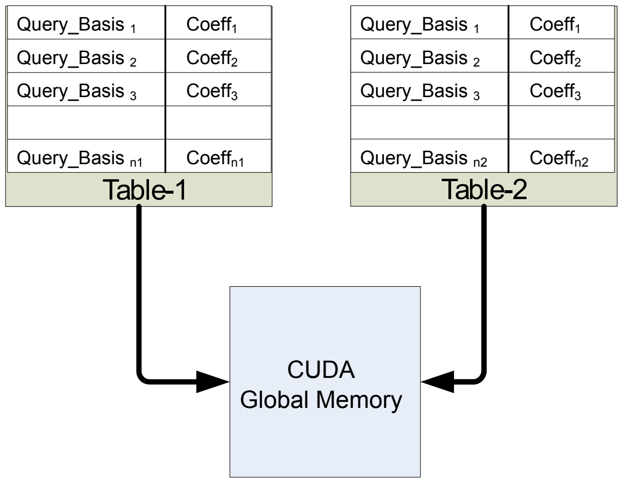


Fig. 6.5. Copying Tables into CUDA Memory

6.2.5 Phase B: Encode Table-1 in BF

In phase-B of the processing, we encode the contents of Table-1 into the Bloom Filters. This is performed using a number of concurrent kernels that run on the CUDA processors. In each kernel, a given $Query\_Basis_i$ is encoded into the Bloom Filter as originally discussed in Section 6.2.3. As shown in Fig. 6.6, we make $n1$ concurrent kernel calls (independent threads) so that each row of Table-1 is served by at least one CUDA thread. The CUDA occupancy calculator provided by NVIDIA as part of its CUDA toolkit allowed us to calculate the appropriate device parameters to ensure that each multiprocessor has a sufficient number of free registers (prevents blocking).

In the CUDA implementation, we implemented the Bloom Filter Index generation using two hash functions $Hash_1$ = FNV [30] & $Hash_2$ = JS Hashes [63] (is based on the Hash function methodology mentioned in [64]). These are implemented as device functions because of their computational simplicity. They produce two 64-bit hash values for each $Query\_Basis_i$ term that they operate upon. These hash values are then used to compute k different bloom filter index values as described in

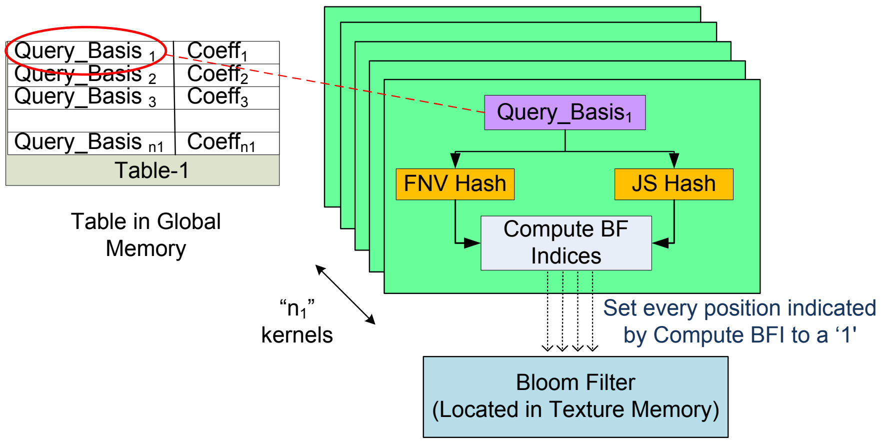


Fig. 6.6. Encoding Elements of Table-1 using Concurrent Kernels

Equation 6.1. In the equation, int_r in a random integer value. By placing the Bloom filter (bit-array of size n) in texture memory we avoid the latency in memory access amongst the concurrently running kernels.

$$BFI_k = Hash_1(Element) + int_r \times Hash_2(Element) \tag{6.1}$$

6.2.6 Phase C: Encode Table-2 and Test in BF

Fig. 6.7 shows the implementation of Phase-C. This phase is similar to Phase B with two major differences;

1. Instead of setting a bit position in the Bloom Filter to 1, it tests for the presence (or absence) of the $Query\_Basis_i$ term that the kernel is operating on. (Membership testing phase)

2. If all the bits indicated by the BF Index for the given $Query\_Basis_i$ are "1" in the Bloom Filter, then the corresponding index i is stored in shared memory to be used in the next phase of computation.

We launch $n2$ (or more) concurrent kernels in this phase. Every kernel instance performs three steps (a) encodes a row of Table-2, (b) tests with the previously encoded BF (Phase B) and (c) stores the index values of "matches found".

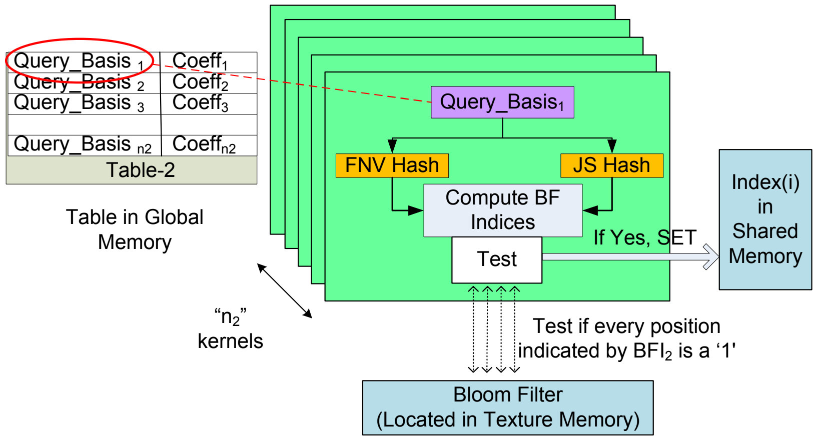


Fig. 6.7. Encoding Elements of Table-2 using Concurrent Kernels

6.2.7 Phase-D: Extract Coefficients and Parallel Reduction

In phase-D, the final stage of the Semantic comparison algorithm is performed. The CUDA implementation of this algorithm is shown in Fig. 6.8. Phases A-C enabled the "filtering" (Identification) of the common basis vectors with $O(1)$ complexity. These "filtered" common basis vector terms are then loaded from shared memory (Step-1) and the corresponding Coeff2 value is fetched from Global Memory (Step 2). In step 2, the corresponding $Query\_Basis_i$(from Table2) is located in Table 1 (Extraction of Common Basis Vectors). This is a computationally intensive process but occurs only for a small subset of the identified elements. This stage also performs the elimination of false positives due to the Bloom Filter. Once the corresponding coefficients (Coeff1 and Coeff2) have been obtained, a partial product is calculated (Step 4). In Step 5 we perform parallel reduction of the interim products from each kernel to obtain a final sum. This final reduction is optimized using the guidelines published in [65].

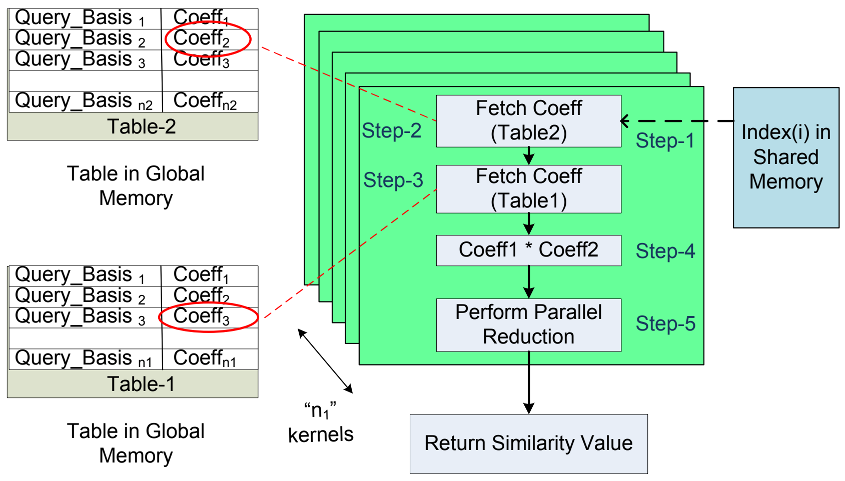


Fig. 6.8. Extracting Coefficients and Performing Parallel Reduction

Extracting the coefficients (step2 in Fig. 6.8) can be done in one of two methods. Test filtered Table-1 against copies of Table-2 or test Test table-2 against filtered copies of table-1. While at a high level, both methods involve the same process, the ability to parallelize on CUDA is greatly exploited in the second case. Fig. 6.9 shows the variation in the processes. It can be seen that be testing table-2 against copies of the filtered table-1, we can utilize the processors more effectively be exploiting the parallel cores of the processor.

In comparison, if we do the first method, we are artificially restricting the number of parallel cores to that of the number of filtered rows of table-1, and hence not fully utilizing the processor. Hence, using method-2 (shown in Fig. 6.9(b)) gives better performance than the method of Fig. 6.9 (a).

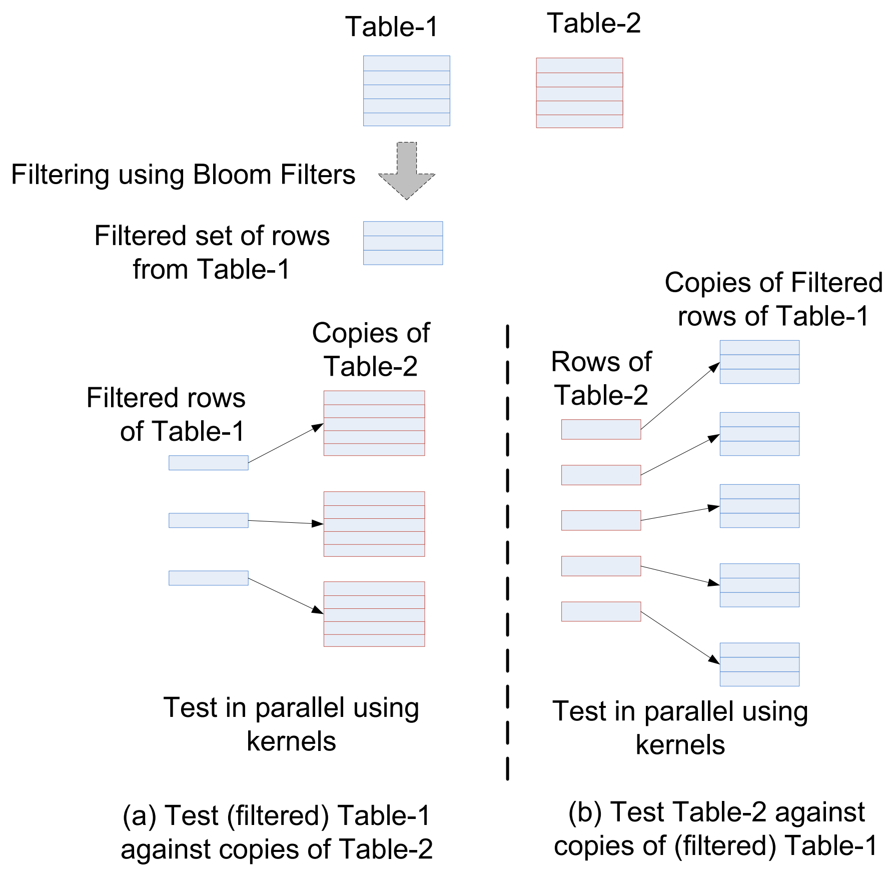


Table-1
Table-2
Filtering using Bloom Filters
Filtered set of rows from Table-1
Copies of Table-2
Filtered rows of Table-1
Test in parallel using kernels
(a) Test (filtered) Table-1 against copies of Table-2
Copies of Filtered rows of Table-1
Rows of Table-2
Test in parallel using kernels
(b) Test Table-2 against copies of (filtered) Table-1

Fig. 6.9. Alternative Methods of Performing Extraction of Common Coefficients

6.2.8 Experimental Setup

The GPGPU card that we used is a NVIDA Tesla C870 (Compute Capability 1). The GPU contains 16 stream processors with a total of 128 cores each running at 600 MHz. The card has 1.56 GiB of RAM running at 1.6 GHz [62]. We achieved the rated memory bandwidth of 76.8 GB/s for all experiments. The interface to the host PC is over a 16x PCI-Express bus. CUDA Toolkit version 3.1 was used for compilation.

Power Profiling

The power monitoring was done using "Watts' Up? Pro" power analyzer from Electronic Educational devices [66]. This device measures overall (bulk) system power consumption. This device is connected in line with the power supply to the host computer as shown in Fig. 6.10. It would be useful to measure more accurate power values, but this would require significant resources such as monitoring probes that are inserted in-line with the PCI-E bus lanes. In our current set of experiments this wasn't considered, but it could provide better insights for future work in this area.

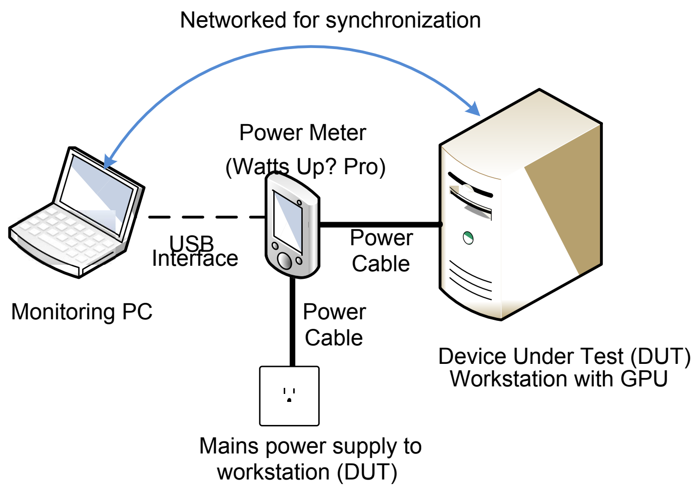


Fig. 6.10. Setup for Power Profiling

Each experiment was run so that the overall program executes for at least 10 seconds (multiple iterations used when necessary). This ensures that the readings from our power monitoring device to be stable so that we are not affected by the transient surges that could be present due to the startup and shutdown spikes in power consumption from both the Host PC as well as the GPGPU itself. In table 6.2 system base power is the static power consumed by the host computer without the GPU present in it. System Idle power is the power consumed with the GPU present but in cold shutdown state (We define cold shutdown state as the state where GPU has not been activated by software since the booting of the host PC). GPU idle power is the power consumed with the GPU awake but not running any specific computation. This is computed by subtracting System Idle Power from power consumed by the system after the GPU is awake but in idle state. The CUDA cards have a documented effect whereby, if the card is in cold shutdown state, the card consumes less power than after being activated once. Once the card has been activated for the first time, it transitions to the idle-state when not being used and not to the cold shutdown state. Currently, the only way to return the C870 Tesla card to the cold shutdown state (of power consumption) is to reboot the host PC.

Table 6.2
Baseline Power Figures

System base Power	115W
System Idle Power (GPU Cold shutdown)	150W
System Idle Power (GPU Awake, idle)	186W
GPU Idle Power	36W

Execution Time Profiling

The time-accurate simulator which implements the algorithms described in Section 3.2 was used in the computations. This is the same code that was used to estimate the cycle time for the pure hardware design. We use the CUDA API timers and CPU system time to measure the run-time execution time of the core computational kernels and equivalent CPU code respectively. In order to ensure complete utilization of all the CPU cores, we set the block size (number of threads per block) to 384. (Max supported by the C870 card for concurrent execution when number of blocks is greater than 1). According to the CUDA programmers guide for the C870 card [62], the processor contains 16 multi-processors each with 8 SIMD cores.

Each of the 16 streaming processors can handle up to 32 threads at a time, however, the internal scheduler will first schedule the first 24 threads and then run the next 8 threads. Hence, by extrapolating this figure, we can see that by scheduling 384 threads at a time, we can fully utilize all 128 processor cores each running at their max capacity of 3 threads each.

6.2.9 Results

In this section, we present and analyze overall execution time, power and throughput for the semantic comparator core on a CPU and a GPU respectively. We have conducted all our experiments with $n1 = n2 = N$. In a real-life semantic router the coefficient tables will have sizes $n1 << n2$. Our results represent the worst case situation. We experiment for :

1. N varying from 100 to 150000 rows and
2. Similarity c varying between 0.1 and 1(All Match) in the two coefficient tables.

Overall Execution Time

Table 6.3 shows the overall execution time of Phases B-D as outlined in Section 6.2.3 with varying input size of the coefficient tables under experimentation (N). These results are shown for the situation $c = 0.1$ (similarity between simulated tables=10%) and $n1 = n2$. The execution time of the CPU increases exponentially as the number of entries increases whereas the same operation on the GPU is an order of magnitude faster and does not rise exponentially. The numbers represented are the result of running the simulation multiple times and averaging the time across the individual runs.

As can be seen from Table 6.3, there is a minimum execution time for the GPU even at a low number of rows, higher than the CPU time. This is a combination of the

Table 6.3
Execution Times of GPU and CPU

Number of Rows	CPU Time (ms)	GPU Time (ms)
8	0.002	1.842
64	0.056	1.832
128	0.124	1.919
256	0.525	1.907
512	1.908	1.937
1024	7.820	2.144
2048	31.492	2.651
3000	75.949	4.286
4096	124.330	4.010
5000	185.189	5.077
10240	779.319	13.857
150000	184818.945	1885.930

time required to transfer the kernel code to the GPU from the host PC as well as the overheads of the hardware scheduler. As the number of rows of execution increases, the inherent parallelism of the GPU and the parallel algorithm allows for faster processing compared to the $O(n^2)$ computation required by sequential processor of the host CPU which leads to an exponential growth in execution time.

Overall Power Consumption

Fig. 6.11 shows the variation of power consumed by the CPU and GPU respectively with varying table size. This experiment was conducted for $c = 0.5$ (50% similarity between table entries). The dynamic power for the GPU is lower than that consumed by the CPU. GPU Power approaches that of the CPU for extremely large datasets (above 50,000 entries). This is a known problem for GPUs [67] - they are energy efficient and not necessarily power efficient.

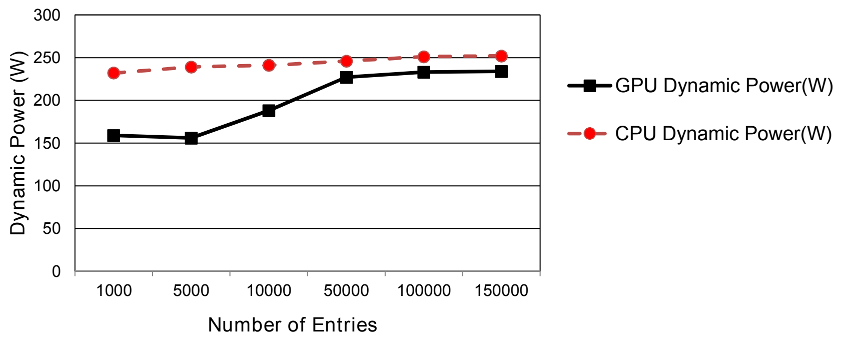


Fig. 6.11. Dynamic Power Consumption

Energy Consumption

Energy consumption for $5000 < N < 150000$ and $c = 0.75$ is shown in table 6.4. We present the average power consumption for the CPU and GPU respectively when running Phases A-D for varying sizes of N and $c = 0.75$ (for a single comparison). The average energy saved across varying table sizes is $\approx 78\%$.

Table 6.4
Energy Consumption for Different Table Sizes

Table Size	CPU Avg. Power (W)	GPU Avg. Power (W)	Energy Saved (%)
5k	232	159	67.59
10k	239	156	79.65
50k	241	188	77.64
100k	246	227	77.27
150k	251	233	77.96

These results show that in the long-term a semantic comparator using a CPU-GPU hybrid in its compute nodes can (a) Reduce its energy footprint and/or (b) Increase its throughput while maintaining the same energy footprint. Hence, the parallel architecture that we propose can contribute to energy savings both when implemeted on an ASIC design or on the NVIDIA CUDA platform.

6.3 FPGA Implementation

6.3.1 Implementation Details

The proposed hardware architecture was implemented onto a Xilinx Virtex-5 FPGA board [17]. The architectural layout is shown in Fig. 6.12. The layout is similar, with the different stages implemented as slices and the number of slices being varied depending on the corresponding stage. Due to a difference in the number of slices between stages, a scheduler was implemented in between the various sections.

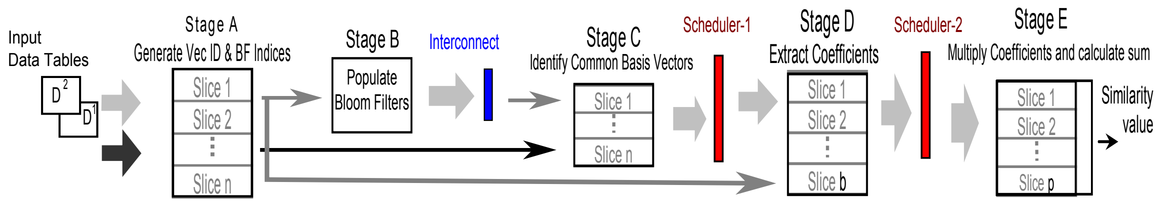


Fig. 6.12. Architecture of FPGA Implementation

A difference in the implementation, was in the design of the allocators (schedulers). In the FPGA version, a simpler allocator was used. The allocator was implemented using a Fixed Priority Arbiter (FPA) as shown in Fig. 6.13. To explain the working of the FPA, consider scheduler-2 which is situated in between the CAM blocks and the multipliers. If there are p multipliers in use, p FPA instances are instantiated. To allocate two of the multipliers to the corresponding data-path, the first FPA grants (g_0) a request (r_0) and sets the corresponding request input of the next arbiter to 0. Thus, the second FPA will not grant the request which was granted by first FPA, and will grant another request at the same clock cycle as the first one. Thus, the cascaded FPAs act as an allocator. Once a request is granted, it is reset to 0 in the next clock cycle. The allocator allocates in a cyclic manner until all the requests are reset to 0.

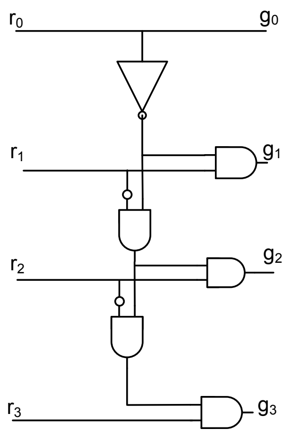


Fig. 6.13. Schematic of Fixed Priority Arbiter

A state machine was implemented to sequence the loading and processing of data across the stages. The state machine has two main control steps: (1) Load Table A & (2) Process Table B. In step-1, Table A is selected in Stage-Init (loading the data) and sent to stage-A. Stage-B processes the output of stage-A and loads BF and Table A in to the memory elements of the design. In step 2, Table B is selected in Stage-Init and sent to stage-A. The output of stage-A is bypassed to stage-C and the computation follows the process outlined in Section 3.3.

6.3.2 Results

A small-scale version of the proposed hardware was implemented onto an Xilinx Virtex-5 board. The design with $n = 8$, $b = 4$ and $p = 2$ was created and synthesized using Xilinx ISE 10.1. The area usage figures are presented in Table- 6.5.

Power numbers obtained using Xpower Analyzer with 25% FF activity is presented in Table- 6.6 for parameters of $n = 8$, $b = 4$ and $p = 2$. Synthesis of a few different sizes showed us that that logic power scales linearly with area, and as the number of rows gets larger, logic power becomes dominant. Thus, overall power also scales linearly with area.

Table 6.5
FPGA Resource Utilization

Parameter	Utilization Number	Percentage Utilization
Number of Slice Registers	5116 out of 69120	7%
Number of Slice LUTs	14587 out of 69120	21%
Number used as Logic	10477 out of 69120	15%
Number used as Memory	4110 out of 17920	22%
Number of Block RAM/FIFO	44 out of 148	29%

Table 6.6
FPGA Power Consumption

Power summary	Current (mA)	Power(mW)
Total power consumption	-	1109
Clocks	74	74
Logic	16	16

Area usage of three implementations for various values of n are shown in Fig. 6.14. In all three cases, there are 4 CAM blocks and 2 multipliers. Area usage scales less than linearly, and the FPGA can hold a design with maximum size of n=12, b=4 & p=2. This behavior is expected since the CAM, RAM and multiplier blocks are the largest consumers of area on the FPGA fabric. Increasing the number of CAM units or multipliers takes up more fabric than increasing the number of rows in the initial stages.

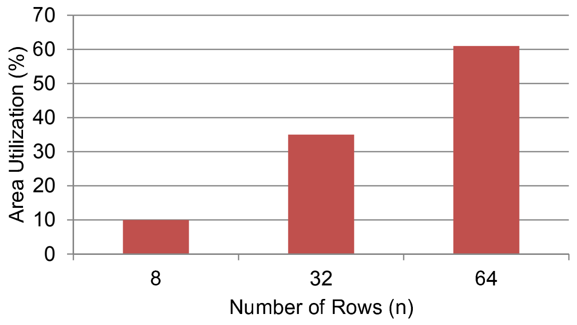


Fig. 6.14. Area Usage on the FPGA for Varying Number of Rows

7. CREATION OF A SEMANTIC BENCHMARK

7.1 Need for a Semantic Benchmark

In previous chapters, we have explained how a Semantic Routed Network (SRN) comprising of multiple semantic routers can be used for two purposes: (1) to selectively forward/route a (query) message to an index shard based on the meaning of the query; and (2) to automatically re-distribute index entries based on meaning of the documents/objects. Semantic routers use a data structure called a semantic key to represent meaning of a message and use semantic comparison (comparison of semantic keys) as a primitive computation to decide the next hop message destination(s). In addition to index re-organization and query delivery, meaning comparison can be also used to carry out index lookup necessary for meaning based search operations [8].

When attempting to evaluate a semantic comparator, there is a notable absence of a valid benchmark. The primary requirement of a semantic benchmark is to test the ability to compare the query against a known corpus and see how similar the corpus and the query are to each other. This is unlike traditional search where the aim is to see if the term/document exists in the corpus. In order to evaluate the performance of efficient semantic comparator designs, a benchmark a required that allows for semantic comparison to take place. Semantic comparison is not a traditional computing primitive, hence traditional performance benchmarks such as SPEC [68] or MediaBench [69] are not adequate.

The goals of the Semantic Benchmark are:

(a) To provide a valid set of semantic data that can be compared using semantic comparison.

(b) Focus on keeping the benchmark data portable so that it can be customized for other applications.

(c) Develop a dataset that can be extended by the research community as research on semantic comparison and semantic routed networks progresses.

7.2 Comparison with Other Benchmarks

there are several well-known and accepted benchmarks. The most commonly used benchmarks include, Dhrystone [70] for embedded systems, TPC-D benchmark [71] for databases, NetBench [72] for Network processors, 3dMark [73] for 3d applications, MediaBench [69] that allows for benchmarking multi-media and related operations and the SPEC benchmark [68] which has been in use since its introduction in the late 80s. Traditional benchmarks like the ones previously mentioned, do not allow you to perform semantic comparison. They look at traditional metrics such as ILP, MIPS and throughput.

The search community has its share of benchmarks including the Billion Triple Challenge from the Semantic Web Challenge [74], LETOR (Learning to Rank) [75] from Microsoft Research and TREC [76]. The traditional search benchmarks do not suit our requirements, since they look at traditional information retrieval paradigms (such as recall rate). In contrast, we are looking for the ability to perform semantic comparison between multiple data-units.

7.3 Description of Corpus

The corpus of data for the benchmark was derived from the Semantic Network published by the NIH [77]. The semantic networks consists of "a set of broad subject categories, or Semantic Types, that provide a consistent categorization of all concepts represented in the UMLS Metathesaurus, and (2) a set of useful and important relationships, or Semantic Relations, that exist between Semantic Types." [21] A sample of a few lines of the corpus data is shown in Fig. 7.1.

```
Anatomical_Structure|part_of|Amphibian
Anatomical_Structure|part_of|Plant
Anatomical_Structure|part_of|Reptile
Anatomical_Structure|part_of|Vertebrate
Anatomical_Structure|part_of|Virus
```

Fig. 7.1. Sample of Corpus from UMLS Semantic Network

7.4 Components of the Semantic Benchmark

The two primary parameters that we define for the semantic benchmark are percentage similarity c and the number of rows of data being compared. The first of these parameters (percentage similarity) is the true measure of how close two objects being compared are. The number of rows determines the size of the corpus. The number of lines that can be handled at a time determines the throughput of the comparator.

```
...
>PlantVirus<: 0.109544516
>VertebrateAmphibianVirusReptile<:  0.020701967
>PlantVirusReptile<: 0.1796053
>VirusReptile<:  0.035856858
>VertebrateAmphibian<:  0.14342743
>VertebratePlantAmphibianVirusReptile<:  0.0621059
>AmphibianReptile<: 0.1796053
>Vertebrate:  0.06324555
...
```

Fig. 7.2. Sample of Tensors after Processing UMLS Corpus

In order to create the benchmark, we first extracted portions of the UMLS data set. These portions where then converted into tensors using the techniques of [37] whereby the UMLS data is first cast into a concept-tree form and then the tensors of that concept tree are then derived. Creation of tensors allows for semantic comparison. In Fig. 7.2 we present a sample of the tensors created from the data. The various levels of the concept tree are delimited by the vectors denoted by $\lhd$ and $\rhd$. Details of the concept trees and its mapping to tensors is presented in [8].

Table 7.1
Datasets in Semantic Benchmark Suite

Dataset ID	Number of Rows	Similarity
A	1024	100%
B	1024	50%
C	1024	25%
D	1024	10%
E	1024	5%
F	1024	1%
G	1024	0.1%
H	1024	10%
J	512	10%
K	256	10%
L	64	10%
M	32	10%
N	16	10%

By varying the major parameters of the benchmark, we have generated thirteen datasets with the parameters as shown in table 7.1. We also provide the entire dataset corpus of 7000 lines along with the individual TF-IDF values. These can be used to extend the benchmark as required for other research.

8. CONCLUSIONS AND FUTURE WORK

8.1 Future Work

This dissertation explored several design and research questions ranging from the design of an application specific hardware co-processor that can perform dot-product (cosine similarity) computation in a power and time efficient manner to the design of a benchmark that provides for semantic comparison. However, there are several areas of research that have not been explored. Some of these potential areas of research are listed here:

1. The semantic comparator architecture is the core of the semantic router. The rest of the semantic router still remains to be designed. While most of the router can be based on traditional network router design principles, there will be architectural designs required to store and process the semantic routing tables. In addition,dealing with the the large amount of data required to process the objects being compared is a challenge to be solved.

2. I/O remains a bottleneck in this design. Loading the tables into memory for the first step of the computation is still the slowest point of the design. Therefore, research into a design that can bring together memory and logic in a single unified design will pay dividends. Examples include approaches like the Computing Cache architecture [78] or perhaps integrating processing cores into DRAM chips.

3. As can be seen from the CUDA implementation, by implementing parallelism at the fine grained level (circuit level, Stage B,C,D in Section 3.3) as well as coarse-grained (multiple processing cores in CUDA, Section 6.2), we can obtain the ability to handle large data sets. Hence hybrid parallelism, involving next generation processors such as the Intel SCC [13] should be investigated for performance.

4. Implementation of the proposed architecture on other parallel platforms such as a Systolic Array [79] or a Coarse Grained Reconfigurable Array (CGRA) [80] [81] could give further insights into the parallelism of the algorithm and could boost performance further.

5. The semantic benchmark can be extended for use in other domains. It can be converted into network packet formats which could allow it to be used in typical network simulators.

6. The proposed pipeline architecture is lock-step in design per comparison pair. Unlocking the pipeline to enable use of the resources between pairs will enhance throughput further.

7. Existing popular network simulators such as GloMoSim [82] or NS-2 [83] can be extended to simulate and evaluate the performance of overlay semantic routed networks on traditional networks.

8. In this dissertation, we discussed one application of SRN - in a Search Engine. The SRN is good at information location as well as retrieval, its application to other fields should be studied. Some examples of areas where the SRN could be deployed include Emergency Response, Disease tracking, Autonomous Robots and Big Data Curation.

9. The ability to perform semantic comparison - comparing based on the meaning/content has wide applications from Machine Learning [84] to Intelligent Agents [85] to Human Computer Interaction [86].

10. One of the key requirements for successful semantic comparison, is the creation of tensors from existing data sources. This is currently a manual process requiring human curation. Automation of this area using research from the fields of Natural Language Processing [87] and Artificial Intelligence [85] has the potential to improve and enhance the entire experience.

8.2 Conclusion

In applications where threads are short and need limited memory, fine grained circuit level parallelization can be a viable alternative to multi-core processor enabled parallelization. The proposed parallelization scheme avoids the expensive hardware-software design effort and high overheads associated with multi-core processor based designs.

In this dissertation we showed how a hardware circuit algorithm can enable circuit level parallelization, deliver superior performance as compared to contemporary hardware designs or purely software implementations. We presented the application context and design specifications, which involved: design of a hardware centric algorithm & mapping of the algorithm to hardware. The low power architecture presented consumes 82% less power and demonstrates a speed-up in the order of 10^5 compared to a contemporary hardware design, and in the order of 10^3 compared to software approach for large number of basis vectors.

We also presented a pipelined architecture for performance improvement and comparison with implementations on two contemporary platforms - an NVidia CUDA and an FPGA. We also presented the creation of a semantic benchmark for validation purposes.

The semantic comparator presented here is the core processing unit for a Semantic router, which is the key networking component for a Semantic Routed Network. SRN has the potential to improve performance on networks dealing with information retrieval. To that extent we presented Search as a potential application and explained how the SRN can help improve search performance. The high performance low power architecture can be used to elegantly implement energy efficient distributed search engines.

REFERENCES

[1] A. Patriquin, "March search market share: Record query growth and the yahoo/microsoft search deal by the numbers," *Compete, Inc*, Apr. 2008, http://blog.compete.com/2009/04/13/search-market-share-march-google-yahoo-msn-live-ask-aol-2/; Accessed Apr. 15th, 2009.

[2] A. Agarwal, "Single google query uses 1000 machines in 0.2 seconds," *Digital Inspiration*, Feb. 2009, http://www.labnol.org/internet/search/google-query-uses-1000-machines/7433/; Accessed Apr. 15th, 2009.

[3] L. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *Micro, IEEE*, vol. 23, no. 2, pp. 22 – 28, Mar-Apr. 2003.

[4] U.S. Environmental Protection Agency, "Report to congress on server and data center energy efficiency public law," U.S. EPA ENERGY STAR Program, Washington, DC, Tech. Rep., Aug. 2007, http://www.energystar.gov/index.cfm?c=prod_development.server_efficiency_study; Accessed Oct. 28th,2010.

[5] N. E. C. A. (NECA), "Data centers - meeting today's demand," Electrical Design Library, Tech. Rep., Aug. 2007, http://www.necanet.org/files/ACF41A4.pdf; Accessed Mar. 15th, 2009.

[6] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manage.*, vol. 24, no. 5, pp. 513–523, Jan. 1988.

[7] J. C. Perez, "Google joins crowd, adds semantic search capabilities," *ComputerWorld*, Mar. 2009, http://www.computerworld.com/s/article/9130318/; Accessed Apr. 15th,2009.

[8] A. Biswas, S. Mohan, J. Panigrahy, A. Tripathy, and R. Mahapatra, "Representation of complex concepts for semantic routed network," in *Proc. IEEE 10th Int'l Conf. Distributed Computing and Networking (ICDCN)*, Jan. 2009, pp. 127–138.

[9] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval.* Harlow, England: Addison-Wesley, 1999.

[10] A. Biswas, S. Mohan, and R. Mahapatra, "Search co-ordination by semantic routed network," in *Proc. IEEE 18th Int'l Conf. Computer Communications and Networks (ICCCN09).* IEEE, Aug. 2009, pp. 1–7.

[11] A. Biswas, S. Mohan, and R. Mahapatra, "Optimization of semantic routing table," in *Proc. IEEE 17th Int'l Conf. Computer Communications and Networks (ICCCN08).* IEEE, Aug. 2008, pp. 298–303.

[12] A. Biswas, S. Mohan, A. Tripathy, J. Panigrahy, and R. Mahapatra, "Semantic key for meaning based searching," in *Proc. IEEE 3rd Int'l Conf. Semantic Computing (ICSC09).* IEEE, Sep. 2009, pp. 209–214.

[13] Intel Corporation, “Intel single-chip cloud computer,” *Intel Labs*, February 2010, http://techresearch.intel.com/ProjectDetails.aspx?Id=1;Accessed Mar. 21st,2011.

[14] Intel Corporation, “Intel thread building blocks, Version 3.0,” *Intel Press*, 2010, http://www.threadingbuildingblocks.org; Accessed Jul. 15th,2010.

[15] J. Hoberock and N. Bell, *Thrust: A Parallel Template Library Version 1.3.0*, 2010, http://www.meganewtons.com/.

[16] NVIDIA Corporation, *NVIDIA CUDA Programming Guide 3.0*, February 2010, http://developer.nvidia.com/cuda-toolkit.

[17] Xilinx Inc, *Xilinx DS100 Virtex-5 Family Overview*, Feb. 2009, http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf; Accessed Sep. 21st, 2010.

[18] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, “SOAP version 1.2,” World Wide Web Consortium (W3C), Tech. Rep., Apr. 2007.

[19] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller, “WordNet: An on-line lexical database,” *International Journal of Lexicography*, vol. 3, pp. 235–244, Dec. 1990.

[20] The Gene Ontology Consortium, “The gene ontology project in 2008,” *Nucleic Acids Research*, vol. 36, no. S-1, pp. D440–D444, Jan. 2008.

[21] O. Bodenreider, “The unified medical language system (UMLS): Integrating biomedical terminology,” *Nucleic Acids Research*, vol. 32, no. suppl 1, pp. D267–D270, Jan. 2004.

[22] D. J. Watts, “Networks, dynamics, and the Small-World phenomenon,” *The American Journal of Sociology.*, vol. 105, no. 2, pp. 493+, Sep. 1999.

[23] S. Hares, Y. Rekhter, T. Li, and E. Addresses, “A border gateway protocol 4 (BGP-4),” Internet Requests for Comment (RFC 4271), RFC Editor, Tech. Rep. 4271, Jan. 2006, http://www.ietf.org/rfc/rfc4271.txt.

[24] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and L. Beck, “Improving information retrieval using latent semantic indexing,” *Proc. 1988 Annual Meeting of the American Society for Information Science*, pp. 36–40, 1988.

[25] A. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[26] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: A scalable wide-area web cache sharing protocol,” *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[27] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proc. 14th Annual European Symposium on Algorithms (ESA 2006)*, no. 4168. LNCS, Sep. 2006, pp. 684–695.

[28] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *Proc. 15th Annual ACM-SIAM Symposium on Discrete algorithms (SODA04)*. Society for Industrial and Applied Mathematics, 2004, pp. 30–39.

[29] P. Almeida, C. Baquero, N. Preguica, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, Mar. 2007.

[30] G. Fowler, L. C. Noll, and P. Vo, *Fowler / Noll / Vo (FNV) Hash*, Sep. 1991, http://isthe.com/chongo/tech/comp/fnv/; Accessed October 24th, 2009.

[31] R. Rivest, "The MD5 Message-Digest algorithm," Internet Request for Comment (RFC 1321), RFC Editor, Tech. Rep. 1321, Apr. 1992, http://www.ietf.org/rfc/rfc1321.txt.

[32] P. Gallagher, "FIPS PUB 180-3 secure hash standard (shs)," National Institute of Standards and Technology, Information Technology Laboratory, Tech. Rep. Oct., 2008, http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.

[33] A. Augello, G. Vassallo, S. Gaglio, and G. Pilato, "Sentence induced transformations in "conceptual" spaces," in *Proc. IEEE Int'l Conf. Semantic Computing, (ICSC08)*. IEEE, 2008, pp. 34–41.

[34] D. Widdows, "A mathematical model for context and word-meaning," in *Proc. 4th Int'l and interdisciplinary conference on Modeling and using context*. Springer-Verlag, 2003, pp. 369–382.

[35] D. Widdows, "Semantic Vector Products: Some Initial Investigations," in *Proc. Second Conference on Quantum Interaction*. University of Oxford, College Publications, Mar. 2008, pp. 1–8.

[36] J. Mitchell and M. Lapata, "Vector-based models of semantic composition," in *Proc. Association for Computational Linguistics (ACL08: HLT)*. ACL, Jun. 2008, pp. 236–244.

[37] J. Panigrahy, "Generating tensor representation from concept tree in meaning based search," Master's thesis, Texas A&M University, College Station, TX, May 2011.

[38] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: building a better bloom filter," in *Proc. 14th Annual European Symposium*, vol. 14. Springer-Verlag, 2006, pp. 456–467.

[39] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "CACTI," HP Laboratories, Palo Alto, Tech. Rep., 2008, http://www.hpl.hp.com/research/cacti/.

[40] S. Mohan, A. Tripathy, A. Biswas, and R. N. Mahapatra, “Parallel processor core for semantic search engines,” in *Proc. IEEE 25th Int’l Parallel and Distributed Processing Symposium (IPDPS 2011) - IPDPS Workshops*, May 2011, pp. 1767–1775.

[41] Synopsys Inc, “1-Hot priority coder,” *DesignWare Component Reference*, Aug. 2008.

[42] K. F. Ng, “Novel low power CAM architecture,” Master’s thesis, Rochester Institute of Technology. Computer Engineering, New York, 2008.

[43] M. Freeman, M. Weeks, and J. Austin, “Hardware implementation of similarity functions,” in *Proc. IADIS Int’l Conf. Applied Computing*. IADIS, Feb. 2005, pp. 329–332.

[44] D. G. Perera and K. F. Li, “On-chip hardware support for similarity measures,” in *Proc. IEEE Pacific Rim Conf. Communications, Computers and Signal Processing (PACRIM)*. IEEE, Aug. 2007, pp. 354–358.

[45] D. G. Perera and K. F. Li, “Parallel computation of similarity measures using an fpga-based processor array,” in *Proc. IEEE Int’l Conf. Advanced Information Networking and Applications (AINA)*. IEEE, Mar. 2008, pp. 955–962.

[46] S. Mohan, A. Biswas, A. Tripathy, J. Pannigrahy, and R. Mahapatra, “A parallel architecture for meaning comparison,” in *Proc. IEEE 17th Int’l Parallel and Distributed Processing Symposium (IPDPS 2010)*. IEEE, Apr. 2010, pp. 1–10.

[47] A. Tripathy, S. Mohan, and R. N. Mahapatra, “Optimizing a semantic comparator using CUDA-enabled graphics hardware,” in *Proc. IEEE 5th Int’l Conf. Semantic Computing (ICSC)*, Sep. 2011, pp. 125–132.

[48] C. S. Kouzinopoulos and K. G. Margaritis, “String matching on a multicore GPU using CUDA,” in *13th Panhellenic Conf. on Informatics*, Sep. 2009, pp. 14–18.

[49] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *ACM Queue*, vol. 6, no. 2, pp. 40–53, March 2008.

[50] J. Huang, S. P. Ponce, S. I. Park, Y. Cao, and F. Quek, “Gpu-accelerated computation for robust motion tracking using the CUDA framework,” in *Proc. IET 5th Int’l Conf. Visual Information Engineering (VIE 2008)*, vol. 1. IET, Aug. 2008, pp. 437 –442.

[51] R. G. Belleman, J. Bedorf, and S. F. P. Zwart, “High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in CUDA,” *New Astronomy*, vol. 13, no. 2, pp. 103 – 112, 2008.

[52] T. Reichl, J. Passenger, O. Acosta, and O. Salvado, “Ultrasound goes GPU: Real-time simulation using CUDA,” in *Proc. Medical Imaging 2009: Visualization, Image-Guided Procedures, and Modeling*, vol. 7261, no. 1. SPIE, 2009, pp. 726 116–20.

[53] L. Ligowski and W. Rudnicki, “An efficient implementation of smith waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases,” in *Proc IEEE Int’l Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2009, pp. 1–8.

[54] F. P. Miller, A. F. Vandome, and J. McBrewster, *AMD FireStream*. Mauritius: Alphascript Publishing, 2009.

[55] D. Mertens, “Perl’s first real CUDA bindings released,” Jun. 2011, http://blogs.perl.org/users/david_mertens/2011/06/perls-first-real-cuda-bindings-released.html, Accessed Oct. 1st, 2011.

[56] A. Klockner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA and PyOpenCL: a scripting-based approach to gpu run-time code generation,” *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012.

[57] M. Wolfe, “CUDA fortran: The next level,” *PGI Insider*, Sep. 2010, http://www.pgroup.com/lit/articles/insider/v2n3a1.htm, Accessed Oct. 5th, 2011.

[58] G. Dotzler, R. Veldema, and M. Klemm, “JCudaMP: OpenMP/java on CUDA,” in *Proc. 3rd Int’l Workshop on Multicore Software Engineering (IWMSE 2010)*, May 2010, pp. 10–17.

[59] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Boston,MA: Addison-Wesley Professional, Jul. 2010.

[60] NVIDIA Corporation, *NVIDIA Quadro FX 370M*, 2011, http://www.nvidia.com/object/product_quadro_fx_370_m_us.html; Accessed May 10th, 2011.

[61] NVIDIA Corporation, *TESLA M-CLASS GPU Computing Modules: Product Brief*, 2011, http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf; Accessed Oct. 10th, 2011.

[62] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “Nvidia tesla: A unified graphics and computing architecture,” *Micro, IEEE*, vol. 28, no. 2, pp. 39 –55, Mar.-Apr. 2008.

[63] Arash Partow, *General Purpose Hash Function Algorithms*, http://www.partow.net/programming/hashfunctions/; Accessed Dec. 15th, 2008.

[64] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston,MA: Addison Wesley, Jan. 1986.

[65] M. Harris, G. Blelloch, B. Maggs, N. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha, P. Smolarkiewicz, L. Margolin *et al.*, “Optimizing parallel reduction in cuda,” *Proc. ACM SIGMOD*, vol. 13, no. 21, pp. 104–110, 2007.

[66] Electronic Educational Devices, *Watts up? Pro*, Denver, CO, 2009, http://www.wattsupmeters.com/; Accessed Jun. 15th, 2010.

[67] Y. Jiao, H. Lin, P. Balaji, and W. Feng, "Power and performance characterization of computational kernels on the GPU," in *Proc. Int'l Conf. Green Computing and Communications & Int'l Conf. Cyber, Physical and Social Computing (GREENCOM-CPSCOM 2010)*, 2010, pp. 221–228.

[68] J. J. Dujmovic and I. Dujmovic, "Evolution and evaluation of SPEC benchmarks," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 26, pp. 2–9, Dec. 1998.

[69] C. Lee, M. Potkonjak, and W. H. Mangione Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Annual ACM/IEEE Int'l symposium on Microarchitecture (MICRO 30)*, 1997, pp. 330–335.

[70] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Commun. ACM*, vol. 27, pp. 1013–1030, Oct. 1984.

[71] M. Thiyagarajah and B. J. Oommen, "On benchmarking attribute cardinality maps for database systems using the TPC-d specification," in *Proc. 10th Int'l Conf. Database and Expert Systems Applications (DEXA99)*, 1999, pp. 292–301.

[72] G. Memik and W. H. Mangione Smith, "Evaluating network processors using NetBench," *ACM Trans. Embed. Comput. Syst.*, vol. 5, pp. 453–471, May 2006.

[73] F. N. Sibai, "Performance analysis and workload characterization of the 3DMark05 benchmark on modern parallel computer platforms," *SIGARCH Comput. Archit. News*, vol. 35, pp. 44–52, Jun. 2007.

[74] C. Bizer and D. Maynard, "The semantic web challenge, 2010," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 9, no. 3, pp. 315+, Sep. 2011.

[75] T. Qin, T.-Y. Liu, J. Xu, and H. Li, "LETOR: A benchmark collection for research on learning to rank for information retrieval," *Information Retrieval*, vol. 13, no. 4, pp. 346–374, Aug. 2010.

[76] J. L. Vicedo and J. Gómez, "TREC: Experiment and evaluation in information retrieval: Book reviews," *J. Am. Soc. Inf. Sci. Technol.*, vol. 58, pp. 910–911, Apr. 2007.

[77] A. Ruttenberg, T. Clark, W. J. Bug, M. Samwald, O. Bodenreider, H. Chen, D. Doherty, K. Forsberg, Y. Gao *et al.*, "Advancing translational research with the semantic web," *BMC Bioinformatics*, vol. 8, no. S-3, pp. S2+, 2007.

[78] R. Sangireddy, H. Kim, and A. K. Somani, "Low-power high-performance reconfigurable computing cache architectures," *IEEE Trans. Computers*, vol. 53, no. 10, pp. 1274–1290, Oct. 2004.

[79] N. Petkov, *Systolic Parallel Processing.* New York, NY: Elsevier Science Inc., 1992.

[80] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing.* New York, NY: Springer Publishing Company, Inc, 2007.

[81] Y. Kim and R. N. Mahapatra, *Design of Low-Power Coarse-Grained Reconfigurable Architectures*, 1st ed. Boca Raton, FL: CRC Press, Dec. 2010.

[82] X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: A library for parallel simulation of Large-Scale wireless networks," in *Workshop on Parallel and Distributed Simulation*, 1998, pp. 154–161.

[83] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. Mccanne, K. Varadhan, Y. Xu, and H. Yu, "Advances in network simulation," *Computer*, vol. 33, no. 5, pp. 59–67, 2000.

[84] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY: McGraw-Hill Science, Engineering, Math, Mar. 1997.

[85] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed., ser. Prentice Hall series in Artificial Intelligence. Upper Saddle River, NJ: Prentice Hall, Dec. 2002.

[86] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates, Feb. 1986.

[87] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall, May 2008.

VITA

Name:	Suneil Mohan
Address:	Department of Computer Science and Engineering 301, H.R. Bright Building Texas A&M University TAMU-3112 College Station, TX 77843-3112
Email:	suneilmohan@tamu.edu
Education:	B.E. Electronics and Communication Engineering Anna University, Chennai, India. 2006
	Ph.D Computer Engineering Texas A&M University, TX, USA. 2012