HIGH PERFORMANCE INFORMATION FILTERING ON MANY-CORE PROCESSORS

A Dissertation

by

AALAP TRIPATHY

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

DOCTOR OF PHILOSOPHY

Chair of Committee,	Rabi N. Mahapatra
Committee Members,	Gwan S. Choi
	Yoonsuck Choe
	James Caverlee
Head of Department,	Nancy Amato

December 2013

Major Subject: Computer Engineering

ABSTRACT

The increasing amount of information accessible to a user digitally makes search difficult, time consuming and unsatisfactory. This has led to the development of active information filtering (recommendation) systems that learn a user's preference and filter out the most relevant information using sophisticated machine learning techniques. To be scalable and effective, such systems are currently deployed in cloud infrastructures consisting of general-purpose computers. The emergence of many-core processors as compute nodes in cloud infrastructures necessitates a revisit of the computational model, run-time, memory hierarchy and I/O pipelines to fully exploit available concurrency within these processors.

This research proposes algorithms & architectures to enhance the performance of content-based (CB) and collaborative information filtering (CF) on many-core processors. To validate these methods, we use Nvidia's Tesla, Fermi and Kepler GPUs and Intel's experimental single chip cloud computer (SCC) as the target platforms. We observe that ~290x speedup and up to 97% energy savings over conventional sequential approaches. Finally, we propose and validate a novel reconfigurable SoC architecture which combines the best features of GPUs & SCC. This has been validated to show ~98K speedup over SCC and ~15K speedup over GPU.

DEDICATION

To my parents

ACKNOWLEDGEMENTS

Working towards my Ph.D. and writing this dissertation would have been impossible without the guidance of many individuals who in their own unique ways helped me persevere through my graduate school experience. It is a great privilege to be able to convey my gratitude to you all in this document.

First and foremost, I would like to thank my advisor & committee chair, Prof. Mahapatra for believing in my ability, supporting and guiding me through the course of this research. I shall always remain indebted to my committee members, Prof. Choi, Prof. Choe and Prof. Caverlee, for their valuable feedback & guidance at critical points of this research and in completing my Ph.D.

I would not even have embarked on this journey had it not been for the firm belief and encouragement of my parents. I would like to thank them for their support, prayers and advice through the ups-and downs of the graduate school experience. Conversations with them helped me keep my sanity intact and reinforce the belief that I could get through it.

Many thanks are due to my present and former colleagues in the Embedded Systems Co-design Group especially Amitava, Suneil, Atish, Deam, Suman, Nikhil, Ron, Jagannath, Jason. Conversations with some you that began with “Do you have a minute?” led to debugging of a critical piece of code or pointing me in direction of a new approach, much of which forms part of this document. You have helped me understand

what research meant and how it should be done right. I shall forever remain indebted to you all.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience. Thank you Dave, Jeremy, Brad, Tony, Bruce and everyone at CSG who fixed things up and patiently answered the many hundred problems I may have come to you with over the years. Thank you for being so awesome! Thank you Tina, Marilyn, Sybil and rest of the Advising, Accounting and Administrative staff of the Department.

I also want to extend my gratitude to Carbon Design Systems, ARM Inc, Intel Inc. Nvidia Inc., Synopsys Inc., Mentor Graphics & Cypress Semiconductor Inc. who provided their tools, libraries and resources for research use.

Thank you Prof. Gutierrez, Prof. Song, Prof. Shell & Prof. Choe for having me as your Teaching Assistant for CSCE 483 & CSCE 312. Working for you has been an incredible experience, which I would not trade for another. You not only helped me realize what it takes to deliver quality teaching but also gave me the autonomy to shape it. I shall remain forever indebted to you for your support. Lastly, thanks Prerna for your patience and understanding during the final stretch.

NOMENCLATURE

API	Application Programming Interface
BF	Bloom Filter
CF	Collaborative Filtering
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFS	Distributed File System
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
JVM	Java Virtual Machine
MCPC	Management Controller Personal Computer
MPB	Message Passing Buffer
NFS	Networked File Sharing
$P_{false+ve}$	Probability of False Positives
RAM	Random Access Memory
SATA	Serial Advanced Technology Attachment
SCC	Single Chip Cloud Computer
SMP	Symmetric Multiprocessor
SoC	System-on-Chip
TF-IDF	Term Frequency – Inverse Document Frequency
TLP	Thread-level parallelism

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

1. INTRODUCTION

The era of achieving a faster and more capable uniprocessor every 18 months has now ended. Moore's law and Dennard's scaling rules, which defined the design, manufacture and marketing of microprocessors over the past five decades no longer holds true. Chip manufacturers have realized that it is advantageous to add more cores to a chip rather than increase clock speeds. Therefore, while multi-core processors are now common, many-core processors in cloud infrastructures are on the horizon.

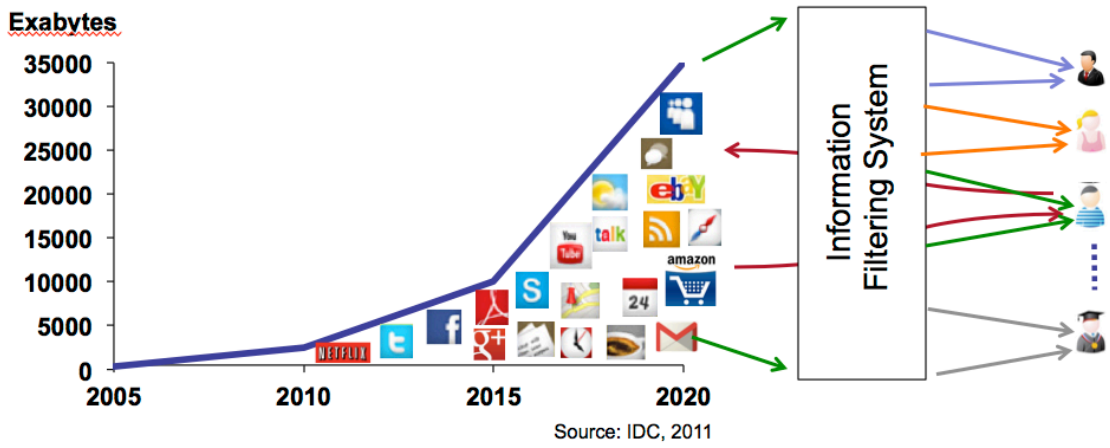

Figure 1. Information Filtering in the Petabyte Era

At the same time, the problems that we want to solve using these processors are increasing in size faster. This is especially true for problems in social network graphs and big-data problems. For instance, the amount of digital information created, captured or replicated worldwide is expected to reach 35000 Exabytes by 2020 [1] (Figure 1). It is

no longer sufficient to provide semantically accurate query results; users expect to be provided a somewhat intelligent choice of items based on their previously expressed preferences and those of "similar" users. This situation has led to the widespread use of information filtering systems most significant of which are Recommendation systems (RS) [2]. RS's learn about a user's preference and figure out the most relevant information for them using machine learning techniques. Driven by sophisticated algorithms, recommenders help consumers by selecting products they will probably like and might buy based on their past browsing, searches, purchases, and preferences. Almost all online e-commerce platforms deploy them to boost sales, drive and retain traffic; recommenders are already a huge and growing business.

A multitude of recommendation techniques involving content-based and collaborative filtering have been developed in literature and this remains an active area of investigation. Such techniques are useful only with several thousand GB's of training data and involve large-scale data-intensive computations to determine similarity between users and items, make predictions and recommendations. To be scalable, they are deployed in cloud infrastructures using distributed computing paradigms such as Mapreduce[3]. These frameworks split the computation into small partitions (key-value pairs), which can then execute independently on multiple nodes in parallel. A programmer has only to specify the serial parts necessary in transforming one key-value pair to another. An underlying parallel run-time is responsible for ensuring parallelization, inter-machine communication and failure-recovery.

Distributed computing frameworks such as MapReduce assume that compute nodes are single-core; limited extensions of these frameworks are available for shared memory multi-processor systems (multi-threaded). However, new challenges arise when a compute node is a many-core system; in itself capable of running a distributed computing run-time for enhanced performance. New highly parallel, energy-efficient many-core SoC architectures need to be created to run such information filtering applications on such compute nodes of the future.

The key aim of this research, therefore, is to explore the fit, energy efficiency of such many-core architectures in the design of recommendation systems and search engines in cloud-data centers of the future. The research challenges that this thesis addresses are:

1. How to map the data-intensive computational kernels of RS on Many-Core systems?
2. How to exploit concurrency within the machine boundary efficiently?
3. What is the appropriate application stack for RS on Many-Core processors?
4. How to alleviate the computation and communication bottlenecks for RS on Many-core processors available today?
5. How to design an energy-efficient parallel reconfigurable System-on-Chip (SoC) architecture for data-intensive information filtering applications?

2. BACKGROUND AND MOTIVATION

This chapter provides a high level overview of active information filters (recommendation systems), the algorithms used to realize them, an overview of many-core platforms and a literature review of the efforts in academia for high performance information filtering.

2.1 Recommendation Systems Overview

Recommendation systems are active information filters. They help users discover new "information" (items) based on their past (implicit) or expressed (explicit) preferences. They add predictions/ratings to the information flowing to a user enabling them to "discover" new information. We live in an increasingly social and real-time world, the number of things to recommend & users expressing opinions (tastes) over the web and mobile are growing exponentially. Algorithms deployed by the industry to make predictions are already known to be accurate to within 0.5 stars (on a 1-5 star scale), 75% of the time. However, retraining a dataset of 1.4 Bi ratings takes over 48-compute hours (Netflix's Cinematch Recommender System) [4]. With this background, we introduce the two major classes of recommendation engines in operation today – the first based on the content/meta-data associated with the information item (semantics), the second based on peer-ratings (collaborative).

2.1.1 Semantic Information Filtering

The key hypothesis in a content-based (semantic) recommendation system is: "Recommend items to a user which have highest similarity in attributes to items

previously seen by him". Search engines/information retrieval systems have traditionally deployed vector-based models as the underlying technology to calculate similarity, ignoring the semantics involved in representing a user's query (intention) or a document [5, 6]. Consequently two phrases such as: *"American woman likes Chinese food"* and *"Chinese woman likes American food"* are considered similar because they contain the same keywords although they refer to distinct *concepts*. To alleviate this problem, new techniques have been proposed to represent *composite meaning* in the semantic computing community [7]. They rely on tensors (multi-dimensional vectors) to represent and successfully discriminate between complex *concepts* [8]. However, they have been shown to increase the problem size super-exponentially [9]. Consequently search engines such as Google have only been able to employ these techniques in a limited manner [10]. Performing hundreds of thousands of semantic (tensor) comparisons with each pair consisting of vectors of a similar magnitude is therefore, a computationally challenging problem and requires the exploration of many-core compute clouds.

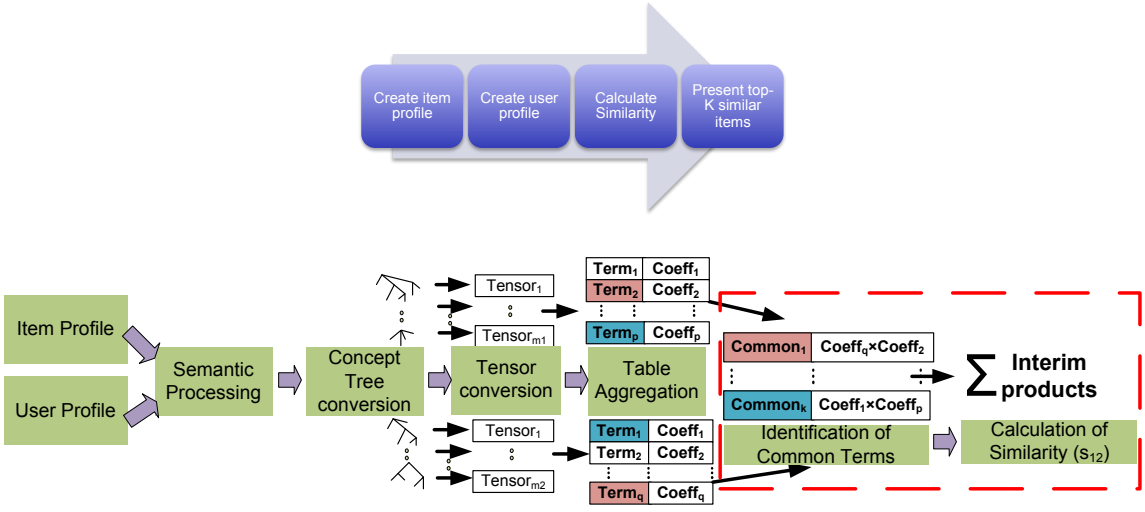

Figure 2. A Semantic Information Filtering System

Figure 2 describes the computational flow of a semantic information filtering system. It consists of four key stages – (1) creation of item profile, (2) creation of user profile, (3) calculation of similarity and (4) presentation of Top-K most similar items as recommendation. An item profile and a user profile are typically created from unstructured meta-data describing the information item. Unstructured data is then semantically processed using a natural language tool-chain to obtain their concept (syntactic/ontology) tree representations (involves sentence segmentation, tokenization, part-of-speech tagging, entity & relation detection) [11]. The leaves of such a concept tree contain *terms* whereas the tree itself encodes *semantics* or *meaning*. Mathematically, a concept tree is a hierarchical n-ary tree where the leaf nodes represent terms and the tree itself describes their inter-relationships (semantics) within the original document. Concept trees are abstract mathematical structures and can have arbitrary structures; therefore are unsuitable for further processing in a fine-grained manner. These trees undergo further transformation into an equivalent tensor representation using rules defined in [7, 8, 12] without loss of any semantic content (Tensor conversion phase in Figure 2). The tensor form can be represented as a table of terms and coefficients. The coefficients denote the relative importance of each term in describing an item. It is easy to see that the tensor representation of an item/user profile can be arbitrarily large. The final stage involves the computation of a semantic similarity score. This proceeds first with the identification of common terms in the user and item tensors followed by the multiplication of their corresponding coefficients and summation of these interim

products. The top-K items with the highest similarity to a user's profile are returned as recommendations to a user.

2.1.2 Collaborative Information Filtering

A collaborative information filtering system takes an alternate approach. It makes no assumptions about the availability of additional information/meta-data describing the item or a user. Instead a large number of ratings are available as input, which could have been collected via explicit or implicit means. The key hypothesis is – "similar users tend to like similar items". A similarity score is computed by mining existing user-item relationships. For example, two items are considered similar if a large number of users who have liked one item have also expressly liked (have given higher ratings) for another.

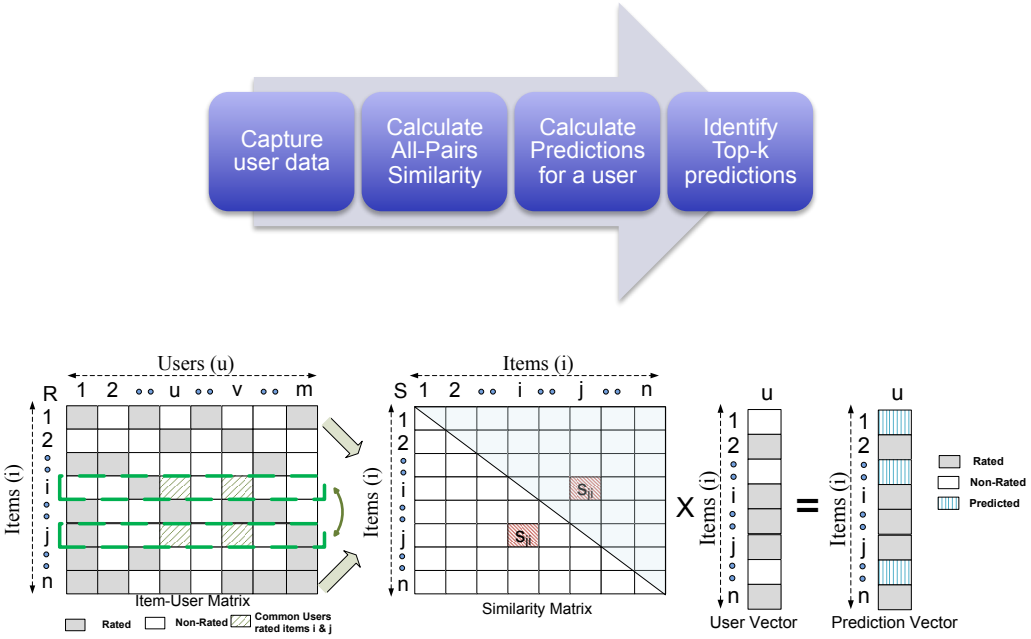

Figure 3. A Neighborhood-based Collaborative Information Filtering System

Figure 3 shows the flow of a CF recommendation system. It consists of four key stages – (1) capture user data, (2) calculation of all-pairs similarity, (3) calculation of prediction and (4) identification of Top-K predictions. This thesis focuses on stages (2)-(4) which we discuss with the aid of a mathematical model.

Suppose there exist a set of m distinct users; $U = \{u_1, u_2, \ldots, u_m\}$ who have rated one or more of n distinct items; $I = \{i_1, i_2, \ldots, i_n\}$. Each user u provides a rating r_{ui} for a subset of items in I (I_u), shown as grayed boxes in Figure 3. These ratings can be represented as an item-user matrix R of size nxm. Not all elements in this matrix will be filled in because most users will not have rated all items. Computation of s_{ij} proceeds by first identifying the set of users who have rated both i & j and then calculating a similarity metric (such as Pearson's correlation coefficient). Once similarity for all i,j is available, the prediction for a user u for all unrated items can be obtained through a similarity matrix – user rating vector product (Figure 3). Once a prediction vector for a user is obtained the top-K highest items are returned as recommendations. This computation is effective only for large datasets and is both data and compute intensive.

2.2 Many-core Platforms Overview

Increasing power consumption and complexity in design and verification has driven the microprocessor industry to integrate multiple cores on a single die. Dual, quad and oct-core processors are now common in the market now, available technology permits integrating 1000's of cores on a single die/platform – a many core future. Consequently, several prototype architectures have been designed in industry and academia. They can be broadly classified into the following design styles [13]: (1) a

symmetric many-core processor that replicates a state-of the art superscalar processor on a die, (2) a symmetric many-core processor that replicates a smaller, more power-efficient core on a die and (3) an asymmetric many-core processor with numerous efficient cores and one superscalar processor as the host processor. High-end multicore processor vendors such as Intel & AMD have chosen the first model in building early prototypes whereas conventional graphics processing units have been adapted to operate using the second approach. In this thesis, we first attempt to extract maximum available parallelism from existing/prototype many-core processors for data intensive applications and then use our findings to propose a new many core architecture that can provide higher performance than those proposed so far. We will now discuss a brief overview of the architecture and programming model of two representative many-core processors evaluated in this thesis.

2.2.1 Graphical Processing Units (GPUs)

GPU's have continued to evolve for general-purpose usage as application-coprocessors and are now widely used in high performance computing due to exceptional floating-point performance, memory bandwidth and power efficiency. GPU manufacturers have released API's and programming models for application development and analysis – Nvidia's implementation is Compute Unified Device Architecture (CUDA) [14] , AMD's implementation is marketed under the name AMD Firestream [15]. One of the goals of this dissertation is to design an efficient computational technique to exploit available fine-grained parallelism on different GPU architectures, analyse their parallel performance for information filtering applications

discussed above. In this dissertation, we run our algorithms on three families of on Nvidia GPU's – Tesla [14], Fermi [16] & Kepler [17]. They key architectural differences between the three families relevant to this dissertation are summarized in Table 1:

Table 1. Key Architectural Features of Nvidia's Tesla, Fermi & Kepler GPUs

Characteristics	***C870 (Tesla)***	***Quadro 2000M (Fermi)***	***GTX 680 (Kepler)***
# of Cores	240	512	1536
Base Clock (MHz)	648	772	1006
Memory clock (MHz)	2484	4008	6008
Memory B/W (Gbps)	76.8	192.4	192.26
TDP (W)	170.0	244	195
# of Transistors (Bi)	3.0	3.2	3.54
Rated GFlops	1063	1581	3090
# of Texture Units	80	64	128

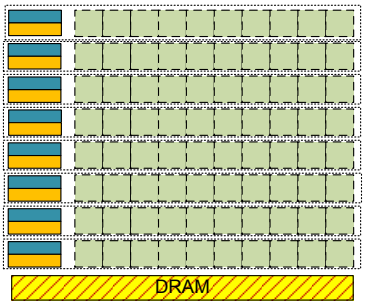

Figure 4. GPU Architecture for General-purpose Computing (GPGPU)

Figure 4 shows the generic architecture of a GPU (subtle differences between families not shown). The functional units are shown in light green (called streaming processors) and consist of several ALU's an instruction dispatch and data collector unit.

A horizontal block with several streaming processors (shown with dotted lines) is a streaming multiprocessor and additionally contains a control unit (blue) and cache (orange). There is a common shared off-chip DRAM (yellow & orange hatches), which is termed global memory. The key operating principle of GPU's is to run numerous simple threads concurrently, not use cache at cores and use massive thread-level parallelism to mask memory latency. Massive TLP enables a GPU control until to run other threads when some are stalled waiting for memory.

2.2.2 Single Chip Cloud Computer (SCC)

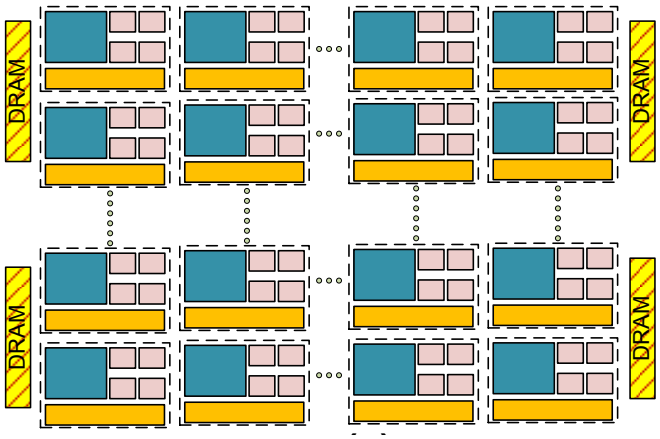

Figure 5. Cluster-on-chip Architecture (Intel SCC, Xeon Phi)

Intel's Single Chip Cloud computer is an experimental research/concept chip made available to select Universities under the Many-Core Applications Research program. It has recently been commercialized as the Intel Xeon Phi. It consists of 48 Intel Architecture (IA) P54C cores on a single die connected by an on-chip mesh for

inter-core communication. It can address up to 64 GB RAM and shares traits of both message passing and shared memory architectures. Data I/O is achieved via an off-chip FPGA that delivers packets from Ethernet/PCIe bus directly onto the on-chip network. Unlike a GPU, an SCC need not be run as a co-processor. Each of the cores can run a Linux operating system image and run application programs (written in C/C++). More information on Intel SCC architecture is available at [18]. Figure 5 shows the architecture of the Intel SCC. Each block marked in dotted lines represents a P54C core consisting of instruction dispatch/control unit, ALU and cache. A group of cores share a common memory controller for DRAM access. Unlike a GPU, the SCC architecture uses caches to mask memory access latency at individual cores. Since, memory coherency is difficult to achieve in hardware/software between 100's of cores, this has not been implemented on the SCC necessitating the design of application software using a distributed memory-programming model.

2.3 Research Overview

The research leading to this dissertation is presented in the following manner:

1. To design suitable data structures and computational algorithm for content-based semantic information filtering (CB)
    a. Validate algorithm's energy efficiency on GPU & SCC, perform detailed architectural profiling to determine performance bottlenecks
    b. Use those insights to redefine computational algorithm or propose change in many-core architecture.

2. To design suitable data structures and computational algorithm for neighborhood-based collaborative information filtering (CF)
    a. Validate algorithm's energy efficiency on GPU & SCC, perform architectural profiling to determine performance bottlenecks.
    b. use insights obtained to redefine computational algorithm or propose changes in many-core architecture.
3. To design an appropriate distributed memory run-time to run on a many-core chip. Map the CF computational model to this run-time.
    a. Determine whether the distributed memory-computing model on the SCC is more energy efficient than Objective 2, perform stage-wise profiling to determine bottlenecks, determine the trade-off between computation & communication time for the application.
    b. If computation is determined to be a bottleneck, design lightweight IP cores, which could efficiently perform the same operation than a Pentium P54C core (on the SCC).
4. To design an energy-efficient parallel reconfigurable SoC architecture to run CB and CF algorithms
    a. This SoC should leverage the best features from existing many-core architectures and propose hardware designs to alleviate identified bottlenecks.

b. Design a Network-on-chip based interconnects for low latency, high bandwidth data I/O between memory & processing elements and between several processing elements.
c. Design lightweight application-specific functional units that consume least die area and dissipate least power. The lightweight cores should be reconfigurable and at the least offer specific programmability.

3. SEMANTIC INFORMATION FILTERING ON MIC*

3.1 Shared Memory Approach

In this dissertation, we describe the design of a novel BF based computational kernel to compute semantic similarity on Nvidia's Tesla GPU & Intel's SCC [19]. This involved the creation of new data structures, validation of proposed algorithm when using a common shared memory between participating cores.

3.1.1 Motivation

Figure 6 shows a high-level overview of the semantic comparison methodology. Two example statements/descriptors need to be compared. This involves (1) conversion from textual representation into a concept tree, (2) conversion of concept tree into a tensor form and (3) representation of the tensor in a tabular form (semantic descriptor). The section marked in blue involves the comparison of two semantic descriptors and will be the focus of this chapter of this dissertation.

We assume that two tensors (sizes n_1, n_2) are provided as input. The similarity metric used is cosine similarity; which will lie in [0, k] provided the coefficients are normalized within their respective tensors.

---

* Parts of this section have been reprinted with permission from A. Tripathy, S. Mohan and R. Mahapatra, "Optimizing a Semantic Comparator using CUDA- enabled Graphics Hardware", in 5th IEEE International Conference on Semantic Computing (ICSC) September 18-22, 2011, Palo Alto, CA, USA. © IEEE 2011

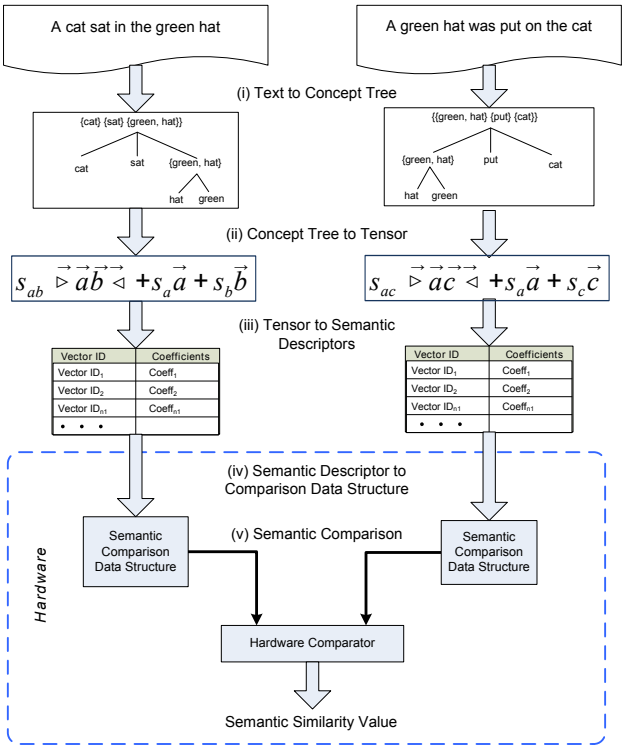

Figure 6. Semantic Comparison Methodology

We note that for a given a tensor pair; identification of common terms, multiplication of their coefficients, and summation of interim terms are independent atomic operations and will benefit from parallelism. Conventional sequential processors (including SMP systems) cannot provide the necessary parallelism available in this computation. In the following section, we discuss how a GPU based semantic comparator is designed to lower query latencies to acceptable, interactive levels.

3.1.2 Bloom Filters for Set Intersection

A Bloom filter [20] is a probabilistic, space efficient data structure that enables a compact representation of a set and computation of intersection between two sets. It is described in the form of an *n*-bit long bit vector. Elements of one set are inserted into an empty Bloom filter (all index positions "0") using *k* independent hash functions. The hash functions generate *k* distinct index values for an item which are turned "1". The resulting bit-vector after inserting *m* elements of a set is a compact representation of the set. Elements of a second set can now be tested against this Bloom filter using an analogous method. If all index positions due to a tested element are "1", it is considered to be *positive* result.

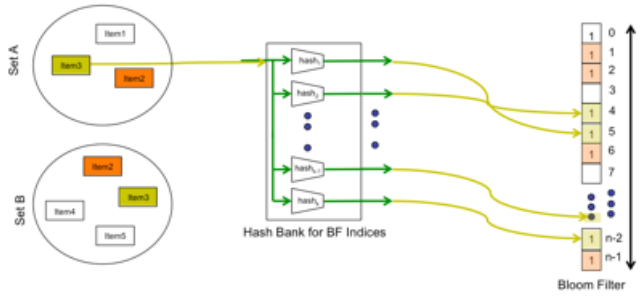

Figure 7. Bloom Filter (BF) Insertion

Figure 7 describes a situation where *m* items: $item_1$, $item_2$, .. $item_m$ from Set A are inserted into the Bloom filter. A bank of *k* hash functions operates on a string representation of an item to produce *k* distinct index values; each of which are turned "1"

in the Bloom filter. For example, $item_3$, turns the BF[4]=BF[5]=BF[27]=BF[n-2] = 1. It is important to note that this insertion operation can be performed in parallel (one per processing element or thread) provided the Bloom filter is located in shared memory.

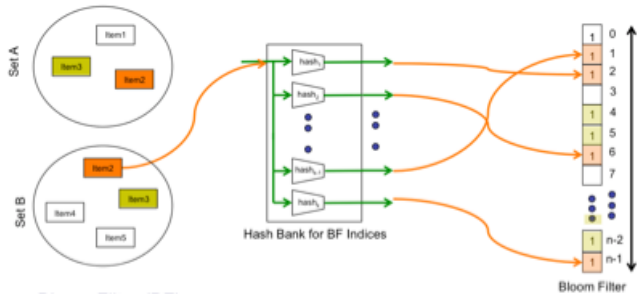

Figure 8. Bloom Filter (BF) Lookup

Figure 8 describes an analogous situation where elements of the second set (Set B) are tested with the Bloom filter created earlier. For simplicity of explanation, we have shown that two elements ($item_2$ & $item_3$) are actually common between Sets A&B. When $item_2$ is passed through the hash functions, they generate the exact same index positions as before. All of them are guaranteed to have been turned on earlier (BF[1]=BF[2]=BF[6]=BF[n-1]="1"). If all the index positions for an element of the test set (Set B) return a true from the BF, we claim that this element is present in A∩B.

Testing whether an arbitrary element of the test set is present can result in false positives (return true, when the element is not present). It is guaranteed to never return a

false negative (return false, when an element is actually present). The probability of false positives ($p_{false+ve}$) is given by:

$$p_{false+ve} = (1 - \left[1 - \frac{1}{n}\right]^{km})^k \approx (1 - e^{\frac{-km}{n}})^k$$

The probability of false +ve can be minimized by choosing appropriate values of size of BF (n), number of hash functions used (k), given that the number of distinct elements likely to be inserted in either set is m. Also, for a given dimensions of n & m, an optimum value of k can be determined (differentiating w.r.t. k) as:

$$k_{optimum} \approx 0.7 \times \frac{n}{m}$$

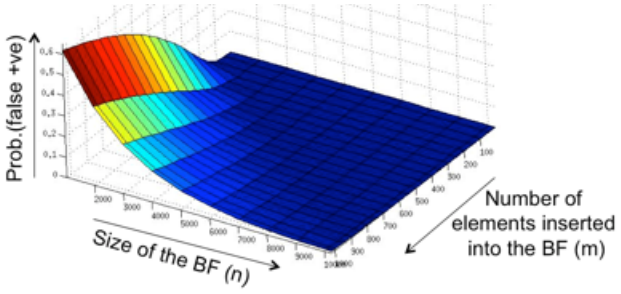

Figure 9. Probability of False Positives in a Bloom Filter

Figure 9 shows the variation of $p_{false+ve}$ with varying n and m for $k_{optimum}$. This shows that $p_{false+ve}$ can be made arbitrarily low for a given number of elements to be inserted (m) by choosing a large size for the BF bit vector (n). For all our experiments in

this thesis, we choose the target $p_{false+ve}$ to be 0.001 and calculate $n=f(m)$ as per the equation shown earlier.

3.1.3 Phases of Semantic Comparison Kernel

The semantic comparison kernel (SCK) Algorithm is implemented in four phases. In Phase 1, the coefficient tables are copied to CUDA Global Memory. In Phase 2, document tensor is encoded into the Bloom Filter using hashing. In Phase 3, the query tensor is tested with this Bloom filter to determine common terms. In Phase 4, the scalar coefficients corresponding to the filtered tensor are extracted, multiplied and summed to generate a similarity value.

3.1.3.1 Phase A – Memory Copy to GPU

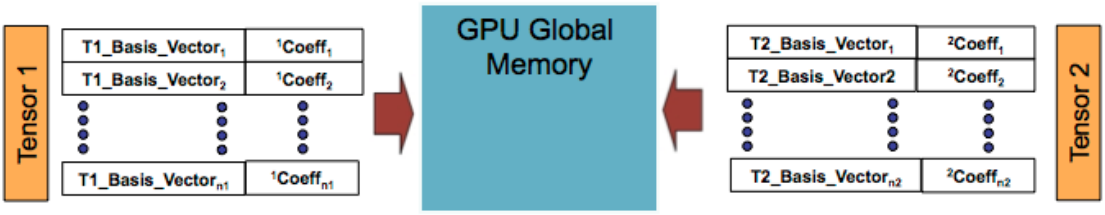

Figure 10. Coalesced Copy from CPU to GPU Global Memory

In the first phase of the computation, Tensor 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 Figure 10. The

transformation into a coalesced memory layout ensures maximum usage of available of PCIe bandwidth for the GPU architectures we experiment with

3.1.3.2 Phase B – Encode Tensor 1 in Bloom Filter

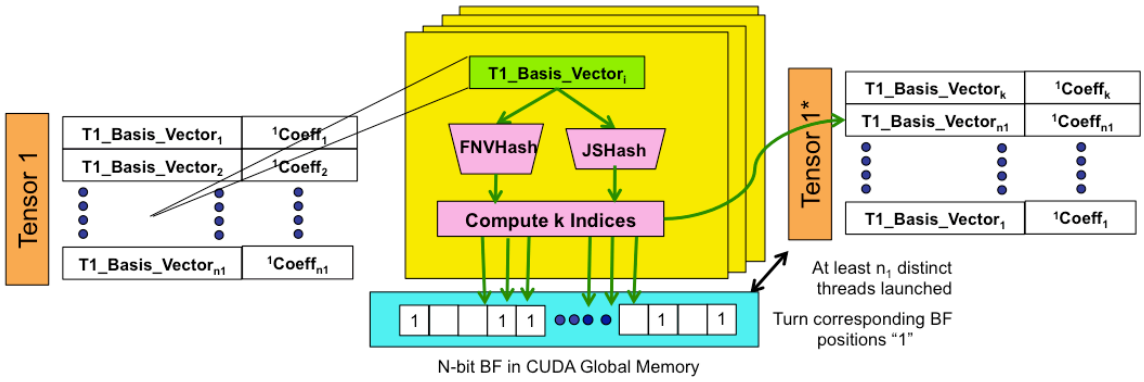

Figure 11. Encoding Tensor1 in Bloom Filter

In this phase, we encode the contents of $Tensor_1$ into the Bloom Filter. This is performed using a number of concurrent kernels ($=n_1$) that run on the CUDA streaming cores. In each kernel, a given *T1_Basis_Vector*$_i$ is encoded into the Bloom Filter using two hash functions. As shown in Figure 11, we make n_1 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). The Bloom

filter bit vector is initially created in CUDA Global memory. For use in the subsequent stage, we transfer it to the GPU's texture cache.

The *k* index positions for every tensor basis vector string ($T1\_Basis\_Vector_i$) is used to organize Tensor1 into a parallel hash table in global memory (Cuckoo hashing[21]). This mechanism will be used to be able to lookup the corresponding coefficient of $T1\_Basis\_Vector_i$ in the subsequent stages if it is determined to be a matching term. This mechanism is used instead of sorting $Tensor_1$ and accessing it via binary search. This is because binary search of a sorted tensor of size n_1 will be expected to have $\log_2(n_1)$ probes in the worst case whereas lookup from a hash table can be made have O(1) number of probes. In addition the process of creating Bloom filter representations for every basis vector term already produces hash positions.

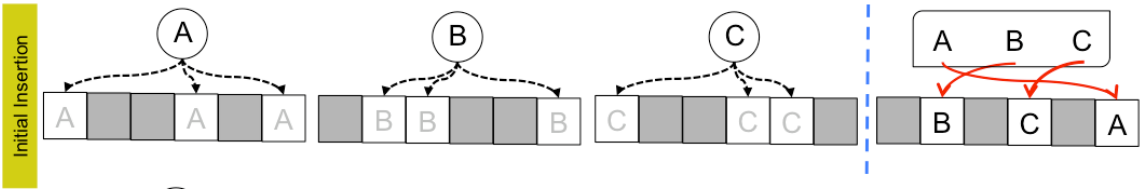

Figure 12. Initial Insertion of Tensor Terms into Hash Table

Figure 12 shows the mechanism for inserting three elements the hash table. Elements marked *A,B,C* have the potential positions marked in gray (indicated by *k* index values). In this case, we store the ($T1\_Basis\_Vector_i$,$Coeff_i$) in consecutive memory locations.

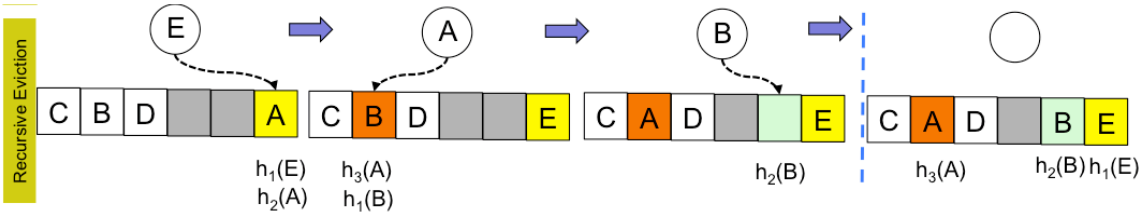

Figure 13. Recursive Eviction Mechanism in Cuckoo Hashing

Figure 13 shows the mechanism for conflict resolution in Cuckoo hashing. For instance let's assume that elements A,B,C,D are already inserted. A new element E is to be inserted and its first hash index $h_1(E)$ indicates a position which is already occupied (shown as A in this example). Our algorithm which is based on [22] follows a greedy approach immediately evicting A and placing it in a reinsertion queue. The same thread performs the reinsertion by looking into the next available position for item A, Since $h_2(A)$ was used, the next position indicated by $h_3(A)$ will be chosen next. In the example in Figure 13, the subsequent position was also occupied, but A is inserted in this position, thereby evicting B. Again B is placed in the reinsertion queue until a free location is provided. In case the k^{th} position of an item is reached, the algorithm would round-robin back to position 1. This procedure ensures that every item inserted in the hash table can be located with a fixed number of probes keeping the number of memory access per retrieval low. The creation of the hash table is an efficient operation because separate threads can insert keys at the same time. Finally, we are taking reusing the hash functions used to probabilistically determine bloom filter indices to determine positions. The only cost associated with this approach is the additional space required to create the

table. We also experimentally determined that a memory allocation of 1.25xSize of Tensor1 was sufficient to store the tensors and that any item can be found after at most *k* probes leading to dramatic improvements in Phase D to be described subsequently.

3.1.3.3 Phase C – Encode and Test Tensor 2 with Bloom Filter

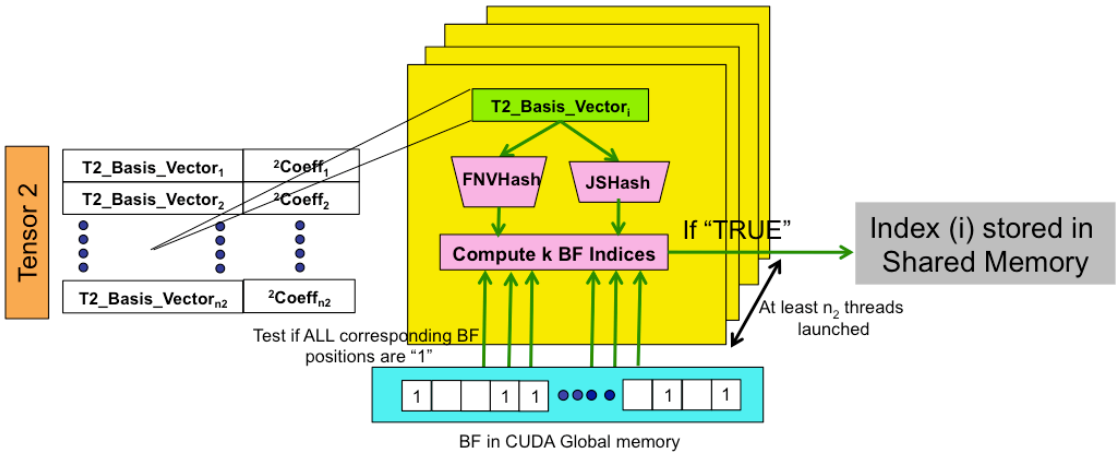

Figure 14. Testing Tensor2 with Bloom Filter

Figure 14 shows the process of testing Tensor 2 with the Bloom filter placed in CUDA global memory in Phase B. This stage is similar to Phase B with two differences:

- Instead of setting a bit position in the Bloom Filter to 1, it tests for the presence (or absence) of the *T2_Basis_Vector*$_i$ that the kernel is operating on.
- If all the bits indicated by the BF Index for a given *T2_Basis_Vector*$_i$ 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 at least n_2 concurrent kernels during 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" in shared memory.

3.1.3.4 Phase D – Compute Intermediate Sum and Perform Parallel Reduction

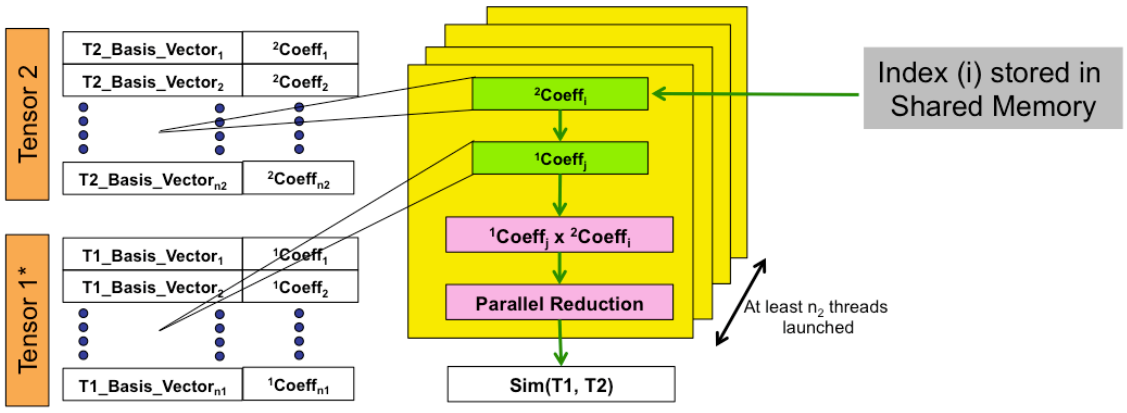

Figure 15. Computation of Intermediate Sum & Parallel Reduction

In this phase (Figure 15), we need to lookup and multiply the corresponding coefficients of the filtered elements (indexes of which were obtained in Phase C) from Table-1 and Table2. Phases A-C have enabled the "filtering" (identification) of the common basis vectors in nearly O(1) time. This stage begins by fetching the index of matching $T2\_Basis\_Vector_i$ and using it to fetch its coefficient from Tensor Table 2 i.e. $^2Coeff_i$ (located at an offset of 64 bits, single memory lookup operation). We now know that its corresponding coefficient is likely (possibility of a false +ve still exists) present in Tensor Table1. This can be obtained using a linear lookup from the original Tensor1

or from its DHT representation as described in the previous section. Fetching from the DHT representation of Tensor 1 can also be accomplished with a limited number of probes and is expected to be more efficient as described earlier.

Once the corresponding coefficients have been identified ($Coeff_1$, $Coeff_2$), we generate an interim product (Step-4). The final step in this stage involves the summation of the interim products from each stage. This is done using the parallel reduction primitive discussed in [23].

3.1.4 Experimental Setup

The goal of the experiment is to experiment performance, energy efficiency of the proposed algorithm with a synthetic dataset on different contemporary GPU architectures as compared to the best known CPU algorithm and architecture. The baseline CPU used had an Intel Core i7-3770K processor. Three kinds of GPUs were used to test the performance for semantic search - Tesla C870, Fermi Quadro 2000M, Kepler GTX 680. The semantic comparator on GPU was implemented in C++ using CUDA 5 for device programming. Each basis vector term of the tensor is represented by a unique 64-bit word. Each phase discussed above is implemented as a separate kernel and executed sequentially. The Bloom filter is represented by a byte array whose size depends on the tensor size. We experienced that bloom size should be at least one byte per term i.e. a tensor of size 10K resulting in a 10Kb bloom filter. The number of hash functions used is k=7.

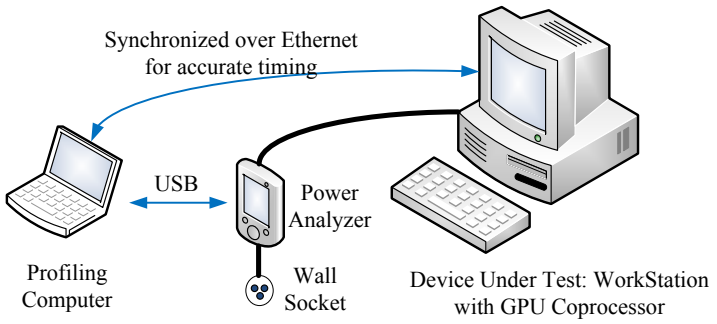

Figure 16. Equipment Setup for Power Profiling

The power monitoring was done using Watts' Up Pro power analyzer from Electronic Educational devices. This measures overall system power consumption. This device is connected in line with the power supply to the host computer as shown in Figure 16. Table 2 reports system base power (no GPU installed) and with the GPU in cold shutdown and idle. We recognized that once the GPU (Tesla C870) is initiated once but not used subsequently, it reverts into a idle power state which is higher than in cold shutdown.

Table 2. System Baseline Power Measurements

System Base Power	115W
System Idle Power (GPU cold shutdown)	150W
System Idle Power (GPU Awake, Idle)	186W
GPU Idle Power	36W

Algorithm 1 lists the pseudo code for the semantic comparator used for our baseline measurements on the CPU: function *computeTensorProduct()*. The constituent

basis vectors from tensor Table1 are inserted as pairs into a Red-Black tree data structure (*rbtree*, line 2) leading to the creation of a self-balancing binary search tree in $O(\log n_1)$ time. Next, the constituent basis vectors of tensor Table2 are tested with this BST (line 5). If this test is successful, then *Table2_BasisVector[j]* is a common basis vector (Line 6). Now their corresponding coefficients are pair-wise multiplied and summed (Line 7). The overall complexity for the latter search operation is $O(n_2 \log n_1)$. The balanced binary tree forms the basis for the C++ STL Map container and is expected to be the most efficient implementation for SMP systems.

Algorithm 1: Semantic Comparator on Conventional Symmetric Multiprocessors

inputs: Tensor Table1, Tensor Table2
output: Semantic (dot) product
function *computeTensorProduct*()
1: **for** i ← 0, (n_1-1) **do**
2: rbtree.insert(<*Table1_BasisVector*[i], *Table1_Coeff*[i]>)
3: **end for**
4: **for** j ← 0, (n_2-1) **do**
5: rbtree_ptr ← rbtree.find(*Table2_BasisVector*[j])
6: **if** rbtree_ptr != NULL **then**
7: *dot* ← *dot* + (*Table2_Coeff*[j] x *rbtree_ptr.value*)
8: **end if**
9: **end for**
10: return *dot*
end function

3.1.4 Results and Discussion

In this section, we discuss the execution time and energy consumption obtained due to the proposed algorithms running on three Nvidia GPU families over the state of the art serial implementation. We also profile the semantic kernels to examine which

stage remains a bottleneck and examine how our proposed algorithms can help alleviate it.

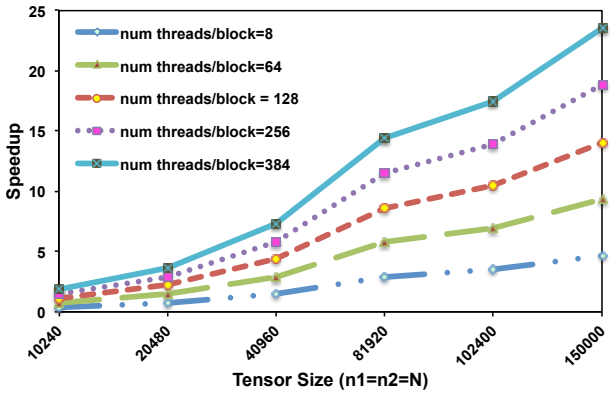

Figure 17. Speedup of Semantic Kernel (Linear Lookup)

Figure 17 shows the variation of speedup with varying tensor size and number of threads in use on an Nvidia GTX680. When we reach the maximum number of threads/block (=384) available on the Tesla GPU, we see a speedup of up to ~25x.

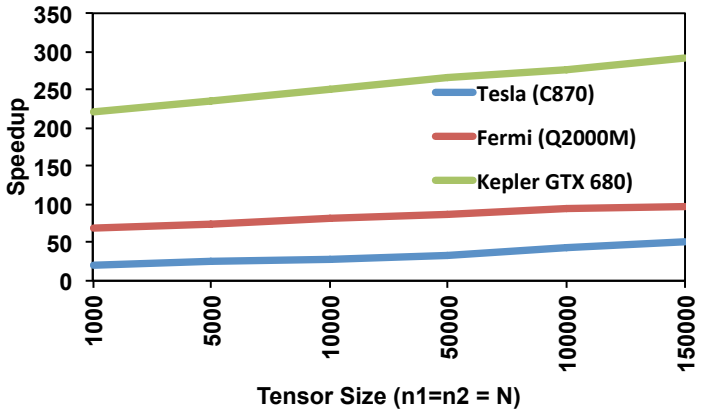

Figure 18. Speedup of Semantic Kernel (Cuckoo Hashing)

Figure 18 in contrast shows the speedup possible with varying tensor size for the maximum number of threads in use for Tesla, Fermi and Kepler GPUs with the same baseline. In this case, we can observe a speedup of up to ~50x for the Tesla GPU as compared to ~25x in Figure 17. This shows that our choice of Cuckoo hashing is appropriate for the lookup of the matching coefficients in Phase D of the proposed algorithm.

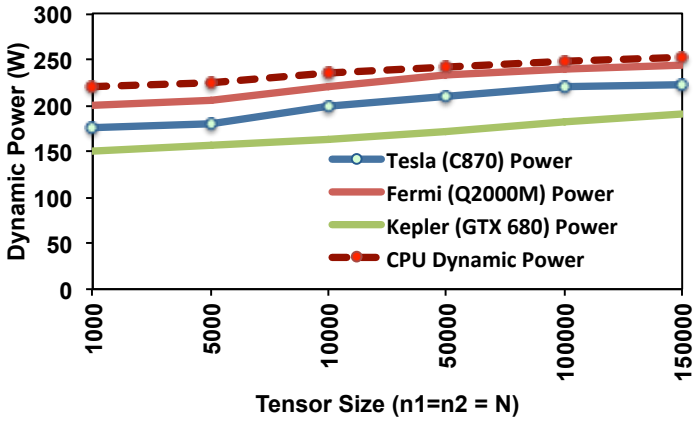

Figure 19. Power Consumption with Proposed Algorithm

Figure 19 shows the bulk power consumption for the Tesla, Fermi and Kepler GPUs when running the proposed algorithms. No significant difference was observed between the linear and cuckoo hashing based lookup variations in Phase D. This is because GPU power consumption is directly proportional to the number of cores in use. It is also well known that GPUs are energy efficient but not necessarily power efficient –

GPU power approaches that of the CPU for larger data sizes. In addition, we also observe that GPU dynamic power for the Fermi family (released 2011) > Tesla (released 2010) > Kepler (release 2013) for the same algorithm and data sizes.

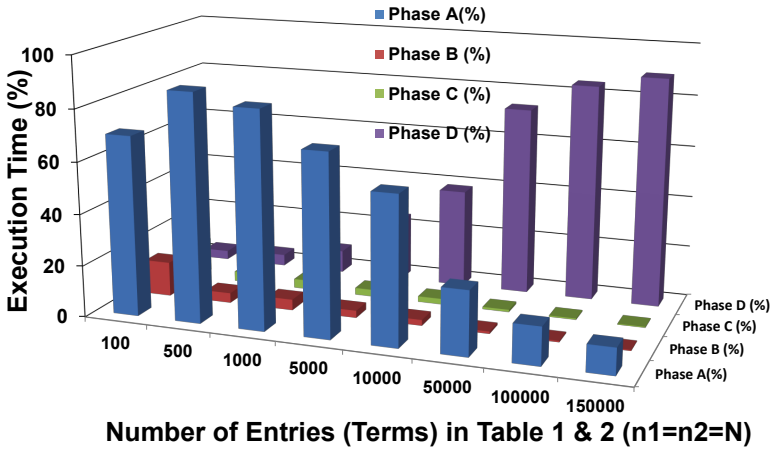

Figure 20. Profiling Semantic Kernels with Linear Lookup

Figure 20 shows a phase-wise profiling of the execution time for the proposed algorithm with linear lookups in Phase D. This was done to understand which phase consumes the highest fraction of total execution time. It conveys the following information: (1) Filtering of the common basis vector terms (Phases 2, 3) take up an almost negligible fraction of the time, (2) IO (Phase 1) to and from the GPU is a bottleneck, (3) coefficient lookup (Phase 4) occupies the largest fraction of the total execution time (almost 80%). We alleviate this problem using the Cuckoo Hashing algorithm to reorganize Tensor1 in a DHT as described earlier in this section.

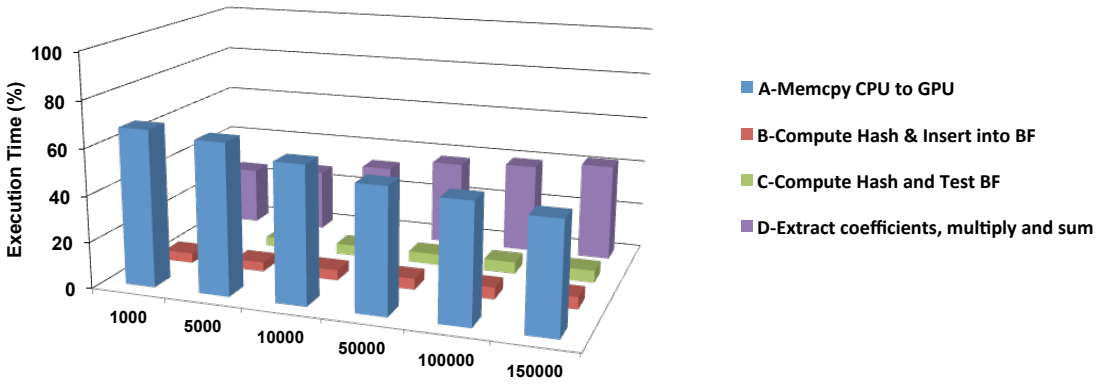

Figure 21. Profiling Semantic Kernels with Hash Lookup

Figure 21 shows a similar phase-wise profile generated for the Cuckoo hashing variation of the proposed algorithm. It shows that Phase D consumes a maximum of 40% of the execution time for the largest tensor size (=150k) which is significantly lower than the nearly ~90% observed with the linear lookup. This has enabled the realization of almost 50x speedup as opposed to 25x for the Tesla GPUs. Memory copy from the CPU to GPU remains a bottleneck with almost 50% of the execution time. Phases B & C which involve computation of the hash functions and reorganization of $Tensor_1$ in a DHT continues to occupy minimal fraction of the execution time. The profile is similar for the three GPU families – Tesla, Fermi & Kepler.

3.1.5 Related Work

To the best of our knowledge, this is the first work of its kind to enable semantic comparison on Nvidia's Tesla, Kepler and Fermi architectures. Our prior work [9] was only able to demonstrate a limited 4x speedup on the Tesla architectures for tensors of up to 150k entries. There have been several other attempts to accelerate traditional

keyword based search mechanisms on GPU's. GPUminer [24] describes the implementation of k-means clustering and frequent pattern mining. Likewise [25] describes the process of adapting a document similarity classifier on two massively multi-core platforms: Tilera 64-core SoC and the Xilinx Virtex 5-LX FPGA. This work has been done in the context of web-security and demonstrates that an incoming data stream can be filtered using a TF-IDF based dictionary of known attack patterns. This work is interesting because it uses large array of Bloom filters, with each element representing a data-value in the dictionary. They have been able to demonstrate scalability of up 30x on the Tilera 64-core SoC and up to 166x using an FPGA. In sacrificing the accuracy of the similarity computation, they have been able to demonstrate up to 80x speedup over a serial software approach. This paper substantiates our claim of Bloom filters being an appropriate mechanism to quickly identify common terms. We differ from this work in that our Bloom filters are created dynamically for every pair of comparisons. Likewise, our previous work, [26] discusses the design of a fine-grained parallel ASIC for semantic comparison. While this custom ASIC design has demonstrated hypothetical similarity of up to 10^5 using a Bloom filter based algorithm, I/O issues have not been taken into consideration. Further the scalability has been demonstrated only with tensors of sizes <10240 elements. A recent work on FPGAs [27] uses a Bloom filtering approach for accelerating traditional bag-of-words search and has been able to demonstrate a comparable 20-40x speedup over the best known serial implementation on multi-core CPUs. Our proposed approach with Cuckoo hashing has

superior speedup performance on a GPU and can be expected to improve the performance on an FPGA as well.

3.1.6 Section Summary

We have designed a novel multi-stage Bloom-filter based computational kernel to compute item-user semantic profile similarity on GPUs [9, 28]. We use Bloom filters [20] to quickly determine the common terms in two tensors. Once the matching terms have been identified, it is necessary to lookup their corresponding coefficients from memory. In case of Fermi and Tesla architectures, we radix sort Tensor1 whereas in case of Kepler architecture, we use Cuckoo hashing. This demonstrates superiority of shared-memory approach for CB when run on many-core processors. The baseline CPU used for comparison of results was the Intel Core i7-3770K processor. Three kinds of GPUs were used to test the performance for semantic search - Tesla C870, Fermi Quadro 2000M, Kepler GTX 680.

4. COLLABORATIVE INFORMATION FILTERING ON MIC *

4.1 Shared Memory Approach

In this dissertation, we describe the design of a novel counting based computational kernel to compute all-pairs similarity on Nvidia's Tesla GPU & Intel's SCC [19]. This involved the creation of new data structures to index the data, formulation of the all-pairs computation problem as a counting task amenable to efficient parallelization. We describe the motivation, proposed algorithm and the results in the following subsections.

4.1.1 Motivation

The workflow for item-item CF can be described mathematically as follows. Suppose there exist a set of m distinct users; $U = \{u_1, u_2, \ldots, u_m\}$ who have rated one or more of n distinct items; $I = \{i_1, i_2, \ldots, i_n\}$. Each user $u \in U$ provides a rating r_{ui} for a subset of items in I ($I_u \subseteq I$). These ratings can be represented as an item-user matrix R of size nxm. Not all elements in this matrix will be filled in because most users will not have rated all items. Let the total number of ratings provided as input i.e. triples of type (*itemID*, *userID*, *rating*) be T ($<< nxm$). A row vector from R (R_i) is sufficient to describe an item i's interaction history (i.e. all user ratings made for it) whereas a

---

* Parts of this section have been reprinted with permission from A. Tripathy, S. Mohan and R. Mahapatra, "Optimizing a Collaborative Filtering Recommender for Many-Core Processors", in 6th IEEE International Conference on Semantic Computing (ICSC) September 19-21, 2012, Palermo, Italy. © IEEE 2012 &
A. Tripathy, A. Patra, S. Mohan and R. Mahapatra, "Distributed Collaborative Filtering on a Single Chip Cloud Computer", in Proc. IEEE Intl. Conf. on Cloud Engineering (IC2E '13), Mar. 25, 2013, San Francisco, CA, USA. © IEEE 2013.

column vector (R_u) describes a user *u*'s interaction history (i.e. all item ratings made by him).

4.1.1.1 Calculation of Item-Item All Pairs Similarity (s_{ij})

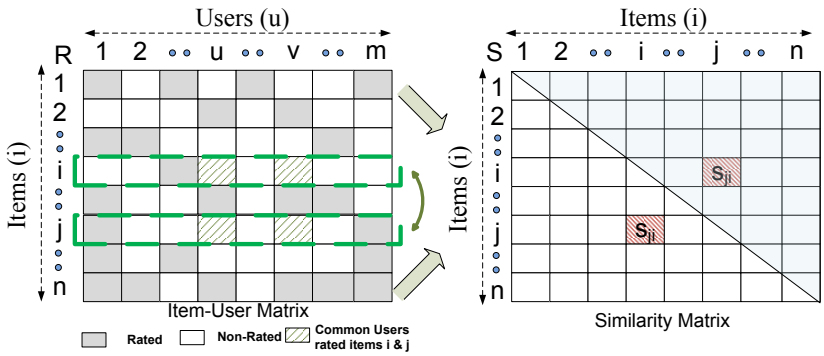

Figure 22. Calculation of Item-Item All Pairs Similarity

Figure 22 shows a sparse item-user matrix R_{nxm} where a total of T ratings have valid entries (shown as grayed cells). Given two arbitrary items i,j $\in$ I, their similarity metric is denoted as s_{ij}. Computation of s_{ij} proceeds by first identifying the set of users who have rated both i & j denoted as Uij $\subseteq U$. $\bar{r}_i$ and $\bar{r}_j$ denote the average rating of items I and j respectively. A similarity metric such as Pearson's correlation coefficient can be defined to compute s_{ij} as follows:

$$s_{ij} = \frac{\sum_{all\ u \in U_{ij}} (r_{ui} - \bar{r}_i)(r_{uj} - \bar{r}_j)}{\sqrt{\sum_{all\ u \in U_{ij}} (r_{ui} - \bar{r}_i)} \times \sqrt{\sum_{all\ u \in U_{ij}} (r_{uj} - \bar{r}_j)}}$$

Pearson coefficient $s_{ij} \in [-1,1]$. In Figure 22, users denoted as u,v are common between items i and j and form U_{ij}. Therefore the ratings in the cells denoted by (i,u) and (j,u) will form r_{ui} whereas (i,v) and (j,v) will form r_{uj}. The denominator terms of sij denote the standard deviations of r_{ui} and r_{uj} respectively whereas the numerators represent their covariance.

4.1.1.2 Estimation of Prediction

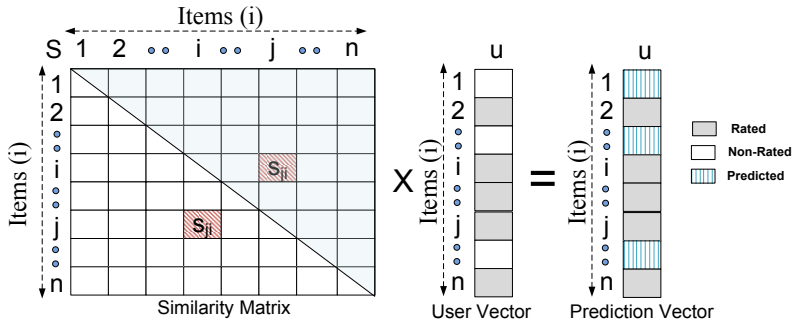

Figure 23. Estimation of Prediction for User(u)

Once the similarity for all i,j $\in IxI$ is available, the prediction for an arbitrary user u $\in$ U for an unrated item i $\in I$ can be calculated (Figure 23). This uses the similarity matrix and his prior rating vector in the following formulation:

$$p_{ui} = \bar{r}_i + \frac{\sum_{j \in I_u} (r_{uj} - \overline{r_u}) \times s_{ij}}{\sum_{j \in I_u} s_{ij}}$$

This operation is equivalent to a matrix-vector product of the previously computed similarity matrix and a users rating vector. The resultant vector of size nx1

will have $p_{ui.}$ This is shown in Figure 23 as the prediction vector with vertically hatched lines.

4.1.1.3 Presentation of Recommendation

Once a prediction vector for user u has been computed (output of Figure 23) i.e. P_u (prediction vector for user *u*), we select the top-K from among them as his recommendations. The number of possible predictions for user u will be $|I| - |I_u| = p$. This is accomplished using a priority queue of size k. This operation can be completed in O(p log k) time since we have to rebuild a priority queue p times (each rebuild of the heap is an O(log k) operation.

4.1.1.4 Key Challenge in All Pairs Similarity Computation

When $|I|=n$, the number of s_{ij} pairs is of $O(n^2)$. For a given (i,j) pair, the average number of ratings to be compared is equal to the average number of ratings per item = T/n. Therefore the overall complexity for the computation of all-pairs similarity is O(nxnxT/n) = O(nT). In case all pairs similarity between all user-pairs (s_{uv}) is to be computed, the overall complexity will be O(mT). We will consider user-user similarity to further analyze this problem and propose the intuition for our algorithm. The analysis will be analogous for item-item similarity; user-user variation is described in this thesis for ease of explanation.

Let us hypothetically assume that distributing the computation of all-pairs item-item similarity into p tasks is possible where p~n. The hypothetical complexity of this task would become O(nT/p)~O(T). The situation would appear as in the following simplified diagram (Figure 24):

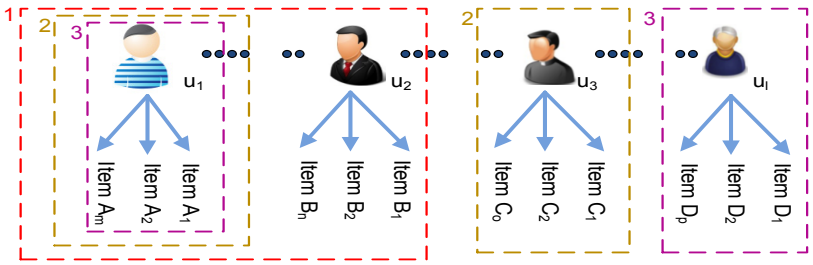

Figure 24. Brute Force User-User Similarity

Figure 24 shows users $u_1,u_2,u_3,..u_l$ who have rated m,n,o,p items each. The pairs (u_1,u_2), (u_1,u_3), (u_1,u_l) are parallelized simultaneously (shown with dotted lines). For every such pair of items, say (u_1,u_2) the common items need to be identified. This is O(mxn) or in general O(T) in general. Many-core systems such as GPU's provide a natural choice to achieve the first level of parallelization described above. However, these architectures do not provide a second level of parallelization. Techniques using Bloom Filters [9] which were used to find the common items between two sets $\{A_1,A_2,..A_m\}$and $\{B_1,B_2,\ldots B_n\}$do not apply anymore since there is no second-level of parallelism.

Contemporary works in this domain such as [29] take this approach and propose further improvements in parallelizing effectively in an architecture-aware manner: reducing memory bank conflicts in GPU's, using multi-dimensional grids but ultimately resort to a brute force method of identifying common items in the Item set. Under these circumstances no performance better than O(nT/p) is possible.

4.1.1.4 Motivation for Proposed Algorithm

The key contribution in this chapter is to recognize that it is indeed not necessary to compare the sets in a brute force manner. This only increases the number of redundant computations being performed in each parallel task/core. Our objective as per Equation (2) is to determine all items $i \in I_{uv}$. Therefore, for a given (u,v) pair, and the items rated by u and v; if we indexed all other users who have seen those I_u and I_v, we have solved our problem.

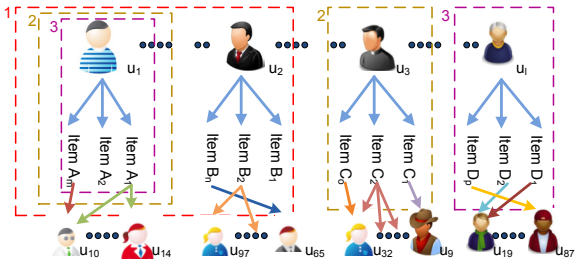

Figure 25. Converting User-User Similarity into a Counting Problem

This situation is demonstrated graphically through Figure 25. Consider a pair say (u_1, u_2) again. u_1 proceeds through all items (A_1,A_2,…,A_m) and seeks out all other users who have rated A_1, A_2, …, A_m – for instance u_{10}, u_{14}. An intermediate data structure for (u_1,u_{10}) and (u_1,u_{14}) is updated with $r_{u1,am}$, $r_{u10,am}$ & $r_{u1,a1}$,$r_{u14,a1}$. These are the only two pairs relevant rating relevant to similarity computation for u_1. In effect, this has reduced the problem of brute-force search into one of counting. The additional cost will be that of creating a suitable data structure, which can carry the necessary additional indices.

We have reduced our time requirement at the cost of increased space needed to store these index values.

The complexity of this approach is now $O(n \times \frac{T}{n} \times \frac{T}{u})$ Since $n \ll \frac{T}{u}$, it reduces to $O(n \times \frac{T}{u})$. If this were to be parallelized in p parallel paths where n~p, the resulting complexity of the algorithm shall be only O($\frac{T}{u}$). Therefore this approach can expect to have an order of magnitude speedup.

We next describe the implementation issues for this algorithm including, creation of the required data-structures, description of the algorithm and its mapping into two representative parallel architectures (Nvidia's GPU's and Intel's SCC).

4.1.2 Proposed Data Structures and Algorithms

This section outlines the algorithm and the necessary data-structures to realize the algorithmic framework described earlier. For a given user u, a list of items he/she has rated is stored in an array *itemByUser*, the ratings given is stored in a second array *ratingByUser* both indexed by the value in $userIndex_u$ and $userIndex_{u+1}$. Likewise for an arbitrary item *i*, a list of users who have rated it is stored in a separate array *userByItem*, the corresponding ratings are stored in a second array *ratingByItem* both indexed by the values in $itemIndex_i$ and $itemIndex_{i+1}$. This data structure is common for both item-item and user-user collaborative filtering. The data structures *userIndex* and *itemIndex* are maintained with the help of two HashTables. This situation is shown in Figure 26.

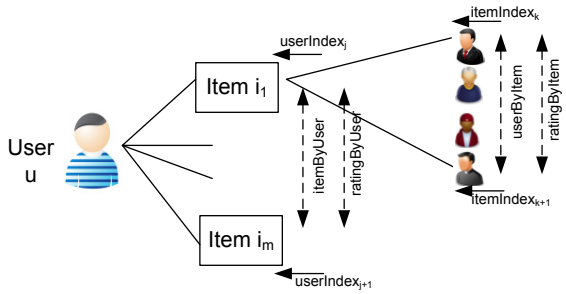

Figure 26. Designing the Required Data Structures

A single processing element (streaming multiprocessor on the GPU) or a processing core (on the SCC) operates on the i^{th} item or u^{th} user. Algorithm 2 shows the pseudo code for calculating user-user correlation.

Line 13 of Algorithm 2, indicates the initialization (to zero) of a three dimensional matrix (inter) with the dimensions Num_Users x Rating_Max x Rating_Max. Rating_Max indicates the highest range in the Rating dataset. In the datasets used in this paper, we have used Rating_Max = 5 (a range of 1-5). For instance, if user u_i detects neighbors' u_{10}, u_{23} and the ratings that they share with u_i are (5,1) and (2,3) respectively the following elements of (inter) will be incremented: [10][5][1] and [23][2][3]. The loop between lines 10-15 parses through all possible items rated by user u_i. For each such item say i_1, the inner loop 11-15 finds all possible users in the rating set that have rated it (through userByItem). For every such pair of users, an intermediate 3-D matrix is updated.

Algorithm 2: Algorithm for User-User all pairs correlation

```
1: load ratingByUser of size NUM_RECORDS
2. load itemByUser of size NUM_RECORDS
3. load ratingByItem of size NUM_RECORDS
4. load userByItem of size NUM_RECORDS
5. load userIndex of size NUM_USERS
6. load itemIndex of size NUM_ITEMS
9. copy all data structures from host to device memory
10. for i=0 to i<NUM_USERS do
11.     define inter[NUM_USERS][R_MAX][R_MAX]
10.     for j = userIndex[i] to j< userIndex[i+1] do
11.             for k = itemIndex[itemID] to k<itemIndex[itemID+1]
12.                     if (userByItem[k] >=i) then
13.
        inter[userByItem][ratingByItem[k]][ratingByUser[j]]++
                     end if
14.             k++
15.             end for
16.     j++;
17.     end for
18.     for j=i; j < NUM_USERS do
19.             for k=1 to k<=R_MAX do
20.                     for l=1 to l<R_MAX do
21.                         val = inter[j][k-1][l-1]
22.                         sum1 += l*val
23.                         sum2 += k*val
24.                         sumsq1 +=l*l*val
25.                         sumsq2 +=k*k*val
26.                         sumpr += k*l*val
27.                        num += val
28.                        l++
29.                     end for
30.             k++
31.             end for
32.             top = (sumpr – (sum1*sum2/num))
33.
        bottom = sqrt((sumsq1-sum1^2/num)* (sumsq1-sum1^2/num)
34.             pearson = top/bottom
35.     j++
36.     end for
37. i++
38. end for
endprocedure
```

/* In case of the SCC */
Follow Line 10 by : **if** (i%NUM_CORES==CURR_CORE) **then**
Until Line 36: **end if**

/*In case of the GPU */
Replace Line 10: Launch kernel_parallel_i_i_corr(..) with all loaded data structures
procedure kernel_parallel_i_i_corr(..)
1: tid = (blockIDx.x*blocDim.x)+threadID.x
2. Replace lines 11-34
end procedure

Once this parsing is complete, a second loop begins for the user u_i (lines: 18-36). This loop walks through the intermediate matrix to generate the correlation coefficient. In this pseudo code, we have used a more computationally efficient to calculate s_{uv} as shown below. In this case, n is the number of common users identified.

$$s_{uv} = \frac{\left(\sum_{all\ i \in I_{uv}} (r_{ui} \times r_{vi})\right)/n - \overline{r_u}\ - \overline{r_v}}{\sqrt{\sum_{all\ u \in Uij} (r_{ui}{}^2) - \overline{((r_u)^2)/n}} \times \sqrt{\sum_{all\ u \in Uij} (r_{uj}{}^2) - \overline{((r_j)^2)/n}}}$$

4.1.3 Results and Discussion

The goal of this section is to investigate the efficiency, execution time performance and power consumption of user-user and item-item collaborative filtering recommender when run on a GPU and SCC. The variables we can experiment with are: (1) Number of Items (or Number of Users), (2) Number of Users/Item (or Number of Items/User), (3) Number of Threads/Block (or Number of Cores used). The objective of our paper is not to measure the quality of recommendations; metrics such as accuracy,

quality, diversity etc. will not be used. The GPUs used in our experiments were NVIDA Tesla C870, Quadro 200M and Kepler GTX680.

The power monitoring on the GPU's was done using Watts' Up Pro power analyzer from Electronic Educational devices [30]. This measures bulk system power consumption (Host CPU+GPU). Measurement of Power consumed by the SCC cores is achieved via a Board Management Controller (BMC) that monitors the voltage and current drawn from the power rails feeding the SCC cores. These values are available through a series of system calls from the MCPC. We synchronize these calls with the execution of our algorithm, average the results. Execution Time on the GPU is measured using standard CUDA timers whereas wall clock is used on the CPU and SCC to measure the execution time of algorithms.

In the following section, we report & analyze overall execution time, power, energy, speedup for the recommender core on: (1) Intel's experimental single chip cloud computer (SCC), (2) NVIDIA's CUDA-enabled GPGPU acting as a co-processor and (3) traditional server class x86 (Xeon) processor. To be able to better control the behavior of our algorithm, we decided to implement a synthetic dataset where the first two parameters can be tuned. However, to verify whether our system would operate on real-world datasets, we used Flixster [31]. The Flixster dataset consists of 147,612 users who have rated a total of 48,794 unique movies (items) at least once. The total number of ratings available in the dataset is 8,196,077. Other datasets which have been used for functional correctness are: Bookcrossing [32] & Movielens [33].

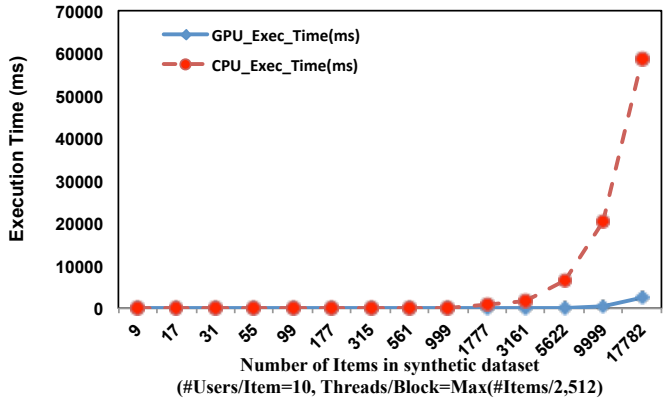

Figure 27. Execution Time for Item-Item Correlation (Tesla C870)

Figure 27 shows the overall execution time for a synthetic dataset by varying #items (n) while keeping #(users/item) constant at 10. Sequential execution performed by the CPU causes an exponential increase in the execution time as the number of items increases whereas the same operation on the GPU is faster ~30x on average.

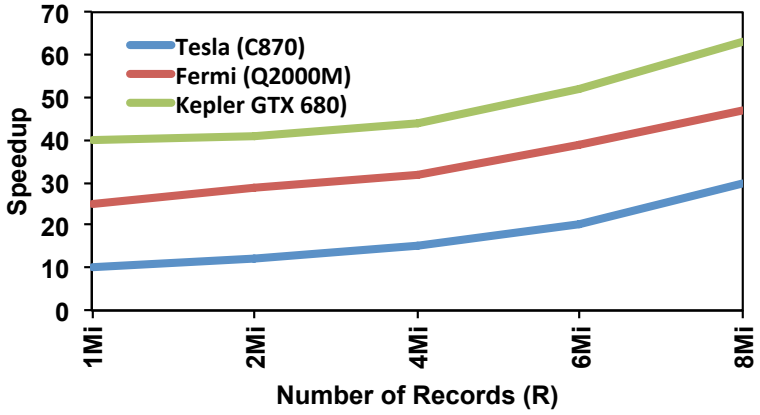

Figure 28. Speedup for Item-Item Correlation for Tesla, Kepler & Fermi GPUs

Figure 28 shows the experimentally determined speedup for item-item correlation as compared to the base-line i7 CPU core for the Tesla, Fermi & Kepler GPUs. In this case a synthetic dataset was used with varying number of items while keeping the #(users/item) constant at 10. Sequential execution by the CPU causes an exponential increase in the execution time whereas the Tesla, Fermi & Kepler perform ~30x, ~47x and ~63x better.

Figure 29 shows variation in execution time for a synthetic dataset due to increase in #Threads/Block from 2 to 512 with fixed #items = 10k & #Users/Item = 1778. This corresponds to |R|=T=485k unique records. It demonstrates that the reduction in execution time for the algorithm is insignificant beyond #Threads/Block=32.

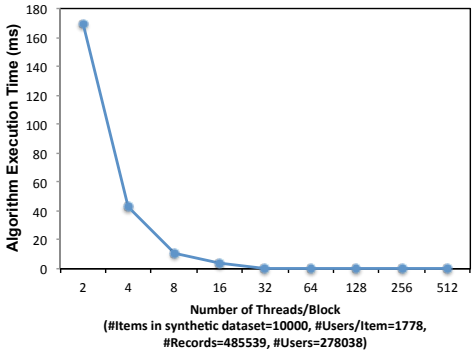

Figure 29. Variation of Execution Time with varying Threads/block

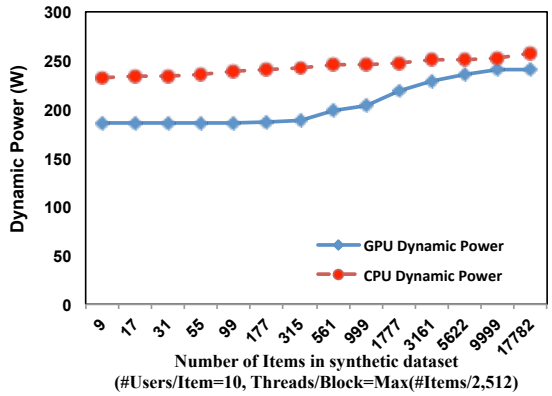

Figure 30. Dynamic Power Consumption using Tesla GPU & Intel Xeon GPU

Figure 30 shows the variation of power consumed by the CPU and GPU respectively with varying #items keeping all other parameters constant. This experiment was conducted with #users/item = 10. The dynamic power for the GPU is lower than that consumed by the CPU but approaches that of the CPU for large datasets. It is known that GPUs are energy efficient but not necessarily power-efficient.

Table 3. Energy Savings with Proposed Algorithm on a GPU

Number of Items (n)	CPU Exec. Time t_{CPU} (s)	CPU Avg. Power (W)	GPU Exec. Time t_{GPU}(s)	GPU Avg. Power (W)	Speedup $\frac{t_{CPU}}{t_{GPU}}$	Energy Saved (%)
1000	0.248	246	0.0078	204	31.79	97.39
1778	0.730	247	0.022	219	33.16	97.32
3162	1.174	250	0.0669	229	26.50	96.54
5623	6.484	251	0.1995	235	32.50	97.11
10000	20.22	253	0.6154	240	32.86	97.11
17783	58.57	258	2.3279	241	25.15	96.28

Table 3 shows the execution time, average power consumption CPU and GPU respectively when using the Flixster dataset and constant #Threads/Block = 512. The average energy savings is ~97% which is mostly due to the massive ~30x speedup in execution time (especially since the power levels of the GPU approach that of the CPU)

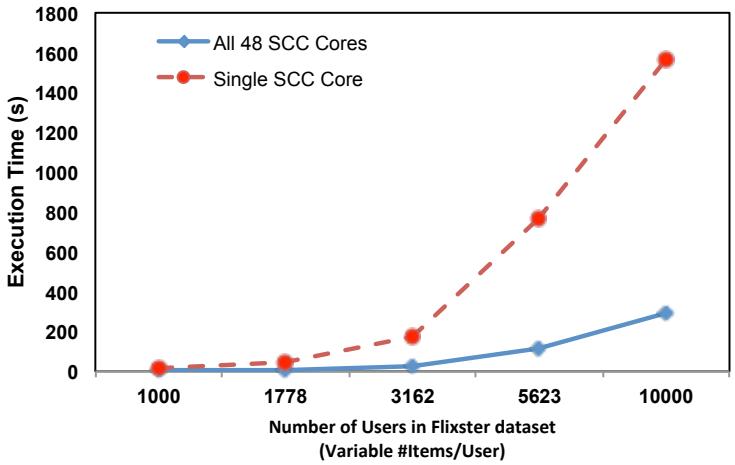

Figure 31. Execution Time for Proposed Algorithm on Intel SCC

Figure 31 shows execution time for the Flixster dataset by varying #items (n) while keeping #(users/item) constant at 10 on the SCC. Since the SCC cores are the much older 1st generation Pentium's, it is not fair to compare them with the state-of-the-art Xeon processors. We believe that the lower speedup (compared to ~30x on the GPU is due to the much lower available parallelism on the SCC).

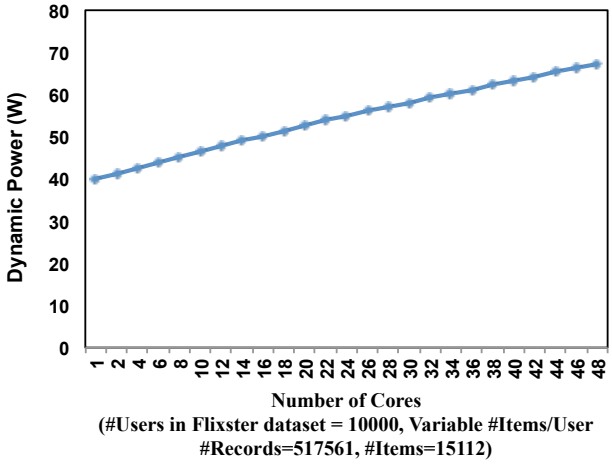

Figure 32. Dynamic Power Consumption for Proposed Algorithm on Intel SCC

Figure 32 shows the variation of power delivered by the SCC supply rails with increasing participation of cores in the computation. This power was measured via on-board ADCs on the Board Management Controller. In this case, #Cores used = 1 to 48 with fixed #users = 10k. The dynamic power consumed by the SCC is ~3-6x lower than that consumed by the GPU.

Table 4 shows the execution time, average power consumption of the SCC: single core (t_{SIN}) and all 48 cores (t_{48}) respectively when using the Flixster dataset and 1K<#Users<10K. The average energy savings is ~82.62% which is primarily due to the ~10.56x speedup in execution.

Table 4. Energy Savings with Proposed Algorithm on Intel SCC

Number of Users (u)	Single Core Exec. Time t_{SIN} (s)	Single Core Avg. Power (W)	All Cores Exec. Time t_{48}(s)	All Cores Avg. Power (W)	Speed up $\frac{t_{SIN}}{t_{48}}$	Energy Saved (%)
1000	26.238	40.04	3.470	76.176	7.561	74.83
1778	74.789	41.24	8.893	78.454	8.409	77.37
3162	334.23	43.87	22.669	74.548	14.743	88.47
5623	1390.2	48.28	113.17	79.431	12.284	86.60
10000	2874.7	52.35	291.965	73.157	9.846	85.80

4.1.3 Related Work

The closest work to this paper is the CADAL Top-N recommender [34] which has been accelerated on a GPU. They perform user-user collaborative filtering and also use the Pearson correlation coefficient. They use a more recent Tesla C2050 GPU (however, maximum number of threads/block = 1024). We compare our results with that of the CADAL Top-N recommender in Table 5.

Table 5. Comparison with Related Work

Number of Users (u)	Exec. Time t_{CADAL} (s)	Exec. Time t_{GPU} (s)	Speedup $\frac{t_{CADAL}}{t_{GPU}}$	Exec. Time t_{SCC} (s)	Spee dup $\frac{t_{CADAL}}{t_{SCC}}$
1000	2.1	0.0005	4384	3.47	0.61
2000	8.8	0.0007	12662	10.22	0.86
4000	35.5	0.0012	29194	33.84	1.05
8000	142.1	0.0021	67187	198.09	0.72

Speedup for our GPU algorithm greatly exceeds that reported by CADAL. This demonstrates the superiority of our counting based method that greatly reduces redundant computations of the same items for a given user pair. This is because CADAL

performs the naïve user-user similarity matching as described earlier in this section. Our counting based method requires marginally more memory (additional index pointers need to be transferred to the GPU) but provides an order of magnitude larger speedup. Speedup for our counting method on the SCC is almost equal to that reported by the authors of CADAL. This is despite the much fewer number of cores in use (the degree of parallelization (p) is =48 compared to =1024 in case of the GPU). As discussed earlier in this chapter, this difference is because: the order of computation in our algorithm is $O(\frac{n}{p} \times \frac{T}{u})$ compared to $O(\frac{n}{p} \times T)$ for the CADAL system.

4.2 Distributed Memory Approach

Sec. 4.1 described the design of the CF algorithm using a shared-memory model. This necessitated the loading of the entire dataset (including our custom indexes) into the main memory of each core, although each participating core was designed to operate only on a subset of the dataset. Although this enabled significant speedup, it requires the sharing of common index values in shared memory. This is not scalable for larger datasets due to the limited amount of shared memory available and the overhead in accessing them for a cluster-on-chip architecture like the Intel SCC.

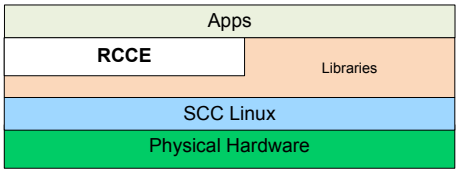

Figure 33. Conventional SCC Programming Model

Figure 33 shows the programming model that was used for Sec. 4.1. The application/algorithm was developed with the help of Intel provided software toolkit [18] operating on the physical hardware. The application was designed & and run with OS support (SCC Linux) using one-sided core-core communication library (RCCE) and other architecture specific libraries to achieve core-core synchronization.

The existing SCC programming model requires that an application developer significantly modify an existing parallel application or rebuild it from scratch using standard parallel programming constructs (like MPI, OpenMP etc.) with knowledge of the SCC architectural details. This will entail significant redevelopment costs for different applications. The development of a common base framework such as Mapreduce that abstracts away communication & synchronization primitives required for applications will allow for faster application development. Most significantly, this is likely to allow for cores to execute parts of the application independent of each other from their own allocated regions of DRAM/HDD.

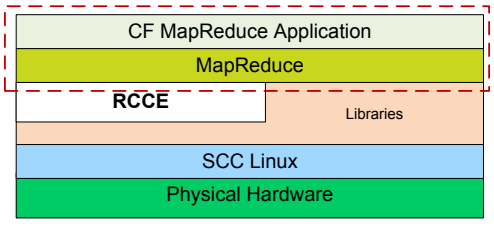

Figure 34. Proposed SCC Programming Model

4.2.1 Motivation for Proposed Approach

We reused the commonly used MapReduce programming model (on distributed clusters) within the SCC chip itself and run our application on top of it (in a distributed manner). In the cloud computing community, a large number of applications (in various domains) already have been expressed using the MapReduce [3, 35] programming model. The addition of the MapReduce run-time and application layer (CF Mapreduce Application) on top of the standard SCC programming model is shown in Figure 34.

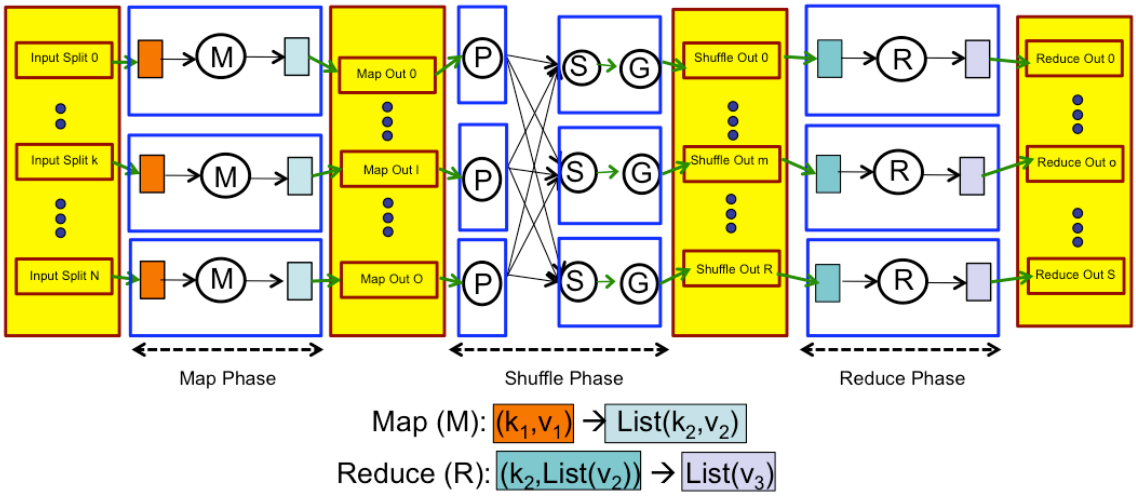

Figure 35. Mapreduce Programming Model on Many-core Processors

MapReduce [3] is a high-level parallel programming model which has become popular for data intensive computing on shared-nothing clusters. It requires a designer to specify two task primitives *map* and *reduce* which run on participating compute nodes.

Communication task primitives - *partition, sort* and *group* (the three together called *shuffle*) exchange and aggregate the intermediate output from *map* into *reduce*. Input data to be processed is split and stored block-wise across the machines participating in the cluster, often with replication for fault tolerance using a distributed file system (DFS). Mathematically the task primitives can be expressed as:

$$\text{Map: } (k_1, v_1) \rightarrow \text{list}(k_2, v_2)$$

$$\text{Reduce: } (k_2, list(v_2)) \rightarrow List(v_3)$$

The execution of a typical MapReduce program can be expressed in four phases (Figure 35). In the beginning, the *map* function runs on parallel workers (all participating machines in the cluster; called *mappers*) to produce a set of intermediate *(key,value)* pairs: $list(k_2,v_2)$. Next, a *partition* function exchanges intermediate data between the workers; a *sort* function sorts them at each worker node; a *group* function pools together all values for a key to produce ($k_2,list(v_2)$). Here, the shuffle stages (partition, sort, group) potentially require all-all communication and transform $list(k_2,v_2)$ to $(k_2,list(v_2))$. Finally, the reduce function operates on this $list(v_2)$'s for k_2 at each parallel worker (now called *reducers*) to produce the final *(key,value)* output: (k_3,v_3). This final output from *reduce* is stored back into the DFS (i.e. the HDD at each compute node). This model has become popular because it abstracts details of parallelization, fault tolerance, locality optimization and load balancing away from an application developer. An application developer only has to define the content of the *map (M), reduce (R), partition (P), sort (S)* and *group (G)* functions and leave the rest of the process to the framework.

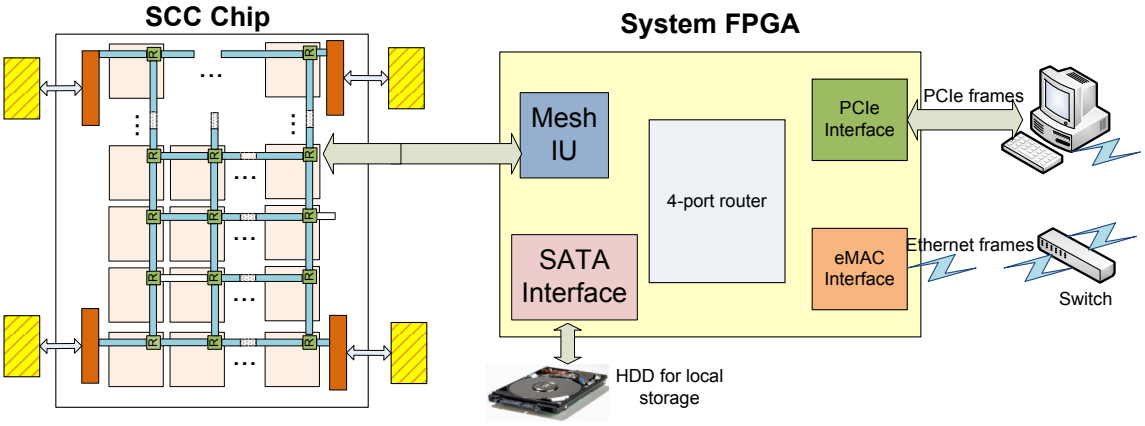

Figure 36. FPGA Interface to Intel SCC

A key requirement for deploying the MapReduce model is the availability of a distributed file system, which can store the intermediate results from each stage. Unfortunately, there is no direct HDD interface to the cores of an SCC. Although the System FPGA (on the SCC motherboard, shown in Figure 36) has the capability to directly interface SATA hard disks, this capability has not been made available (as of writing of this thesis) and when made available would still constitute a bottleneck. Therefore, the only way a HDD can be used is via a NFS mounted share on the MCPC. This System FPGA (shown in Fig. 5) acts as a software-controlled port/router into the SCC's on-die mesh routing programs & data as desired. It provides stand-alone Ethernet ports (called eMAC) and a PCIe interface to the MCPC. At this point, data I/O can be done either through the PCIe interface (called Ethernet over PCIe) or through Ethernet ports (called Ethernet over MAC) but not both simultaneously. Ethernet over PCIe

suffers from a known high-IO load bug; therefore we use Ethernet over eMAC in this paper (1Gbps).

A second problem is that currently MapReduce run-times such as Apache Hadoop would use TCP/IP communication primitives. This will be inherently inefficient because packets will need to travel off-chip to the FPGA for routing and not take advantage of the on-die routing capabilities (though the message passing buffer and on-chip routers available through the Intel provided RCCE library [18]. Thirdly JVM support for the first generation Pentium is deprecated, a plug-n-play of an open-source framework like Hadoop is out of the question and expected to be inefficient. It was hypothesized that a MapReduce model built from the ground up using architecture specific communication primitives may provide better performance.

The specific tasks that were performed are:

1. Design a MapReduce runtime to operate on a many-core processor using first-principles as described in [3] using the standard software environment provided by Intel (sccKit v1.4.1.3), a modern linux kernel (2.6.38.3) and RCCE, the Intel one-sided communication library.
2. Build an algorithmic framework (define stages) to perform scalable neighborhood-based CF top operate on the above run-time.

4.2.2 Design of Mapreduce-on-chip Framework

We implemented MapReduce-on-chip using first-principles using the standard software environment provided by Intel (sccKit v1.4.1.3), a modern linux kernel (2.6.38.3) and RCCE, the Intel one-sided communication library. The lack of a DFS

does not impede progress; all intermediate data for a MapReduce stage is stored in main memory.

We next describe a four-stage Map-Reduce pipeline specific to the Intel SCC architecture. It consists of the stages: *map, shuffle (partition, sort, group), reduce* and *merge.* For simplicity of the design, we make all participating cores in the SCC (#cores) execute all stages. Barriers are placed in between stages to ensure that execution proceeds from one stage to another only if all cores have completed the execution of a stage. Since each core will operate on a different subset of the input data, it is likely to complete execution of a stage slightly sooner than a sister core. A typical MapReduce cluster would consist of heterogeneous nodes and use one of the participating nodes as a scheduler. Since, we are operating on a homogenous system, task allocation is un-necessary and would actually be an overhead.

4.2.2.1 Map Stage

In this stage, we use a splitter function that splits the input data into as many parts as the number of participating cores (operating as mappers in this stage). Each core now executes the user-defined map function over its subset (chunk) of the input data (k_1,v_1) and produces (emits) a new key-value pair List(k_2,v_2) which is stored in a contiguous buffer per core; stored in off-chip DRAM – the fastest available memory.

4.2.2.2 Partition Stage

The intermediate (k_2,v_2) pairs from *map* need not all be processed by the same core in the reduce stage. An all-to-all exchange of the intermediate data is necessary to place them in their right cores. This is done using a user-defined partitioning function –

for ex. *hash(key)mod(#cores)*. The resulting value determines the destination core for a (k_2,v_2) pair. Transmission is accomplished through pair-wise exchanges between cores through Intel's RCCE [18] get-put communication library (which employs the MPB).

4.2.2.3 Sort Stage

Following transmission, the resulting intermediate buffer at each core may have the same *key*'s at different positions (corresponding to the other cores where they were received from). A Quick Sort is performed on this intermediate buffer in O(*nlogn*) time where *n* is the number of keys. We use the standard g*libc* provided quicksort and a user provided comparator function to perform this sorting. Although quick-sort operates in-place, we take additional care to swap only pointers to the (k_2,v_2) pair making the cost of swap independent of the size of data.

4.2.2.4 Group Stage

This stage operates on the sorted list of (k_2,v_2) pairs to produce a list of values for one key i.e. (k_2,*list*(v_2)) Since the list of (*key,value*) pairs has been sorted, this stage is accomplished by iterating through the sort buffer and storing values sequentially until a new key is encountered.

4.2.2.5 Reduce Stage

The preceding three shuffle phases have enabled the creation of (k_2,*list*(v_2)) pairs. Every such pair is now passed through a user-defined reduce (aggregation) function which produces a *List*(v_3) as the final output.

4.2.2.6 Merge Stage

Since it is often necessary to write the resulting output to disk, we first need to bring them together from all participating cores into one. This can be accomplished by an all-to-one communication of (*key,value*) pairs to one pre-defined core. A more efficient version of the same is possible by merging them binomially (intermediate cores acting as temporary aggregators) as discussed in [36].

4.2.3 Modeling the Item-Item CF Computation through Mapreduce Jobs –Approach A

We express the item-item collaborative filtering problem (initially described in Sec. 2.1.2 of this dissertation) in 4 stages (6 mapreduce jobs). A similar design for user-user CF can be accomplished by replacing users by items in the following formulation:

- (Stage 1) – Compute the average rating by all users for each item (i.e. calculate $\bar{r}_i$)
- (Stage 2) - Compute all item-pairs similarity (or correlation, producing s_{ij})
- (Stage 3) – Compute predictions for all items given a subset of users (i.e. $\forall u \in U$ produce p_{ui})
- (Stage 4) – Select the Top-k predictions for a user and present them as a recommendation

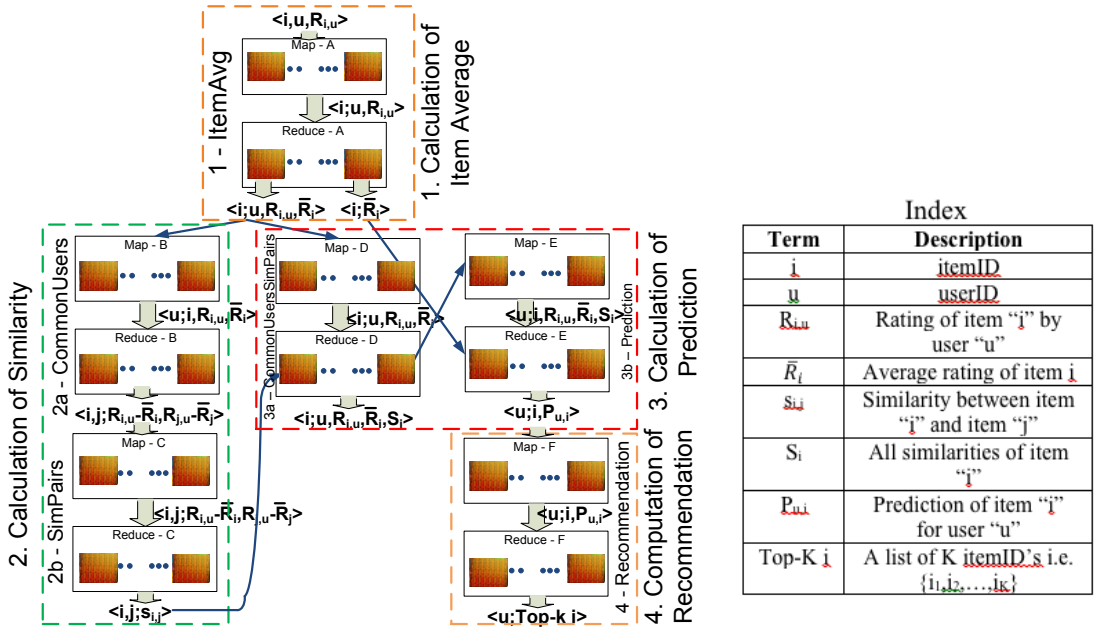

Figure 37. Computational Flow for Item-Item CF on Intel SCC – Approach A

Figure 37 shows the computational flowchart for Item-Item CF using MapReduce model suitable for the SCC (the four stages are shown with dashed blocks). Each stage may consist of more than one Map-Reduce job. The input and output of each stage consists of (*key,value*) pairs described using the notation <*key;value*>. When either the key or value in a pair consists of several distinct terms, commas separate it.

ItemAvg (Map-A & Reduce-A) calculates average rating for all items. *CommonUsers* (Map-B & Reduce-B) identifies all pairs of items that have been rated by a common user and emits the deviation of their ratings from the average. *SimPairs* (Map-C & Reduce-C) operates on this to compute the similarity between all pairs of items.

Once an all-pairs correlation has been computed, it is possible to compute a prediction for a user *u* for an item *i* that he/she has not already rated. This requires finding all items already seen by a user *u* and the similarity vector for each of those items. *CommonUsersSimPairs* captures the similarity vector for each item (*i*) and emits the user who has already seen it (*u*), his rating ($R_{i,u}$), the average rating of the item ($\bar{R}_i$) and its similarity vector (S_i). This is fed as input to *Prediction. Prediction* (Map-E & Reduce-E) groups this data by user and makes a prediction for all users as per Eq. (3). A list of userID's (*u*) and predictions for each unrated item (i.e. (i,P_{ui})) is fed into *Recommendation* where top-K highest predicted items for a given user are identified using a priority queue and presented as recommendations. We will now explain each MapReduce stage in detail in the following sections with specific reference to our target platform.

4.2.3.1 Calculation of Item Averages

The input to the systems are tuples of type *(i, u, $R_{i,u}$)* which represent only the non-null ratings corresponding to user *u* for item *i*. Map-A emits <*i;u,Riu*>. This ensures that *(userID, rating)* tuples for the same *itemID* are shuffled to the same core. In database parlance, this corresponds to "group-by itemID" operation except that it is performed in a distributed manner. Once all tuples of the type (userID, rating) for a given itemID (say equal to *numValues*) are aggregated at a core, the Reduce function (Reduce-A) performs the averaging operation:

$$\bar{R}_i = \frac{\sum_{u=1}^{numValues} R_{i,u}}{numValues}$$

This stage emits two tuples; one consisting of the <*itemID;ItemAverageRating*> and the other <*itemID;userID,rating, ItemAverageRating*>. The latter is to be used immediately while the former is to be used during calculation of predictions. The output at each core is *merged* before it is written to disk.

4.2.3.2 Calculation of All Pairs Similarity

Computation of similarity between items *i* & *j* requires the identification of the set of users who have rated both items denoted as $Uij \subseteq U$. To do so, we *"group by userID"*. This will provide us all the items seen by a user. Taking all possible permutations of such items will give us all *(i,j)* pairs which have been seen by a user. Note that the same *(i,j)* pair may occur due to multiple users; in fact that is what we need to capture. This is an indirect and elegant way to form all-possible pairs that have been rated by a common user. This is accomplished through two MapReduce jobs as discussed below.

4.2.3.2.1 CommonUsers Job

Map-B operates on <*itemID;userID,rating, ItemAverageRating*> and emits <*userID;itemID,rating, ItemAverageRating*>. This ensures that the *(itemID, rating,ItemAvgRating)* triple for every *userID* is shuffled to the same core for the Reduce-stage. For every pair of *itemID*'s say (*i,j*) thereby available, we can compute s_{ij}. Given the limited memory capabilities of the SCC cores and that we do not know how many such pairs will be produced (this is data driven), it is better to save the required intermediate result for every such *itemID* pair into disk i.e. Reduce B stores the first two bracketed terms in the numerator and denominator in the equation for s_{ij}.

4.2.3.2.2 SimPairs Job

Map-C acts as a pass through only aggregating all possible *itemID* pairs to the same core for the subsequent reduce stage. Since the key consists of multiple terms, we use a custom comparator for the *partition*, *sort* and *group* functions. Reduce-C operates on all similar (*i,j*) pairs and performs the summation in the numerator of the equation describing s_{ij}.

4.2.3.3 Calculation of Prediction

The calculation of prediction for a given user (say *u*) for an unknown item (say *i*) is performed in this stage. However, before this done, we need to associate the similarity of this unknown item (*i*) with all other unrated items i.e. the similarity vector S_i. This matching up of item's with their similarity vectors is done in *CommonUserSimPairs* job. The subsequent calculation of predictions is actually done in the *Prediction* job. It is important to note that in this stage we are not only calculating the prediction of *a* user for his non-rated items but rather the prediction for *all* users for all their non-rated items. This is expected to be the most computationally expensive operation, and is normally performed in a batch-wise manner i.e. calculate all predictions for non-rated items for a subset of users at a time.

4.2.3.3.1 CommonUserSimPairs Job

Map-D operates on the output of Reduce-A and acts as a pass-through for the data. It only ensures that all (*userID,rating, ItemAverageRating)* triples for a given *itemID* are shuffled to the same core. Reduce-D actually picks up the similarity vector

for that *itemID* and aggregates them to the (*userID,rating,ItemAverageRating*) triple; the resultant quadruple is saved to disk following a *merge.*

4.2.3.3.2 Prediction Job

Map-E ensures that the quadruple of *(itemID,rating,ItemAverageRating,ItemSimilarityVector)* for every *userID* is shuffled to the same core. This is the minimum essential data to make a prediction for a user for an unknown item. Reduce-E calculates the intermediate values & performs the summation in the equation to compute p_{ui}. The result of this stage is a key value pair of the type $<u; i,P_{u,i}>$. This is stored back into disk following a *merge*. Although it is theoretically possible, to make the recommendation computation in Reduce-E itself, we prefer to do it in an independent MapReduce job given the limited memory constraints of the SCC.

4.2.3.4 Calculation of Recommendation

In this stage, Map-F shuffles (*itemID,Prediction*) tuples for every *userID* to the same core. In Reduce-F, the tuples are inserted into a priority queue implemented through a min-heap of size *k*. Since a total of O(i) insertions are possible and rebuilding the heap takes O(log K) time, the total complexity of identifying the items with highest predictions is O(ilog k). The output of Reduce-F is a list of k *itemID*'s with the highest predictions for a given user. Each core produces such a result for every call to Reduce-F. The list of predicted *itemID*'s for a user from every core are merged to a single core and then saved to disk.

4.2.3.5 Experimental Setup and Observations

In this section, we discuss the experiments conducted to determine the validity of our approach. The goal of this section has been (1) to demonstrate that collaborative filtering can be performed with scalability on a many-core processor using the MapReduce paradigm, (2) speedup, (3) energy saving with respect to the state-of-the-art. The goal of this dissertation is not to build a better recommendation system, therefore quality metrics such as precision & recall are not considered. We can experiment with: (1) size of the dataset (#records), (2) number of parallel computational elements.

Our experiments were performed on an Intel provided SCC and software environment: SCCKit 1.4.1.3, *icc* (version 8.1), tiles at 533 MHz, mesh interconnect at 800 MHz, DRAM at 800 MHz. Our SCC had 32GB RAM for all 48 cores i.e. 640 MB per core. Execution time was measured using system calls and power was measured via an independent Board Management Controller (BMC) on the SCC motherboard. To compare this system, with a contemporary cluster, we used a 4-node cluster consisting of dual-core AMD Athlon-64 2GHz processors with 2GB DDR2 RAM. This cluster system ran Apache Hadoop 1.0.1.

To verify that our approach works on real-world datasets, we used a benchmark dataset from the GroupLens research project called Movielens [33] consisting of 100k & 1 Mi ratings. This dataset is typically used in evaluating CF-based recommender systems.

4.2.3.6 Execution Time and Energy Analysis

We measure the execution time of each MapReduce stage separately. Since the partition and merge sub-stages involve inter-core communication, we report them together as communication time. Likewise, the map, reduce and sort sub-stages in each MapReduce job are the only ones which involve computation and are reported together.

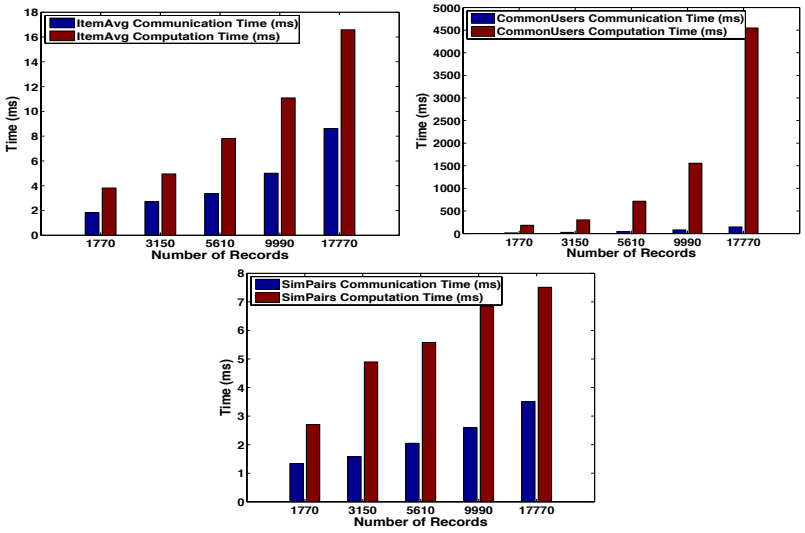

Figure 38. Execution Time for Calculation of All-pairs Similarity

Figure 38 shows the split up of communication and computation times for the first three stages of the MapReduce chain on the SCC with a synthetic dataset with increasing number of items. Here #users/item = 10, 20% of possible pairs have 10% common users.

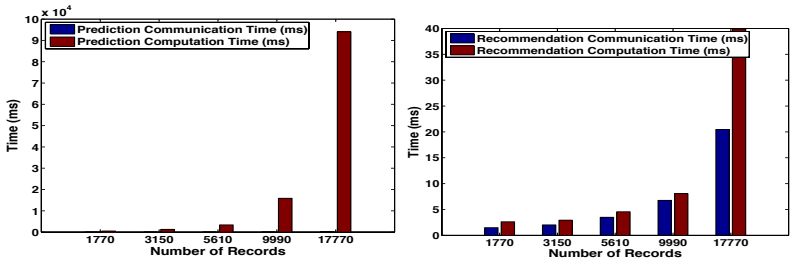

Figure 39. Execution Time for Calculation of Prediction & Recommendation

Figure 39 shows the split up of communication and execution time for the last 2 stages of the MapReduce pipeline on the SCC with a synthetic dataset with increasing number of items. Here, #users/item = 10, 20% of possible pairs have 10% common users.

We observe that the *CommonUsers* & *Prediction* stages are computation dominated whereas *ItemAvg, SimPairs,* & *Recommendation* have an even split. It is possible for us to make this estimation because for this experiment we have used a synthetic dataset whose characteristics have been kept constant with increasing number of records. The communication v/s computation split in percent for the stages (a-e) are (30%-70%), (10%-90%), (30%-70%), (1%-99%) & (40%-60%) respectively on average. We have not been able to run larger datasets due to limited availability of memory at the SCC cores. This is because of the ~640MB per core, nearly ~320 MB is occupied by the OS layer. We believe that to alleviate this by: (1) running the MapReduce model on the SCC in a bare-metal mode[18] i.e. without OS support, (2) chaining the reduce output of one stage directly to the map input of the next without File IO.

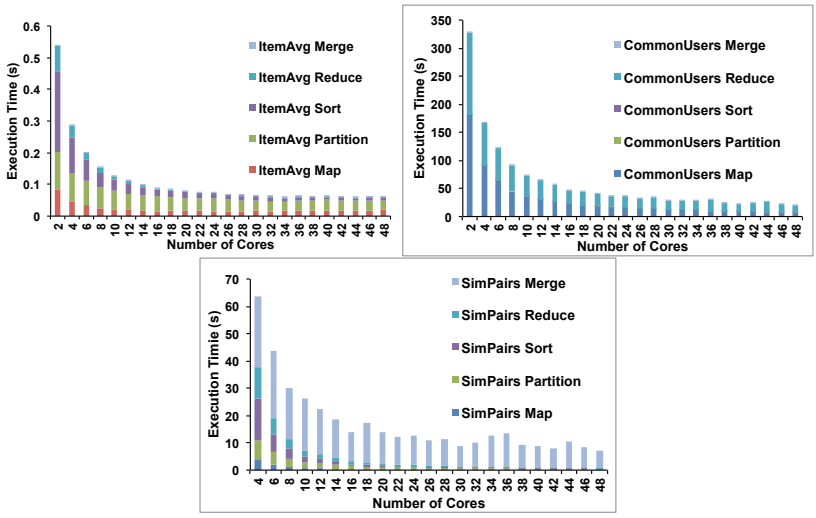

Figure 40. Stage-wise Execution time for Proposed Algorithm

Figure 40 shows the stage-wise timing (*map, partition, sort, group & reduce)* for *ItemAvg, CommonUsers & SimPairs* resp. using the Movielens-100k dataset (#distinct users=1000, #distinct items=1700, #userID's/item~60). These stages are sufficient to compute all-pairs similarity (item-item) for the Movielens dataset. Timing data is captured per core (of the 48 cores used) and the average value reported. We observed that the timing for the all cores was similar. These results do not include data IO & setup time. These results show that more cores are not necessarily better. For instance, *ItemAvg, CommonUsers & SimPairs* do not show appreciable speedup beyond 20, 24 and 16 cores respectively. This is expected and will be a characteristic of the dataset (i.e. sparsity (x,y) where x% of possible item pairs have y% of the users in common).

Further, Figure 40 also provides additional insight into the behavior of the CF algorithm on the SCC using Mapreduce and confirms our hypothesis regarding the

sparsity of datasets (real and synthetic). For instance, in *ItemAvg*, the partition stage involving all-all communication between participating cores continues to remain a bottleneck when all 48 cores are used and the same is reflected in Figure 39 on the synthetic dataset with increasing records. On the other hand in *SimPairs*, the merge stage continues to dominate which contributes to the ~30% contribution of communication time for the synthetic dataset. Also, *CommonUsers* is almost entirely dominated by the *map* & *reduce* stages (Figure 40) which reflects in the ~90% execution time consumed for this stage (Figure 39) for the synthetic dataset.

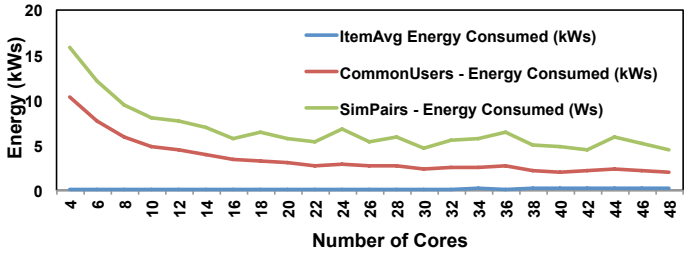

Figure 41. Energy Consumed by Proposed Algorithm on Intel SCC

We have observed in the previous results that the power drawn by the SCC is directly proportional to the number of cores in use. However execution time improves at the same time. We experiment the first three stages (A-C) using the Movielens dataset consisting of 100k and 1 Million records (also recording the overhead of job setup, IO). It is interesting to see the impact of data IO on the overall execution time. For instance, in Figure 40, *ItemAvg* when operating with 4-cores requires ~0.3s for intra-core

communication and computation. However, when including IO to HDD, this takes up to 2.4s (~87.5% due to IO alone).

We also analyze the overall energy profile for the All-Pairs similarity computation (stages a-c) with varying number of cores (using Movielens-100k) to understand the power-performance tradeoffs (Figure 41). While overall power drawn increases from ~40W to ~60W in all cases, the consequent reduction in execution time (~1.2x, ~16x & ~6x) supersedes its effect providing an overall energy reduction.
We record the execution time and average power drawn in Table 6 & Table 7 respectively for the Movielens-100k and Movielens-1M datasets. The average energy consumed is calculated in Table 8 (in units of Ws). The energy saved in using an SCC as the compute node when executing the first 3 stages of CF on the SCC v/s conventional multiprocessor systems running MapReduce is ~94.3%. Since we include the data transfer times to disk after merging, the average speedup at the same time is a modest ~2x over a conventional cluster.

Table 6. Overall Execution Time for Proposed Approach on Intel SCC

Dataset[a]	ItemAvg		CommonUsers		SimPairs	
	SCC	***Hadoop***	***SCC***	***Hadoop***	***SCC***	***Hadoop***
Movielens -100k	2.4s	36s	137.8s	302s	210s	462s
Movielens -1M	12s	42s	394.4s	591s	487s	789s

Table 7. Averaged Power Consumption for Proposed Approach on Intel SCC

Dataset[a]	ItemAvg		CommonUsers		SimPairs	
	SCC	***Hadoop***	***SCC***	***Hadoop***	***SCC***	***Hadoop***
Movielens -100k	46W	250W	46W	251W	46W	287W
Movielens -1M	47W	252W	48W	249W	48W	285W

Table 8. Averaged Energy Consumption (in J) for Proposed Approach on SCC

Dataset[a]	ItemAvg		CommonUsers		SimPairs	
	SCC	***Hadoop***	***SCC***	***Hadoop***	***SCC***	***Hadoop***
Movielens -100k	110	9k	6.3k	75.8k	9.6k	132.5k
Movielens -1M	578	10k	18.9k	147k	23k	224k

4.2.4 Modeling the Item-Item CF Computation through Mapreduce Jobs – Approach B

The observations of the previous section indicated only a modest speedup of ~2x. This section outlines an alternate formulation of the Item-Item CF problem through Mapreduce jobs. The key idea for this approach is that it is not necessary to compute a similarity matrix as per the equation above. A recommendation can be obtained through the use of a co-occurrence matrix. Further the number of Mapreduce jobs and their computational complexity can be potentially reduced.

Therefore, we can restate the 5 stages for the item-item collaborative filtering problem can be restated as: (1) calculating an item vector (group-by *itemID*) (2) calculating a user vector (group by *userID*) (3) calculate an item-item co-occurrence vector producing $count_{ij}$ (where *i,j* are item pairs which have at least one user in common), (4) computing predictions for all items for a user (i.e. $\forall\ u \in U$ produce p_{ui}) and finally (5) calculating predictions and selecting the top-k predictions for a user as recommendations.

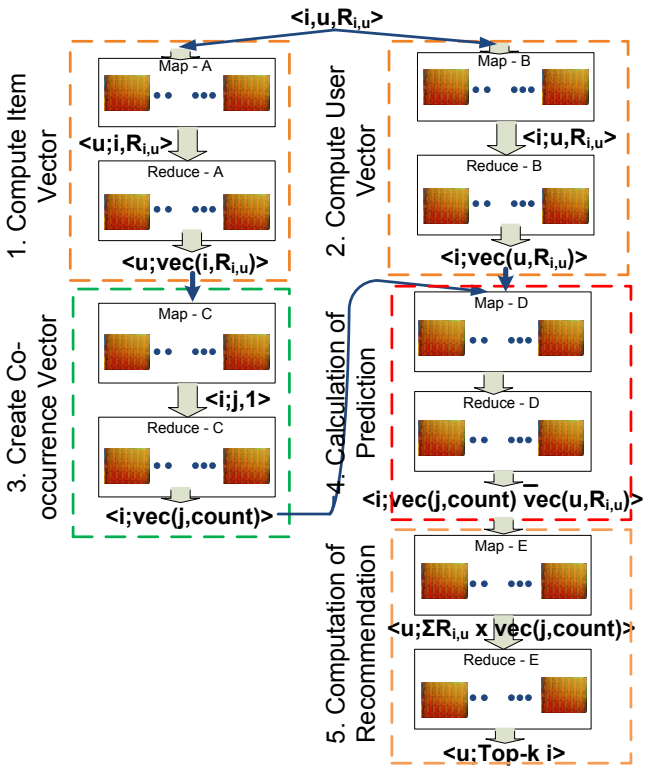

Figure 42. Computational Flow for Item-Item CF on Intel SCC – Approach B

Figure 42 shows the computational flowchart for Item-Item CF for these 5 stages. The input and output of each stage consists of *<key;value>* pairs. *ItemVector* is computed in a distributed manner on the SCC cores by Map-A and Reduce-A. *UserVector* consists of Map-B and Reduce-B and creates a vector of users and ratings (u,r_{iu}) seen by a item *I*. Map-C and Reduce-C creates an item co-occurrence vector for every pair of items (say *i* & *j*) that have been rated by a common user *u*. Map-D and Reduce-D associates the co-occurrence vector of an item with its user vector. Map-E and Reduce-E makes the predictions and recommendations for a user *u* for an item *I*. In

Reduce-E, the *top-K* highest predicted items for a given user are identified using a priority queue and presented as recommendations. Although, this explanation has been made specific to Item-Item CF, an analogous design can be made using similar MapReduce constructs for User-User CF.

4.2.4.1 Calculation of Item Vector (group by *userID*)

Since the input data is typically sparse, we provide as an input tuples of type *(i, u, $R_{i,u}$)* which represent only the non-null ratings corresponding to user *u* for item *i*. Map-A emits <*u;i,*R_{iu}>. This ensures that *(itemID, rating)* tuples for the same *itemID* are shuffled to the same core.

4.2.4.2 Calculation of User Vector (group by *itemID*)

Map-B emits <*i;u,*R_{iu}>. This ensures that *(userID, rating)* tuples for the same *userID* are shuffled to the same core. While the earlier stage corresponded to a "group-by userID", this stage is equivalent to a "group-by itemID" operation.

4.2.4.3 Creation of Co-occurrence Vector

Computation of all-pairs similarity requires us to first identify the set of users who have rated both items *i* & *j* denoted as $U_{ij} \subseteq U$ (item-item co-occurrence vector). Map-C emits all possible permutations of all item pairs *(i,j)* which have been seen by a user. Reduce-C aggregates these pairs and produces a co-occurrence vector for every item.

4.2.4.4 Calculation of Predictions

The calculation of prediction is to be done for a given user (say *u*) for an unknown item (say *i*). For this to be done, we need to first associate every item with two

vectors: (1) the similarity (or co-occurrence) vector of this unknown item with all other items & (2) the user vector for an item. This data aggregation is done through Map-D and Reduce-D.

4.2.4.5 Calculating Recommendations

In this stage, Map-E shuffles (*itemID,Prediction*) tuples for every *userID* to the same core. In Reduce-E, the tuples are inserted into a priority queue implemented as a min-heap of size *k*. Since a total of O(*p*) insertions are possible and rebuilding the heap takes O(*log K*) time, the total complexity of identifying the items with highest predictions is O(*p log k*). The output of Reduce-E is a list of k *itemID*'s with the highest predictions for a given user.

4.2.4.6 Execution Time and Energy Analysis

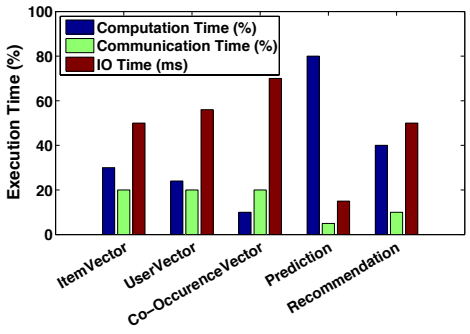

Figure 43. Analyzing Computation, Communication & IO time (Approach B)

Figure 43 shows the split up of communication, computation and IO times for each stage of the MapReduce chain on the SCC for the Movielens-100k dataset. Since partition and merge sub-stages involve inter-core communication, they are reported

together as communication time. IO time includes the time required for load of data from the external HDD to DRAM, latency in access from DRAM to the cores, and setup time. Computation time includes only those operations where data is handled by the cores. We observe that *ItemVector* and *UserVector* stages have similar characteristics (they perform analogous operations). In contrast, *Co-OccurenceVector* is communication dominated and *Prediction/Recommendation* is computation dominated. This is in tune with the expected behavior since the first two stages perform simple comparisons for the group-by operations, whereas the Prediction/Recommendation perform computation intensive operations on a large segment of data. The only stage where communication dominates computation is the *Co-OccurenceVector* stage. This is also expected since a large number of (i,j) tuples are exchanged between cores in the *shuffle* following the *map* operation. All stages are IO dominated (>50%). This demonstrates for further speedup using this computational model on the SCC, IO must be minimized significantly.

4.2.5 Comparison with Related Work

Prior work to this paper can be broadly split in two areas: one involving development of the MapReduce programming model on multi-core and many-core systems [37] and the second involving formulation of the CF problem in MapReduce on traditional clusters [38, 39]. To the best of our knowledge, this is the first work of its kind involving Collaborative Filtering on Many-Core systems using the MapReduce paradigm. In [37], a single-stage MapReduce implementation on the SCC was shown. Although this has been a key stepping stone for our research, its key differences with this paper are: (1) Applications limited to a few map-, partition- and sort- dominated

benchmarks, (2) no support for stage chaining. Also, we examine the design of a multi-stage CF algorithm on top of this model. We differ from [38] in the following ways: (1) we use two MapReduce stages in calculation of similarity whereas they can achieve scalability with one. (2) we use two MapReduce stages in calculating prediction whereas they use one and (3) we model the final Recommendation calculation as a MapReduce step. A single stage calculation of similarity & prediction as envisaged in [38, 39] works well when using conventional compute nodes, which do not have limited main memory and have local storage (hard-disk). Since, we use main memory to store and operate upon intermediate data, we have had to redesign the computation in multiple phases to limit its requirement.

5. RECONFIGURABLE SOC FOR DATA INTENSIVE COMPUTING*

5.1 Motivation

The results from the previous section have shown that GPUs outperform the Intel SCC for both semantic information filtering (CB) and collaborative information filtering (CF) algorithms despite their power inefficiency. The research question that remains to be addressed in this dissertation is whether we can achieve GPU-type performance with an SCC-like power budget for such data-intensive information filtering applications.

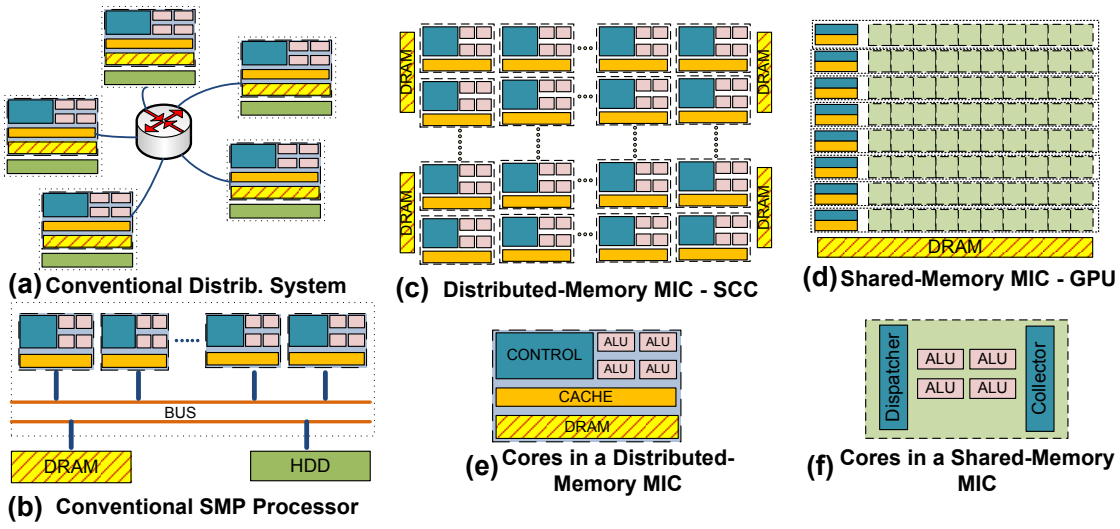

Figure 44. Many-core Architectures for Data-intensive Computing

---

* Parts of this section have been reprinted with permission from A. Tripathy, K.C. Ieong, A. Patra, R. Mahapatra, "A Reconfigurable Computing Architecture for Semantic Information Filtering", in Proc. IEEE Intl. Conf. on Big Data (BigData 2013), Oct. 6-9, 2013, Santa Clara, CA, USA. © IEEE 2013

Figure 44 shows the evolution of many-core architectures. On the one hand, a large number of legacy uniprocessors (Figure 44(e) such as x86) have been put together on the same die to form a cluster-on-chip (Figure 44(c)) such as the Intel's Single Chip Cloud Computer [18] (SCC, commercialized as Intel Xeon Phi). These machines share off-chip memory access between a subset of cores and expect applications to synchronize via message passing (distributed memory MIC). On the other hand, Graphics Processing Units (GPU's) have evolved for general-purpose usage as application-coprocessors (GP-GPU's). GPU's (Figure 44(d)) consist of large number of lightweight streaming processors and employ massive thread-level parallelism (TLP) to mask memory latency (shared-memory MIC). Cores in a GPU are actually scaled-up versions of what CPU manufacturers would have called an ALU (Figure 44(f)). This allows GPU's to integrate a much larger number of *cores* on die compared to that of an SCC (1536 in Kepler [17] v/s 48 on the SCC) leading to a performance difference.

Semantic information filtering (SIF) as a big-data application has been explored on a GPU and SCC in this dissertation in Sec. 3. Despite providing substantial speedups on both, they still remain memory-bound. Poor spatial and temporal locality of memory accesses leads to suboptimal performance levels (execution time dominated by memory latency) on cache-based (such as SCC) and GPU multiprocessors alike. Can this be improved? Second, is it possible to extract higher performance from each compute core by custom-designing its functionality in an application-aware manner. Third, is it then possible to build a reconfigurable many-core computing machine that is a hybrid both shared & distributed memory MIC's? A provision for reconfigurability/programmability

will ensure wider applicability to similar problems. Will such an architecture be scalable, provide high-throughput and enable higher performance gains at a low energy budget for big-data applications?

5.2 Contributions

In this section, we present a novel reconfigurable hardware methodology which addresses some of these challenges while considering semantic information filtering (SIF) as a case study. Our prototype System on Chip (SoC) reconfigurable processor core for SIF was designed from the ground up, evaluated on an industry-standard virtual prototyping platform for performance. We make four key contributions:

1. A reconfigurable hardware architecture which decouples computation and communication thereby issuing multiple outstanding memory requests.
2. A Bloom Filter based randomized algorithm for
3. An in-depth performance evaluation with different sizes and characteristics of benchmark datasets, number of reconfigurable processing units, memory banks etc.
4. A performance comparison of the proposed reconfigurable architecture to prior work in semantic information filtering using state-of-the-art CPU's, GPU's and SCC showing that our reconfigurable architecture outperforms HPC multi-core systems but also achive better performance scaling with respect to data size.

5.3 Proposed Algorithm

As discussed in Sec. 2 and Sec. 3, semantic information filtering (SIF) proceeds with the accurate computation of similarity between every pair of user-item profiles

(represented in their tensor forms as T_1 and T_2). Computation of semantic similarity is done as a cosine product (s_{12}). It is done with (1) identification of the common *terms* (say *k*) in the two tensors of size *p,q* respectively, (2) multiplication of the corresponding *coefficients* of the respective common terms to yield *k interim products* and (3) summation of these interim products to yield s_{12}. The similarity metric will lie in the range *[0,k]* provided the coefficients are already normalized in the source tensors T_1 and T_2. In a real-world recommender system, computation of similarity between item-user profiles is done autonomously and continuously as batch jobs. Given the temporal nature of the underlying source – high performance and energy efficiency in computation of semantic similarity can result in large economic benefit and better user experience.

If a sequential processor is used to compute semantic similarity on two tensors of size *p,q* respectively the identification of common terms has a time complexity of *O(p logq*) or *O(pq)* depending on whether or not a binary or linear search tree is used (Algorithm 3). Such a balanced BS tree is implemented in C++ STL's highly optimized map container which implements Red-Black tree. However, for ease of massive parallelization on MICs and the potential of achieving a time complexity theoretically of *O(1)* with massive parallelization, we use a randomized algorithm using Bloom Filters (Algorithm 4).

Algorithm 3: Efficient Red-Black Tree Computation of SIF on SMP

```
        Inputs: Tensor1(t1i,c1i), Tensor2(t2i,c2i)
        Output: Semantic (dot) product s12
1       foreach (t1i,c1i) ∈ Tensor1 do
2               rbtree.insert(t1i,c1i)
3       end for
4       foreach (t2j,c2j) ∈ Tensor2 do
5               rbtree_ptr ← rbtree.find(t2j)
6               if rbtree_ptr != NULL then
7                       s12 += (c2jx rbtree_ptr.value)
8               end if
9       end for
10      return s12
```

With the above analysis in mind, we can repurpose Algorithm 1 for efficient operation on massively parallel processors illustrated as Algorithm 2. This approach uses a common shared BF bit vector of size *m* to store a space efficient *signature* of the contents of $Tensor_1$ (lines 1-3, BF set operation on $Tensor_1$'s terms $\mathbf{t}_{1i}$). Since these are independent operations and can cause no race condition (BF is never set to 0), this can be delegated to every participating thread or core. Setting the BF would require fast hash functions to compute BF indices. An analogous test operation can then be performed on $Tensor_2$'s terms $\mathbf{t}_{2j}$ using the same hash bank (lines 6-7). This demonstrates our requirement for reconfigurability of the cores because the same IP block (*BF Index generation*) can be reused in both set and test phases. If all the *k* BF indices (BFI_k) return true, $\mathbf{t}_{2j}$ is a candidate match (there is a small probability of false positive defined earlier). Now we locate the corresponding coefficient of $\mathbf{t}_{2j}$ in $Tensor_1$ i.e $\mathbf{c}_{1m}$, if it exists (lines 8-11); multiply and sum this intermediate result.

Algorithm 4: Massively Parallel BF based SIF

Inputs: $\text{Tensor}_1(\mathbf{t}_{1i},\mathbf{c}_{1i})$, $\text{Tensor}_2(\mathbf{t}_{2i},\mathbf{c}_{2i})$
Output: Semantic (dot) product s_{12}
1 **parallel foreach** $\mathbf{t}_{1i} \in \text{Tensor}_1$ **do**
2 compute ∀k, $BFI_k = hash_k(\boldsymbol{t}_{1i})$
3 ∀k $BF[BFI_k] = 1$
4 **end for**
5 **parallel foreach** $\mathbf{t}_{2j} \in \text{Tensor}_2$ **do**
6 compute ∀k, $BFI_k = hash_k(\boldsymbol{t}_{2j})$
7 **if** ∀k $BF[BFI_k] = 1$ **then**
8 **parallel foreach** $\mathbf{t}_{1m} \in \text{Tensor}_1$ **do**
9 **if** $\mathbf{t}_{1m} == \mathbf{t}_{2j}$ **then**
10 s_{12} += $(\mathbf{c}_{2j} \mathbf{x} \mathbf{c}_{1m})$
11 **end if**
12 **end for**
13 **end if**
14 **end for**
15 return $\sum s_{12}$

Locating this corresponding coefficient can be carried out using an off-chip content addressable memory [40] (CAM) lookup mechanism with $\mathbf{t}_{2j}$ as the key; a single cycle operation. This eliminates the need for the loop between lines 8-12. However since the CAM lookup operation is going to be sporadic and involve significantly longer latencies, this stage has been pipelined. Further, lines 1 and 5 can be allocated to independent functional units (coarse-grained parallelism), and lines (2,3) and (6,7) can be internally parallelized (fine-grained parallelism). Each of these functional units would then retain only a partial sum. This sum can be obtained using parallel reduction or centrally on a host processor/controller. For simplicity of design, we chose to implement the parallel sum using the latter choice in this paper. With these design principles in

mind, we will now describe the reconfigurable computing template that we use to realise this computational flow.

5.4 Template for Reconfigurable Computing

The computational and memory access requirements for large-scale data intensive problems are significantly different from mainstream parallel applications, requiring new architectural solutions for efficient parallel processing. Such data-intensive problems are generally characterized by short parallel paths/threads with a small memory footprint, irregular, unpredictable and large memory access requirements.

The need for a reconfigurable processor occurs because the same processing units can be reused to execute a different phase of the computation, while sharing the same interconnect network and conserving die area.

5.4.1 Reconfigurable Processing Elements (RPE)

Designing application-specific reconfigurable processing elements (RPE's) will result in efficient utilization of hardware resources in contrasts to a more general-purpose processing element (as in a GPU or SCC). We can use high-level syndissertation tools to generate efficient implemntation of an individual RPE and provide spatial parallelism by replication.

5.4.2 Combining Coarse and Fine-grained Parallelism

Instantiating a large number of RPE's in hardware operating in a massively multi-threaded fashion will provide high coarse-grained parallelism. Additional parallelism required by an application can be provided by specialized functional units (SFU's) in a fine-grained manner.

5.4.3 Multiple Concurrent Memory Requests

Data-intensive applications in general and SIF in particular does not reuse the same data – cache memories to hide memory latency are useless in this regard. It is advantageous to have a system with single memory hierarchy and make the RPE capable of issuing multiple outstanding memory requests to off-chip memory. Given a large number of parallel RPEs proposed, parallel memory banks will lead to superior memory access performance. Having said that, this paper currently uses state-of-the-art AMBA compliant crossbar interconnect (such as NIC-301), we recognize that an AMBA compliant packet-based interconnect will provide additional scalability and improve performance of the memory subsystem.

5.4.4 Trading Clock Speed for Area

We recognize that the execution times for data-intensive applications and SIF in particular will be dominated mostly by memory latency; the RPE's will be designed to stall when it waits for requests to return from main memory. Having a RPE operate at 30 MHz or 300 MHz will make limited difference because we expect the RPE's to stall for a majority of the time. Overall application speedup will be obtained mostly from higher parallelism (i.e. large number of RPE's) than clock speeds. Therefore, in constraining clock speeds, we can enable the synthesis tool to optimize for area, leading to higher parallelism.

5.4.5 *Decoupling Computation and Communication*

We can create separate IP blocks for the RPE's core functional units (application-dependent logic) and the application independent AMBA 4 Advanced eXtensible Interface (AXI) v2.0 master and slave interfaces. This facilitates reusability of the architectural template for different applications. This also enables the use of any AMBA compliant interconnect network (packet based or crossbar based from the ARM IP library).

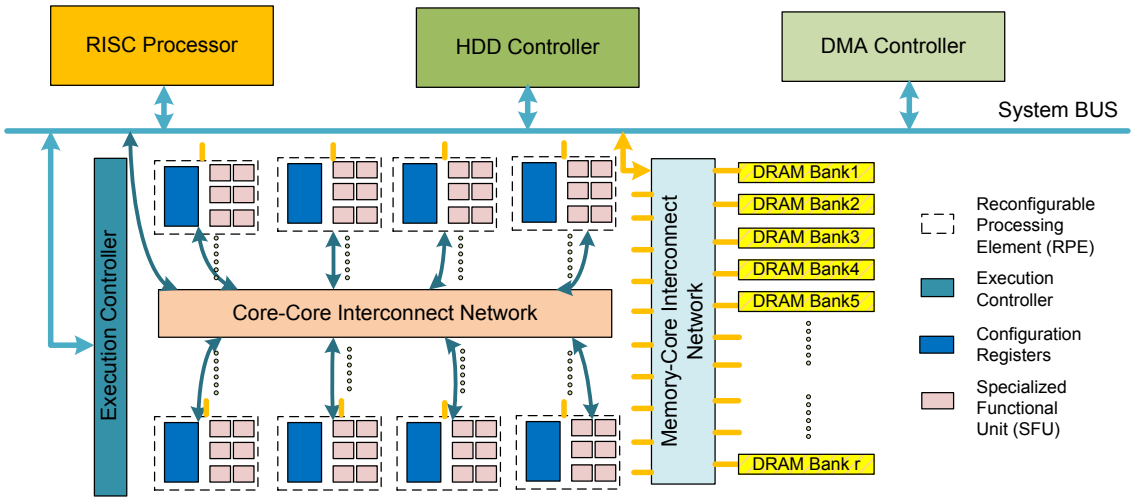

Figure 45. Reconfigurable Architecture Template for Data-intensive Applications

The overall architecture of the reconfigurable computing solution is illustrated at a high level in Figure 45. It comprises of an *execution controller* (EC), multiple *reconfigurable processing elements* (RPEs), two distinct interconnection networks: core-

core and memory-core. The RPE's (shown with dotted lines) contain application-specific logic (which can be internally fine-grained), are replicated and can independently execute application logic.

The RPE's are to be configured based on context word(s) delivered to it (shown in blue as configuration registers) in a similar manner to coarse-grained reconfigurable arrays (CGRAs) [41]. However, there are two significant differences in the design of the proposed RPEs from a conventional CGRA: (1) the base processing elements (PE's) in a CGRA contain ALU, multipliers, shift logic and registers whereas the RPE's in the proposed architecture will contain application specific logic (described subsequently in this Section); (2) the interconnection structure of a PE array in a CGRA is pre-defined and fixed for ease of modeling generic applications whereas the proposed RPE's can use common cross-bar or packet-based (NoC) interconnects and will be memory-mapped.

Two distinct interconnection networks are specified because the memory-core load and core-core load are heavily application dependent – separate choices may be made for them based on throughput/latency constraints (crossbar or packet-based). The memory-core interconnect network links the RPE's to off-chip memory banks. These memory banks may be filled in using the DMA controller independent of processor interaction. The RISC processor and DMA controller are bus masters whereas the reconfigurable co-processor array (RCA) may be considered as a slave device (Execution Controller). The EC manages the operations of the RPE's, including orchestrating initialization, task assignment, synchronization; it also provides an interface to the host CPU processor.

Each RPE is also provided with a limited number of registers to act as private local memory. A group of RPE's may also have shared memory accessible to more than one RPE. Each RPE is capable of independently reading and writing from/to its memory bank based on the configuration data sent to it.

The following section describes how we parallelized the SIF algorithm using the above reconfigurable architecture template with specific attention to the design of the RPE's themselves, the bus interfaces, the interconnect and the co-design of the hardware-software interface to run the application at the processor end. ARM processor architecture and the AMBA bus protocol only as a case-study because of it is an open standard, and ease of integration with other AMBA-compatible IP blocks in our validation tool chain.

5.5 Detailed Architectural Description

Figure 46 shows a high-level overview of the proposed SoC architecture that will be described in detail in this section. In particular, we have used ARM Cortex A9 as the low-power RISC processor. When designed with TSMC's 65nm generic process, it can be clocked at 1 GHz and consume <250mW. The rest of the figure describes the RPE matrix (RPE0-RPE128). Each RPE is provided configuration instructions via an execution controller. The RPE's have been designed to use the AMBA APB [42] bus to receive this configuration information (limited I/O required, few signals necessary). The execution controller has an AMBA AXI master interface which are translated into the APB domain using an AXI to APB bridge [43]. Each RPE consists of two AXI Master ports, which are connected via two independent interconnects (Memory-Core & CAM-

Core) to the off-chip DRAM and CAM banks respectively. A separate 154-bit bus from each RPE feeds into a separate RPE-Sync block on the SoC. Each of the components and rationale for designing them is explained in the subsections below.

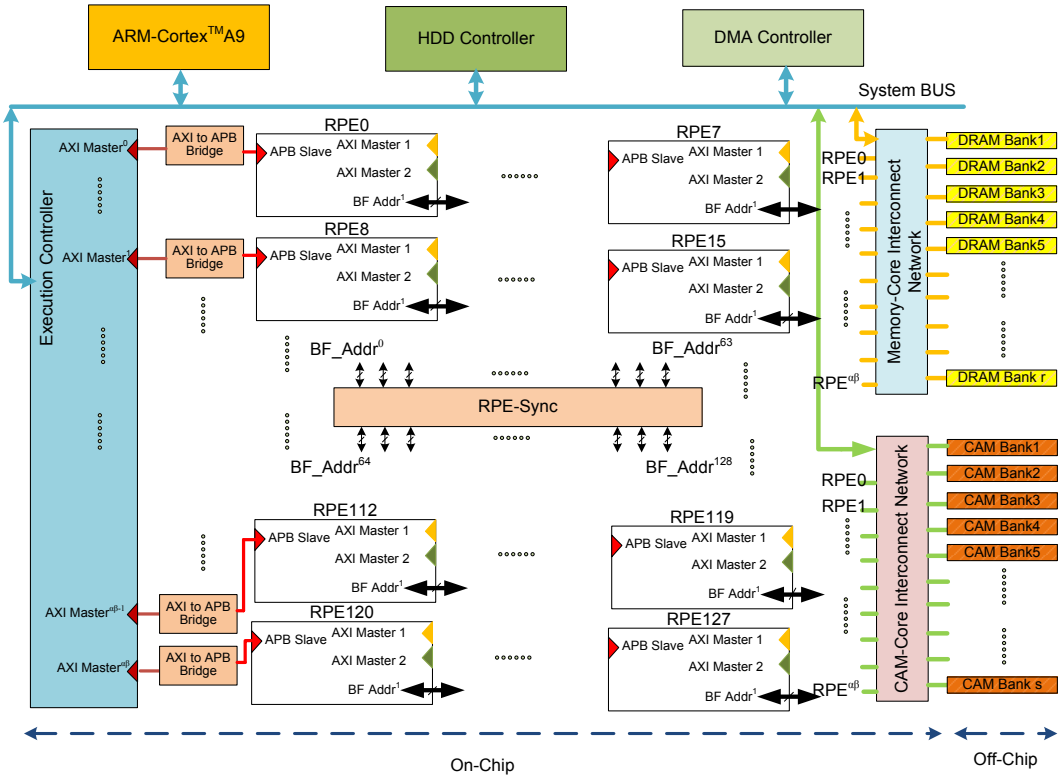

Figure 46. Proposed Reconfigurable SoC for SIF

5.5.1 Role of Host Processor (ARM Cortex A9)

The host processor orchestrates the entire operation of the SoC, partitions and loads data into the memory units, delegates and responds to interrupts from the Execution Controller (EC) and performs the final sum operation. It is to be used as a stand-alone unit and generate configuration instructions for the RPE's (delivered via the execution controller). Depending on user's requirements and system constraints, it will partition the input tensor data ($Tensor_1$ and $Tensor_2$) into the RAM units via the DMA

Controller at different addresses (64-bit addressing is used in the system). $Tensor_1$'s terms and coefficients ($\mathbf{t}_{1i,}\mathbf{c}_{1i}$) are loaded into the CAM unit whereas terms alone ($\mathbf{t}_{1i}$) are loaded into the RAM units at their respective start addresses. $Tensor_2$'s terms and coefficients ($\mathbf{t}_{2j,}\mathbf{c}_{2j}$) are both loaded at consecutive addresses into the RAM units immediately following $Tensor_1$. At the conclusion of the operation of each participating RPE, the execution controller generates an interrupt; triggering the core to issue a single AXI BURST_READ transaction to fetch the partial sums from the RPE's and accumulates s_{12} (line 15 of Algorithm-4).

5.5.2 Design of Execution Controller

The execution controller is designed as an independent unit to: initialize the RPEs, deliver configuration information, monitor their progress and generate an interrupt to the host processor when the delegated task is complete. Its 3 key operations are summarized below:

5.5.2.1 Configuration & Monitoring of BF Set Phase

The EC sends the following configuration registers to the RPE : (*read_start_address*$_1$, *num_data*$_1$, *operation_id*) – the address for where to read $\mathbf{t}_{1i}$'s from, how many entries to read and *operation_id=32'h00000001*. This is performed as a BURST3_WRITE transaction from AXI_Master ports on the EC → NIC-301 → AXI to APB Bridge → RPE's APB slave port. AMBA Advanced Peripheral (APB) bus is used because we need to send configuration data one-time, and it has a much reduced (5) signals – low complexity. The execution controller also receives a completion signal (32

bit, equal to *operation_id*) from the RPE's once the BF set phase is complete. On receiving this signal, the EC proceeds to stage 2.

5.5.2.2 Configuration & Monitoring of BF Test Phase

The EC delivers the following config. Registers to the RPE: (*read_start_address*$_2$, *num_data*$_2$, *CAM_address*, *core_id*, *operation_id*). These are similar in function to the above except that *read_start_address* and *num_data* now represent from where the RPE would read ($\mathbf{t}_{2j}$,$\mathbf{c}_{2j}$) and how many it would read. The operation_id=*32'h00000002*. Since several replicated CAM units are provided to reduce latency of the lookup operation (lines 8-11 of Algorithm 4), the EC would allocate a specific CAM unit to a core. Further, a *core_id* is also provided to the RPE's by each EC as an identity notifier. The RPE would in turn transmit this *core_id* to the corresponding CAM unit to enable the CAM in turn to distinguish between several incoming lookup requests from several RPE's. As earlier, the EC waits for the completion signal line from the core to learn that the BF test phase is complete and proceed to stage 3.

5.5.2.3 Retrieving Intermediate Sum & Generating Interrupts

The BF set phase will execute concurrently on all RPE's unless there is a memory bottleneck. However, the BF test phase on the RPE's will run asynchronously because it is data-dependent. If a particular RPE has a large number of potential matches (*test_success=1*), it will wait on the CAM units longer to return the corresponding coefficient(s) of the candidate matching tensor term(s) ($\mathbf{c}_{1m}$'s in line 10 of Algorithm 4). Thus different RPE's will terminate at different times and will in that order inform the

EC. The EC will generate an interrupt to the host processor, which in turn will fetch the partial sums (AXI BURST1_READ) from the terminated RPEs.

5.5.3 Design of the RPE

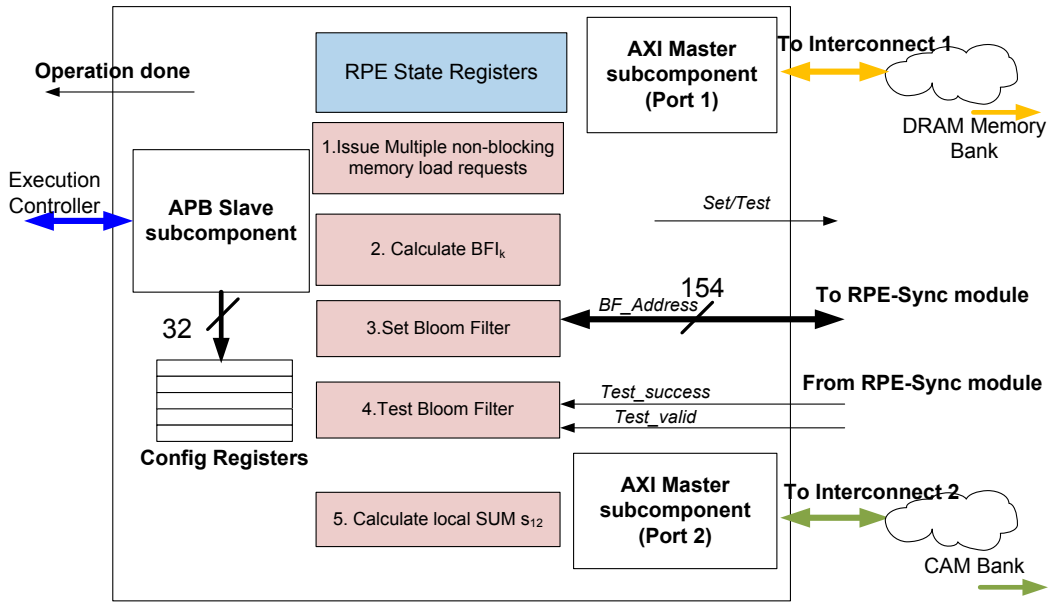

Figure 47. Reconfigurable Processing element (RPE) Design for SIF

Figure 47 presents a schematic overview of the RPE design for the SIF kernel (Algorithm 4). The RPE consists of 5 distinct stages that are executed as part of two phases – *set* and *test*. During the *set* phase, stages 1,2 & 3 are executed serially whereas during the *test* phase stages 1,2,4 and 5 are executed serially. Configuration instructions dispatched from the EC are stored in 32-bit configuration registers (functionality discussed in earlier section). An APB slave subcomponent was implemented to obey the protocol standards. These configuration registers drive the RPE state machine (described

through Figure 48. Memory read requests to the memory banks are issued via the read channel of the AXI master component (also implemented ab-initio to obey protocol standards). We use the AXI_BURST_READ16 in the increment mode for highest throughput. Once a *basis_vector* term is received on the RDDATA lines, it is passed into an IP block to generate the required k Bloom Filter indices (BFI). Each BFI_k is 22 bits long (corresponding to the bit-address of a $m=2^{22}$=4Mi wide BF bit-vector). We use $k=7$ indices per *basis_vector* term, which corresponds to a *BF_Address* bus 154-bits wide. A *BF_Address* line for every core is routed externally to a BF-Sync module (described subsequently) for setting or testing a Bloom filter. Once a sufficient number of data (typically involving multiple AXI_BURST_READ16 transactions) the set phase is complete.

The *test* phase proceeds in an analogous manner. In this phase however, each core is expected to read both the (*basis_vector,coefficient* $\mathbf{c}_{2j}$) corresponding to a subset of $Tensor_2$ allocated to it by the EC (in turn the ARM Cortex A9 processor). The key difference being that once the BFI_k bits for the test *basis_vector* are generated, they are now tested for prescence in the BF. In case, the *Test_success* is a true (shown as an input from the *RPE_Sync* module), it automatically triggers a lookup for the corresponding coefficient of $Tensor_1$ (lines 8-11 of Algorithm 4). This lookup operation is issued by the RPE from its second AXI Master port as an AXI_BURST_READ1 on a 128-bit address. The lower 10-bits of the 128-bit address are used to identify the cores identity (*coreID*). The next 64-bits carry the *basis_vector* of the candidate match. The remaining 54 bits are sufficient to define the routing table for the interconnect. In case a CAM request is

issued, the core does not stall; it proceeds forward in testing the next *basis_vector* in queue (pipelined). When the CAM request is returned with the corresponding coefficient of the candidate *basis_vector* i.e. $\mathbf{c}_{1m}$, it is then multiplied with $\mathbf{c}_{2j}$ yielding the partial sum (line 10 of Algorithm 4).

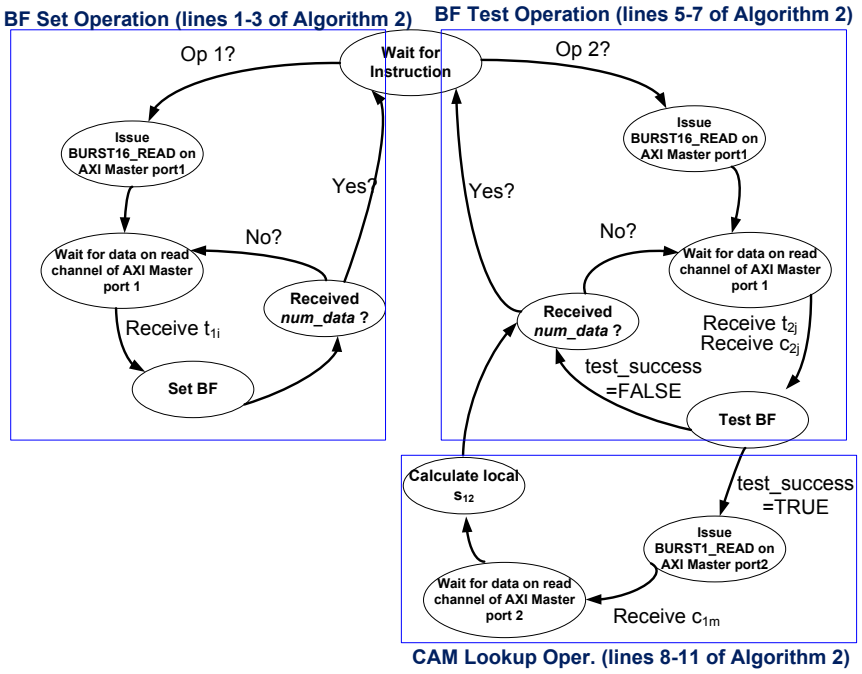

Figure 48. State Diagram for an RPE executing SIF Kernel

5.5.3.1 Computation of Bloom Filter Indices

A Bloom Filter requires the use of *k* independent hash functions to generate *k* index values. The first convenient method of implementing this is to deploy *k* separate hash functions such as *RSHash, JSHash, PJWHash, ELKHash, BKDRHash, SDBMHash,*

DJBHash, DEKHash etc. An alternate method to obtain k independent hash values is to combine the output of only two hash functions $h_1(x)$ and $h_2(x)$ with the formulation $BFI_k = h_1(\mathbf{t}_i) + ih_2(\mathbf{t}_i)$ where $i \in \mathbb{I}$ without any degradation in false+ve probability [44]. In Table 9, we show different alternative mathematical operations that could be used to combine two initial hash functions and generate BFI_k. $\oplus$, $\times$, + $rot(A,j)$ represent the bitwise XOR operation, multiplication, addition and rotation of A by j bits. Each method was verified in a statistical simulator to experimentally measure its $p_{false+ve}$.

It was observed that multiplication (effectively a bit-shift to the left) introduces zeroes into vacated bit positions reducing the entropy. A subsequent XOR operation with these 0's would retain the previous value, several bit-positions would become deterministic; thereby reducing the effectiveness of the bloom filter indices. In contrast a circular rotation operation preserves the entropy in the original data. Further, bit-wise XOR has a significantly lower power draw than an adder; hence method 5 was chosen as the preferred method to generate BFI_k.

Table 9. Alternative Methods to generate Bloom Filter Indices (BFI)

Method	Operation	Power	Randomness
1	$h_1(\mathbf{t}_i) + rot(h_2(\mathbf{t}_i), i)$	557 μW	Fair
2	$h_1(\mathbf{t}_i) + i \times h_2(\mathbf{t}_i)$	88 μW	Poor
3	$h_1(\mathbf{t}_i) + 2^i \times h_2(\mathbf{t}_i)$	637 μW	Poor
4	$h_1(\mathbf{t}_i) \oplus i \times h_2(\mathbf{t}_i)$	58 μW	Poor
5	$\boldsymbol{h_1(t_i) \oplus rot(h_2(t_i), i)}$	***61 μW***	***Excellent***
6	$h_1(\mathbf{t}_i) \oplus 2^i \times h_2(\mathbf{t}_i)$	92 μW	Poor

A further simplification can be done to efficiently generate the two primary hash functions $h_1(\mathbf{t}_i)$ and $h_2(\mathbf{t}_i)$. $h_1(\mathbf{t}_i)$ was generated as the Fowler/Noll/Vo (FNV) hash [45] of t_i as opposed to the more popular MD5 or SHA2 because of its ease of implementation. The values of *offset_basis* and *FNV_prime* are defined in [45]. The second value $h_2(\mathbf{t}_i)$ is generated from the first value by XORing every octet of the first hash function (lines 9-11). The initial value for $h_2(\mathbf{t}_i)$ in turn is obtained by XORing the *offset_basis* with the original string. Both the parallel *foreach* blocks operate on an octet of $\mathbf{t}_{1i}$ (or $\mathbf{t}_{2i}$) and predefined constants; therefore they can be reliably realized as a single-cycle operation. The overall algorithm for the generation of the two hash functions $h_1(x)$ and $h_2(x)$ is shown as Algorithm 5.

Algorithm 5: Generation of $h_1(\mathbf{t}_i)$ & $h_2(\mathbf{t}_i)$ for BFI_k

---

Inputs: $\mathbf{t}_{1i}$ (or $\mathbf{t}_{2i}$)
Output: $h_1(\mathbf{t}_{1i})$ and $h_2(\mathbf{t}_{1i})$
1 //Generate FNV_1A hash of $\mathbf{t}_{1i}$ [45]
2 $h_1(\mathbf{t}_{1i}) = offset\_basis$
3 **parallel foreach** *octet* of $\mathbf{t}_{1i}$ **do**
4 $h_1(\mathbf{t}_{1i}) = h_1(\mathbf{t}_{1i}) \oplus octet(\boldsymbol{t}_{1i})$
5 $h_1(\mathbf{t}_{1i}) = h_1(\mathbf{t}_{1i}) \times FNV_{prime}$
6 **end for**
7 $h_2(\mathbf{t}_{1i}) = offset\_basis \oplus \boldsymbol{t}_{1i}$
8 **parallel foreach** *octet* of $h_1(\mathbf{t}_{1i})$ **do**
9 $h_2(\mathbf{t}_{1i}) = h_2(\mathbf{t}_{1i}) \oplus octet(h_1(\boldsymbol{t}_{1i}))$
10 **end for**

5.5.4 Design of the RPE-Sync Module

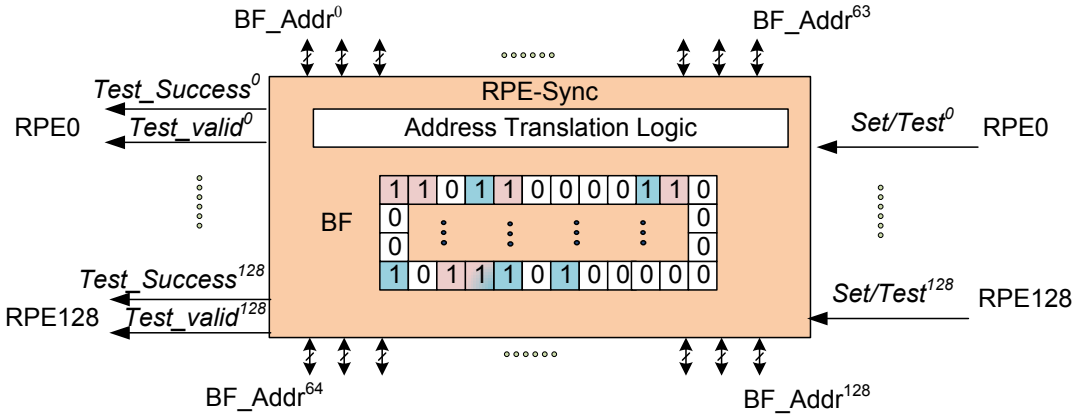

Figure 49. Construction of the RPE-Sync Core

Figure 49 shows the internal construction of the RPE-Sync core. This core enables the creation of a synchronous shared on-chip Bloom Filter for the RPE's. In this dissertation, the size of the bit-vector is assumed to be *m=4Mi*. RPE-Sync receives a *set/test signal* (active high) signal and a 154-bit wide *BF_Addr* from each RPE. The *BF_Addr* bus consists of k=7, 22-bit address bits for setting/testing the BF. In case the set-signal is asserted, this 22-bit address is used to set the corresponding bit-vector position high else is used to test whether the corresponding position is high. Memory is addressed in 1-byte chunks, therefore a simple address translation logic is necessary. Since there is no possibility of race conditions (BF bits are never set to 0) and that the $7\times128 = 896$, 22-bit address lines are guarenteed to be distinct (with $p_{false+ve}$ probability), we can safely perform the setting/testing of all valid *BF_Addr* lines in a

single cycle. It should also be noted that whereas the address bus routing is for all cores, it is not necessary that all cores will be active at the same time (discussed subsequently).

5.6 Analysis of Alternate Compute Models for Organization of RPEs

In this section we analyze three alternate compute models for the design of the RPE's for SIF. This is intended to provide justification for our chosen compute flow for this application. Let's assume that we have to process two tensors of sizes $p=q=N$ (equality assumed for simplicity). This data is organized in r memory banks. Let's also assume that the RPE's can be logically organized in β *groups* of α RPE's each.

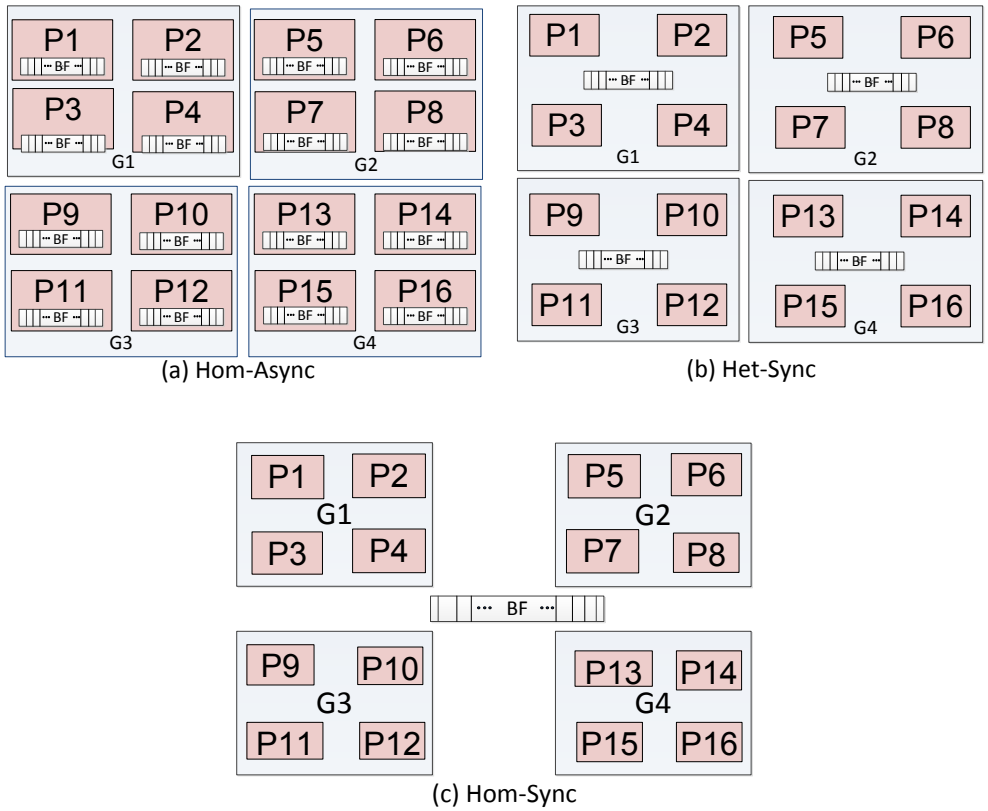

Figure 50. Alternative Computational Models for RPE organization

A group of RPE's could then be designated to be either homogenous and asynchronous (*Hom-Async*) or heterogenous and synchronous (*Het-Sync*) or homogenous and synchronous (*Hom-Sync*) with reference to the location of the Bloom Filter bit-vector. Figure 50 shows a representative example of the possible choices with α=β=4; i.e. total number of RPE's ($\alpha\beta$)=16.

Table 10. Time Complexity of Set and Test Operations with SIF Compute Models

Model	**Complexity (Set Operation)**	**Complexity (Test Operation)**
Hom-Async	$O(\frac{N}{\alpha\beta})$	$O(N)$
Het-Sync	$O(\frac{N}{\alpha})$	$O(\frac{N}{\alpha\beta})$
Hom-Sync	$O(\frac{N}{\alpha\beta})$	$O(\frac{N}{\alpha\beta})$

Table 10 shows the estimated time complexity for the set & test operations for alternate SIF compute models. This is explained in this section. In the homogenous asynchronous model (*Hom-Async*), each RPE is provided with its own independent BF. Each RPE can fetch data from its allocated memory banks and fill in its private BF during the set operation in $O(\frac{N}{\alpha\beta})$. However, during the test operation, each test tensor entry (out of *N* possible) will need to be tested at each of the $\alpha\beta$ RPE's in *O(N)* time (inefficient). In contrast, for a heterogeneous synchronous model (*Het-Sync*), every group of α RPE's share a private BF. Each of the β groups in *Het-Sync* will therefore create distinct fragments of the overall BF during the set phase. Since each of the α

RPE's in a group will operate on different data, *N* entries can be processed by β groups in $O(\frac{N}{\alpha})$ time. Since the BF's becomes the same, all processing units in a group can now test *N* entries of the tensor in $O\left(\frac{N}{\alpha\beta}\right)$ time. The least complexity (but densest interconnects) can be achieved with the homogenous synchronous model (*Hom-Sync*) where a single BF is shared by all participating RPE's. This is the computational model followed in the previous section. Because of homogeneity of the cores, no distinction into groups is also necessary. Each of the $\alpha\beta$ cores can process independent row entries of the tensor during the set and test phases respectively. Although this dramatically reduces the complexity of the set and test operations to $O\left(\frac{N}{\alpha\beta}\right)$, synthesis of the RPE-Sync module will involve a large number of interconnects from the cores. We would also like to note that the *Hom-Async* and *Het-Sync* models will involve multiple copies of the BF bit vector – on-chip registers are expensive real-estate on-chip (inefficient).

5.7 Validation Methodology

This section provides details of the virtual prototyping tool chain and experiments. Initial functional verification of the design was performed using ModelSim from Mentor Graphics after being implemented in RTL (Verilog). The core components of the design were then synthesized using Synopsys Design Compiler at a clock speed of 3 GHz using the 90nm technology library from TSMC. Given the complexity of the reconfigurable IP and the need to examine its behavior in the context of a data-intensive application, a full SoC virtual prototype was created using Carbon Model Studio & Carbon SoC Designer from Carbon Design Systems [46].

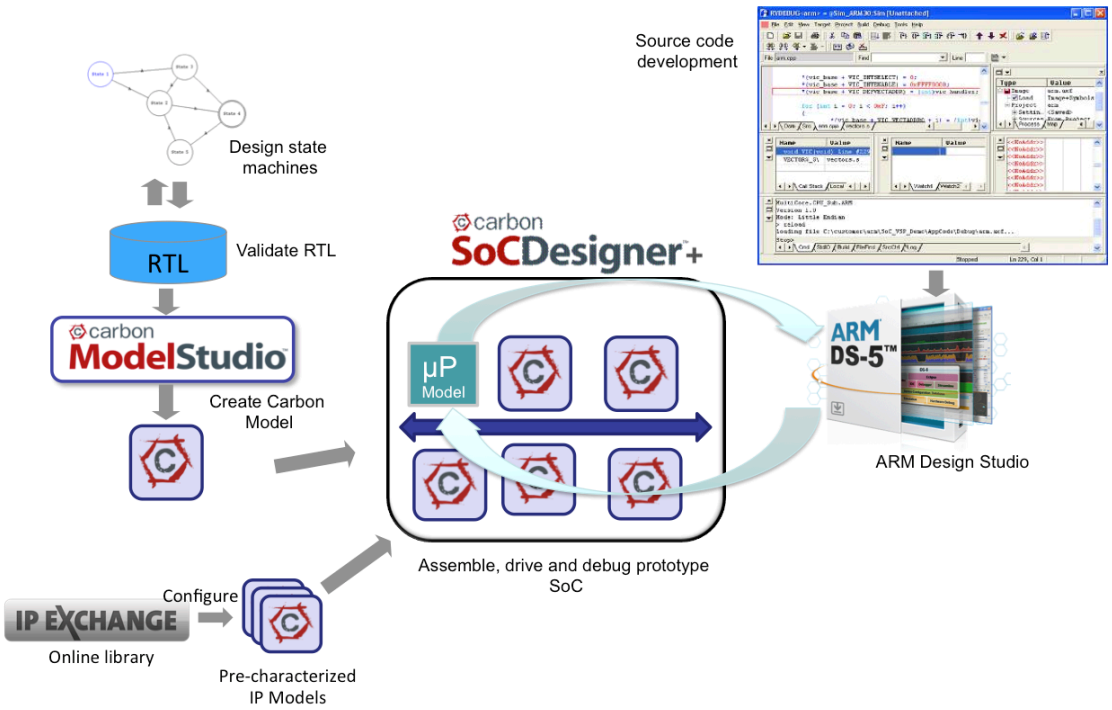

Figure 51. SoC Validation Tool-chain

Carbon Model Studio and Carbon Compiler enables the creation of a high-performance linkable software object that contains a cycle and register accurate model of the hardware design directly from the RTL design files. An object library file, header and database with information on all signals, top-level I/O's together are described as a Carbon Model. Such *carbonized* models were created for the RPE's, RPE-G, BF-Sync and CAM modules (Figure 51), linked with *gcc* and then run on Carbon SoC Designer (essentially an Instruction Set Simulator). Carbon Models for additional components in the SoC such as the ARM Cortex-A9 processor, AXIv2 compliant memory units, memory controller and the AMBA compliant network infrastructure (NIC-301) was obtained with permission from ARM, configured adequately using AMBADesigner Lite

(to generate XML configuration files) and then custom built on Carbon's online IP Exchange. These interconnects are pre-qualified and regression tested by ARM. This entire package of components were assembled and analyzed for system behavior. SoCDesigner provided us detailed visibility and step-by-step execution control of the design. It was particularly helpful to be able to model and examine the behavior of a complex bus architecture such as AMBA AXI, measure interconnect performance and drive real application traffic through the prototype SoC. Suitable design changes were actually made (such as pipelining the CAM lookup operation, separate interconnect and ports for data load and lookup operations) after exploring the design space alternatives and preliminary system integration results. Figure 51 also shows that the ARM processor model (μP model) is driven by source code/kernel firmware written in C/C++ compiled using ARM Design Studio tool chain.

5.8 Results and Discussion

The experiments to validate the hypothesis of the previous section focus on parallelizing a single semantic comparison and were conducted considering the worst case $p=q=N$ in mind. In a real-world search engine or recommender system either p or q is expected to be significantly smaller than the other. We experiment for (1) N varying from 100 to 160,000 rows and (2) similarity c varying between 10% to 100% (complete match), (3) the number of RPE's $\alpha\beta$=n varying from 32 to 128. Initial functional validation & correctness was verified by comparing s_{12} computed using a functional simulator (written in C++) executing Algorithm 2 and the results from Carbon's SoC Designer.

Table 11 shows the constant latencies of some basic operations involving the AMBA AXI & APB protocols in the proposed architecture, latencies for the fundamental operations such as generation of Bloom Filter Index, set and test operation. This table is valid when the RPE has control on the corresponding bus.

Table 11. Latencies of Basic Operations in Proposed Architecture

	Basic Operations	**Latency**
1	*RPE read from memory* (AXI BURST_READx)	5
2	*RPE read from memory* (AXI BURST_READ16 including 16 cycles for 16 entries)	16+5 = 21
2	*RPE read from CAM* (AXI BURST_READ1, includes single-cycle CAM lookup)	9
3	*RPE set configuration* (AXI BURST_READ4)	10
4	*Processing delay at RPE* (Multiplication, partial sum)	~1 cycle
5	*RPE calculate BFI*	1
6	*CAM lookup*	1

5.8.1 Execution Time

We measure the end-to-end execution time under varying conditions: (1) varying sizes of synthetic dataset, (2) change in characteristics of the dataset, (3) number of execution units (RPEs) αβ.

5.8.1.1 Execution Time with Varying Size of Dataset

Figure 52 shows the averaged overall and phase-wise execution time for Tensor sizes p=q=N=25k to 160k when using 32 RPE's. This represents the worst case of the comparisons that RPE's can be expected to perform since in real-life scenarios the two tensors will be expected to be p<<q. In this case, the number of RPEs used $\alpha\beta$=32 i.e. each RPE has access to an independent memory bank (r=32), an independent CAM bank (s=32) – an ideal case for the architecture.

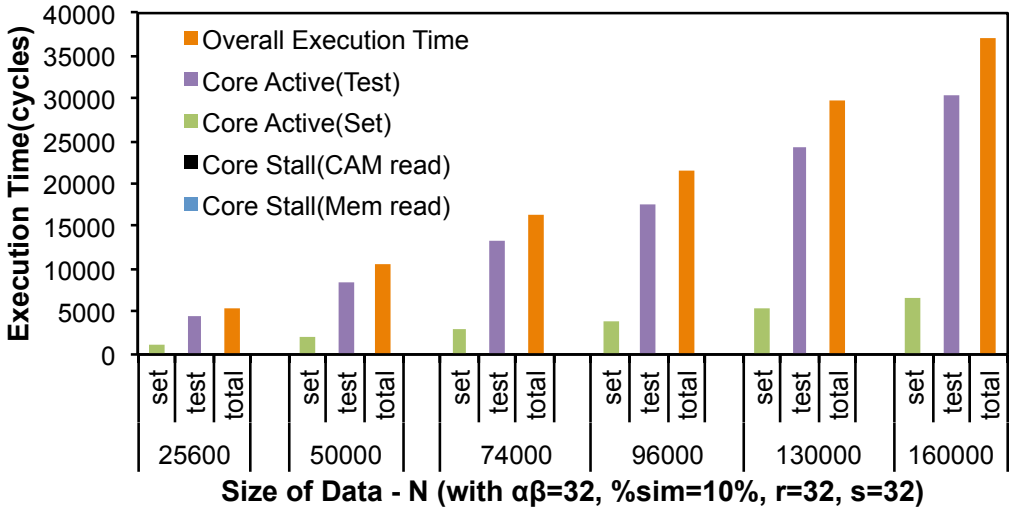

Figure 52. Execution Time with Varying Tensor Size (#Cores=32)

This graph indicates that the set phase occupies less than 5000 cycles for the largest tensor size under experimentation. As described earlier, there is a latency of only one cycle between data received and BFI generated. In our RPE, these operations are pipelined. Therefore, the set phase should execute for exactly 160000/32 = 5000 cycles,

which is indeed the case. Likewise during the test phase, each core will receive (*basis_vector,coefficient*) data (for Tensor 2) from its dedicated memory bank for (5000*2)=10000 64-bit entries. This will take $\frac{10000}{16} = 625$ BURST_READ16 transactions in 10000+625*5=13125 cycles. Each core will then issue separate CAM lookup requests depending on the number of *Test_success* it receives (generation of *Test_success* happens in 1 cycle). In this case, a total of 16000 entries (10%) in $Tensor_2$ are expected to return it (distributed across the 32 cores). On an average, we determined that each core generates 858 *Test_success*. This yields an additional 858*9=7772 cycles. An additional latency of ~1 cycle (Processing delay at RPE) occurs for each of the 10,000 elements. This value is quoted as ~1 cycle because of deep pipelining between the stages. Therefore the total cycles taken is 13125+7772+10000 = 30897 (agrees with the results above).

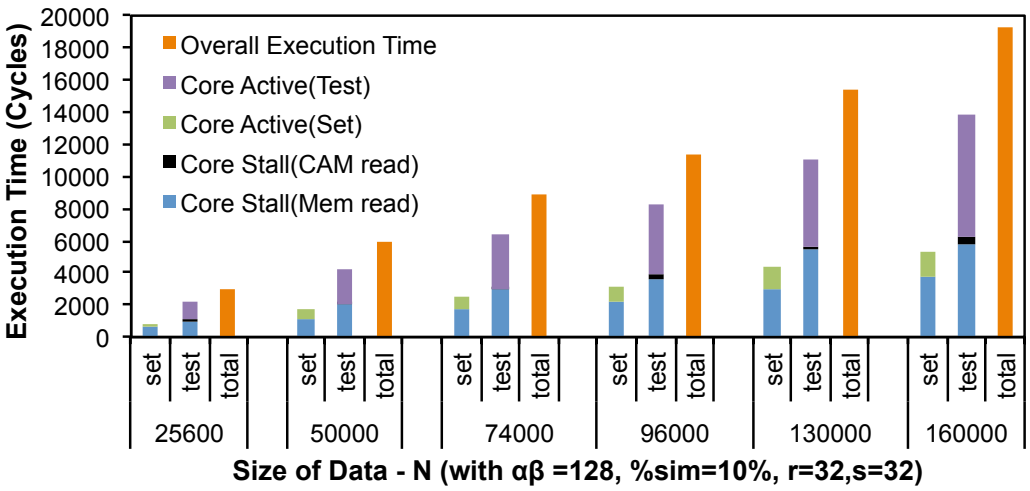

Figure 53. Execution Time with Varying Tensor Size (#Cores=128)

Figure 53 shows the overall and phase-wise execution time for Tensor sizes p=q=N=25k to 160k when using #cores = 128. We observe that the total execution time is lower by ~2x as compared to #cores=32. This is lower than the expected ~4x because of cores stalls. Since #Memory banks (r) and #CAM banks = 32 has been kept fixed, several cores are starved for data. The core stall is worse for the test-phase because it is reading a larger amount of data from memory (both *basis_vector & coefficient* for $Tensor_2$). CAM lookup stalls also manifest for larger data sizes because of a higher probability of *Test-success*.

5.8.1.2 Execution Time with Varying Percentage Similarity between Tensors

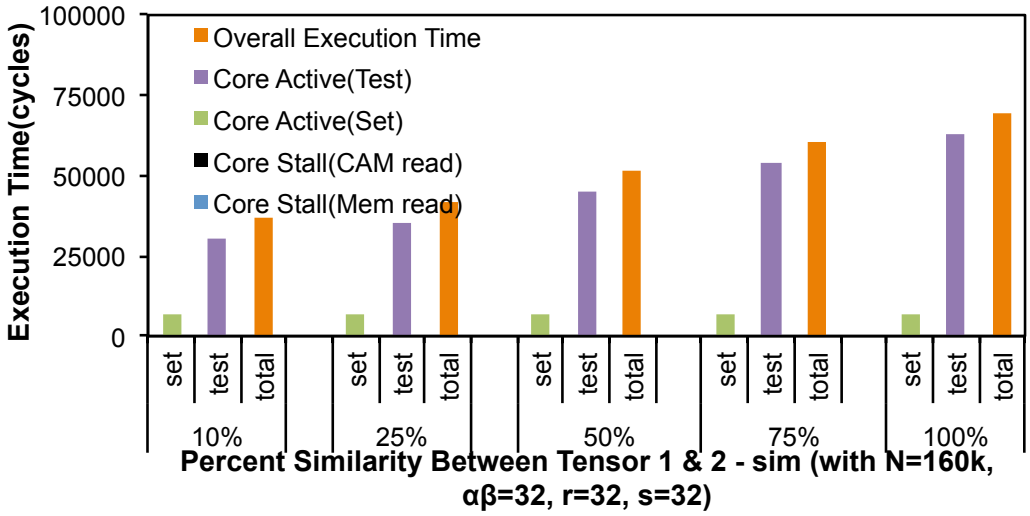

Figure 54. Execution Time with Varying %Similarity

Figure 54 shows the sensitivity of the proposed architecture to variation in similarity (number of common basis vector terms) between two tensors of size

p=q=N=50k. The execution time in the worst case (100% similarity) is ~60000 cycles decreasing to ~35000 cycles for 10% similarity. These experiments were performed with #cores= $\alpha\beta$=32. No core stalls are expected since each core has independent access to its own memory & CAM bank.

5.8.1.3 Execution Time with Varying Number of Execution Units

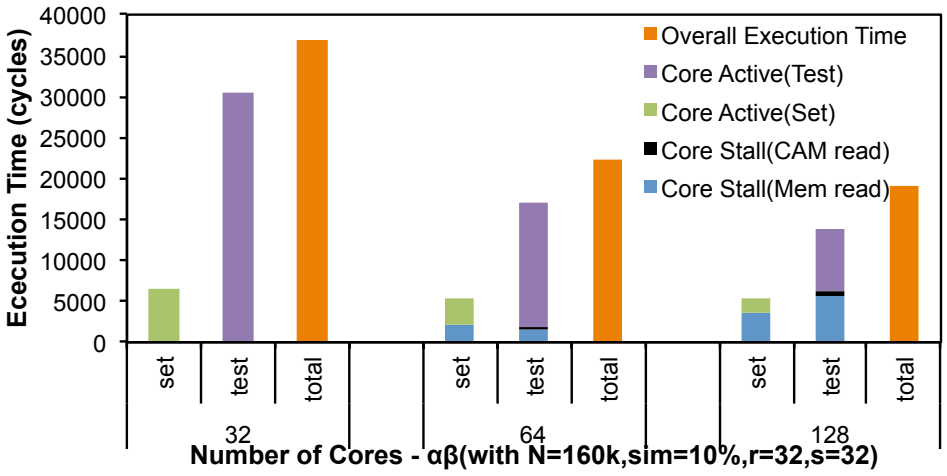

Figure 55. Variation of Execution Time with Varying Number of Cores

Figure 55 shows the execution time for SIF between two tensors of size 160k with #cores in use varying between 32-128. The number of memory banks and CAM blocks is kept constant at 32. Core stall behavior is observed when #cores=64 and becomes progressively worse with #cores=128. Therefore, although the overall execution time does decrease by ~2x on increasing #cores from 32 to 64, this effect is much less pronounced in the jump from 64 to 128. This bottleneck can be resolved in

datacenter environments by increasing the number of off-chip memory banks or CAM units in use.

5.8.2 Comparison with Contemporary Many-core Processors

In Table 12, we present a comparison of SIF on three contemporary many-core processors for *p=q=N=160k*, *sim=10%* and number of cores=32. We can see from the previous sections that the proposed architecture has been simulated to require ~10k cycles. The same algorithm when run on an Intel SCC, Nvidia Tesla C870 and Nvidia Kepler GTX680 require ~3.944 Gi, ~2.784 Gi and ~633 Mi cycles respectively. This provides us a speedup of ~98K, ~68K and ~15K respectively for the proposed many-core SoC architecture over the Intel SCC, Nvidia Tesla C870 and Nvidia Kepler GTX680 respectively. This is despite the fact that Kepler GPUs have a significantly higher number of cores in use (512 v/s 128). The 48-core SCC performs worst because it requires a round-robin baton-passing algorithm to synchronize the BF across cores. Whereas the Nvidia Tesla & Kepler are by design shared-memory many-core processors, they lack a mechanism to perform the CAM lookup (lines 8-11 of Algorithm 2) and the ability to compute Bloom filter indices in a single cycle (due to specialized functional units SFUs). Therefore, we can expect to have a significant performance improvement with the proposed SoC over contemporary many-core architectures.

Table 12. Comparison of Proposed Architecture with Intel SCC and Nvidia GPUs

	Intel SCC [47]	Nvidia Tesla [9]	Nvidia Kepler [28]
Core Architecture	X86 Pentium - I	C870	GTX680
Technology	45 nm	90 nm	28 nm
Number of Cores	48	128	512
Main Memory	32 GB	4GB	6 GB
Memory Bandwidth	800 Mbps	76.8 Gbps	192.4 Gbps
Clock Speed	533 MHz	772MHz	1006 MHz
Execution Time (Cycles)	3.944 Gi	2.784 Gi	633 Mi
Speedup	98605	68700	15825

5.9 Related Work

Tensor analysis and large scale semantic information filtering has received considerable interest in literature. [25] describes the parallel implementation of a document similarity classifier using Bloom Filters on two contemporary many-core platforms: Tilera's 64-core SoC and Xilinx Virtex 5-LX FPGA. This work has been done in the context of web-security and demonstrates that an incoming data stream can be filtered using a TF-IDF based dictionary of known attack patterns. Although this work does not use tensors (reverts to the conventional vector-based models to represent information), they use a large array of Bloom filters at each processing element. We also differ from this work because our Bloom filters are dynamically created on-chip based on input data whereas those in the paper are assumed to be statically generated, offline. Thirdly, we generate a semantic similarity value as an output, which can be used to recommend new items whereas the prior work generates only the filtered data stream. Kang et al [48] describe a MapReduce model to accelerating tensor analysis by 100x. As discussed earlier in this dissertation, a traditional cluster deployment may provide

scalability & performance improvement but at significant infrastructure cost. Our approach instead focuses on improving efficiency at the compute node-level when they are many-core processors. Our previous work [9, 28] describe a method to use a BF-based algorithm on a GPU, which unfortunately is (1) limited by memory throughput, (2) uses general purpose cores to port the BF algorithm discussed above, (3) have no capability to implement custom logic on the cores or interface off-chip CAM to provide fast lookups. Consequently GPU deployments will suffer from an inability to parallelize lines 8-11 of Algorithm 2. Our previous work in [26] presents a fine-grained parallel ASIC to realize Algorithm-2. This does not demonstrate scalability beyond p=q=N=1024 and does not consider the impact of memory latency in presenting results. Secondly, the arbiters designed between the stages are not scalable for big-data workloads. Further, for big-data applications a reconfigurable computing template is favorable than a fixed-function ASIC.

5.10 Section Summary

Workload specific server configurations for Big-Data applications are already a reality [49-51]. With increasing sensitivity in industry and government about energy-efficiency in big-data infrastructures, workload specific accelerators are expected soon. Workload specific accelerators for Big-Data will need to be reconfigurable to be applicable to a wide array of similar applications, provide extra-ordinary energy savings and high-performance to be a compelling alternative. In this paper, we described a novel reconfigurable computing architecture template and an application-specific architecture for efficient and scalable semantic information filtering. We have shown through

experiments that our approach is able to outperform the state-of-the-art SIF implementations on many-core processors such as a GPU and Intel's SCC by more than ~98K times for tensor sizes of 160000. To the best of our knowledge, we are the first in academia to investigate the design, undertake the hardware-software codesign effort to realize a many-core reconfigurable SoC for data intensive applications using industry standard tools. While the initial results are indeed promising, additional scalability benefits can be obtained by using AMBA compliant packet-based interconnects (NoC) for additional scalability and improved throughput, dynamic scheduling to improve workload balance. Other big-data applications in the information-filtering template such as collaborative filtering can also be designed.

6. CONCLUSIONS AND FUTURE WORK

6.1 Future Work

This dissertation explored the design and interplay between a few of layers of the application stack at a many-core compute node for high performance information filtering applications. This dissertation explored the data structure and algorithms required, programming models that could be used, run-time systems that should be deployed, memory and IO pipelines that were necessary to achieve high performance. However, there are still several areas of research, which have not been explored. Some of the potential areas of future work are listed below.

1. Hybrid Information Filtering – While this dissertation explored algorithms & architectures to accelerate content and collaborative information filtering separately, the same will need to be done for them together. New seminal works such as [52] have shown that further improvement in accuracy of recommendation algorithms is possible only due to their combination. In addition, this dissertation has explored the neighborhood based collaborative filtering methods; techniques to accelerate latent factor methods remain to be explored.
2. Expansion of Information Filtering benchmarks – A few automated techniques to generate tensor representations from arbitrary sentences were demonstrated. Further improvements such as including discourse analysis can be added to further improve its effectiveness and scope. Human trials could be conducted to conclusively demonstrate the

superiority of the tensor method for semantic information filtering as compared to the traditional vector based methods.

6.2 Conclusion

This research proposes techniques for mapping data-intensive computational kernels for semantic and collaborative information filtering on many-core architectures. Two representative many-core architectures - GPUs and Intel's SCC have been examined, efficient data structures & algorithms have been designed, shared and distributed programming models/run-times designed, architectural features have been explored to exploit concurrency within the machine boundary efficiently. A computing run-time for distributed memory many-core processor (such as the Intel SCC) has been designed and information filtering applications modeled on this run time. Finally, a reconfigurable SoC template for data intensive applications has been proposed. This has been tested with semantic information filtering as the application context to obtain significant speedups. Therefore, we can claim that in applications where compute requirements are short; a scheme combining both fine-grained and coarse-grained parallelism can provide highest performance.

REFERENCES

[1] Gantz, J., and Reinsel, D., "Extracting Value from Chaos", http://www.emc.com/digital_universe, 2011, Accessed: 06-01-2013.

[2] Adomavicius, G., and Tuzhilin, A., "Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-art and Possible Extensions", IEEE Transactions on Knowledge and Data Engineering, 2005, 17, (6), pp. 734-749.

[3] Dean, J., and Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters", Communications of the ACM, 2008, 51, (1), pp. 107-113.

[4] Bennett, J., "The Cinematch system: Operation, Scale Coverage, Accuracy Impact", http://borrelli.org/assets/pdfs/netflix/bennett.pdf, 2006, Accessed: 09-12-2009.

[5] Mitchell, J., and Lapata, M., "Vector-based Models of Semantic Composition". Proc. ACL-08: HLT, 2008 pp. 236-244.

[6] Barroso, L.A., Dean, J., et al., "Web Search for a Planet: The Google Cluster Architecture", IEEE Micro, 2003, 23, (2), pp. 22-28.

[7] Biswas, A., Mohan, S., et al., "Semantic Key for Meaning Based Searching". Proc. 2009 IEEE International Conference on Semantic Computing, Washington, DC, USA, 2009 pp. 209-214.

[8] Biswas, A., Mohan, S., et al., "Representation of Complex Concepts for Semantic Routed Network". Proc. 10th International Conference on Distributed Computing and Networking (ICDCN), Hyderabad, India, 2009 pp. 127-138.

[9] Tripathy, A., Mohan, S., et al., "Optimizing a Semantic Comparator Using CUDA-enabled Graphics Hardware". Proc. 5th IEEE International Conference on Semantic Computing (ICSC2011), Palo Alto, CA, 2011 pp. 125-132.

[10] Perez, J.C., "Google joins crowd, adds semantic search capabilities", http://www.computerworld.com/s/article/9130318/Google_joins_crowd_adds_semantic_search_capabilities, 2009, Accessed: 03-04-2010.

[11] Bird, S., Klein, E., et al., "Natural Language Processing with Python" (O'Reilly Media, Inc., 2009).

[12] Panigrahy, J., "Generating Tensor Representation from Concept Tree in Meaning Based Search". Master's Thesis, Texas A&M University, 2010.

[13] Woo, D.H., and Lee, H.-H., "Extending Amdahl's Law for Energy-efficient Computing in the Many-core Era", Computer, 2008, 41, (12), pp. 24-31.

[14] Lindholm, E., Nickolls, J., et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture", Micro, IEEE, 2008, 28, (2), pp. 39-55.

[15] Miller, F.P., Vandome, A.F., et al., "AMD Firestream: ATI Technologies, Stream Processing, Nvidia Tesla, Advanced Micro Devices, Gpgpu, High-Performance Computing, Torrenza, Radeon R520, Shader" (Alpha Press, 2009).

[16] Wittenbrink, C.M., Kilgariff, E., et al., "Fermi GF100 GPU Architecture", Micro, IEEE, 2011, 31, (2), pp. 50-59.

[17] Nvidia, "Nvidia's Next Generation CUDA compute architecture - Kepler GK110", http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2013, Accessed: 04-09-2013.

[18] Mattson, T.G., Van der Wijngaart, R.F., et al., "The 48-core SCC Processor: the Programmer's View". Proc. Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC2010), , 2010 pp. 1-11.

[19] Tripathy, A., Mohan, S., et al., "Optimizing a Collaborative Filtering Recommender for Many-Core Processors". Proc. 6th IEEE International Conference on Semantic Computing (ICSC), Palermo, Italy, 2012 pp. 261-268.

[20] Broder, A., and Mitzenmacher, M., "Network Applications of Bloom Filters: A Survey", Internet Mathematics, 2004, 1, (4), pp. 485-509.

[21] Hwu, W., "GPU Computing Gems Jade Edition. Applications of GPU Computing Series" (Morgan Kaufmann, 2011).

[22] Alcantara, D.A., Sharf, A., et al., "Real-time Parallel Hashing on the GPU". Proc. ACM Transactions on Graphics (TOG), 2009 pp. 154.

[23] Harris, M., "Optimizing Parallel Reduction in CUDA", http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf, 2007, Accessed: 07-06-2010.

[24] Fang, W., Lau, K.K., et al., "Parallel Data Mining on Graphics Processors ", Technical Report HKUST-CS08-07, 2008.

[25] Ulmer, C., Gokhale, M., et al., "Massively Parallel Acceleration of a Document-Similarity Classifier to Detect Web Attacks", J. Parallel Distrib. Comput., 2011, 71, (2), pp. 225-235.

[26] Mohan, S., Tripathy, A., et al., "Parallel Processor Core for Semantic Search Engines". Proc. Workshop on Large-Scale Parallel Processing (LSPP) co-located with IEEE Int. Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, Alaska, USA, 2011.

[27] Chalamalasetti, S., Margala, M., et al., "Evaluating FPGA-Acceleration for Real-time Unstructured Search". Proc. Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on, 2012 pp. 200-209.

[28] Gonen, O., Mahapatra, S., et al., "Exploring GPU Architectures to Accelerate Semantic Comparison for Intention-based Search". Proc. 6th Workshop on General Purpose Processor Using Graphics Processing Units, Houston, Texas, 2013 pp. 137-145.

[29] Li, R., Zhang, Y., et al., "A social network-aware top-N recommender system using GPU". Proc. 11th annual international ACM/IEEE joint conference on Digital libraries (JCDL '11), Ontario, Ottawa, Canada, 2011 pp. 287-296.

[30] Watt's-up, "Electronic Educational Devices, Watts-up Pro", http://www.wattsupmeters.com/, 2009, Accessed: 10-11-2011.

[31] Jamali, M., and Ester, M., "Modeling and Comparing the Influence of Neighbors on the Behavior of Users in Social and Similarity Networks". Proc. 2010 IEEE

International Conference on Data Mining Workshops (ICDMW), Sydney, Australia, 2010 pp. 336-343.

[32] Ziegler, C.-N., McNee, S.M., et al., "Improving recommendation lists through topic diversification". Proc. 14th international conference on World Wide Web, Chiba, Japan, 2005 pp. 22-32.

[33] Herlocker, J., Konstan, J., et al., "An Algorithmic Framework for Performing Collaborative Filtering". Proc. 22nd Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR'99), Berkeley, CA, 1999 pp. 230-237.

[34] Dar-Jen, C., Desoky, A.H., et al., "Compute Pairwise Manhattan Distance and Pearson Correlation Coefficient of Data Points with GPU". Proc. Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD '09. 10th ACIS International Conference on, 2009 pp. 501-506.

[35] Dean, J., "Designs, lessons and advice from building large distributed systems". Proc. 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, Big Sky, MT, 2009.

[36] Thakur, R., and Rabenseifner, R., "Optimization of Collective Communication Operations in MPICH", International Journal of High Performance Computing Applications, 2005, 19, pp. 49-66.

[37] Anastasios Papagiannis, and Nikolopoulos, D.S., "Scalable Runtime Support for Data Intensive Applications on the Single Chip Cloud Computer". Proc. 3rd

Many-core Applications Research Community (MARC) Symposium, Ettlingen, Germany, 2011.

[38] Jing, J., Jie, L., et al., "Scaling-Up Item-Based Collaborative Filtering Recommendation Algorithm Based on Hadoop". Proc. Services (SERVICES), 2011 IEEE World Congress on, 2011 pp. 490-497.

[39] Schelter, S., Boden, C., et al., "Scalable Similarity-based Neighborhood Methods with MapReduce". Proc. 6th ACM conference on Recommender systems, Dublin, Ireland, 2012 pp. 163-170.

[40] Pagiamtzis, K., and Sheikholeslami, A., "Content-addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey", IEEE Journal of Solid-State Circuits, 2006, 41, (3), pp. 712-727.

[41] Kim, Y., and Mahapatra, R.N., "Design of Low-Power Coarse-Grained Reconfigurable Architectures" (CRC Press, 2010, 1st edn).

[42] ARM Inc., "AMBA APB Protocol Specification v2.0", http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0270b/BABEIDCD.html, 2013, Accessed: 02-18-2013.

[43] ARM Inc., "PrimeCell Infrastructure AMBA AXI to APB Bridge (BP 135)", http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dto0014a/BEIJGAAH.html, 2011, Accessed: 12-25-2012.

[44] Kirsch, A., and Mitzenmacher, M., "Less Hashing, Same Performance: Building a Better Bloom Filter", Random Structures & Algorithms, 2008, 33, (2), pp. 187-218.

[45] Fowler, G., Noll, L.C., et al., "Fowler / Noll / Vo (FNV) Hash", http://isthe.com/chongo/tech/comp/fnv/, 2001, Accessed: 06-12-2009.

[46] Carbon-Design-Systems, "Carbon Design Systems Inc. ", http://www.carbondesignsystems.com/, 2011, Accessed: 10-12-2012.

[47] Held, J., "Introducing the Single-chip Cloud Computer - Exploring the Future of Many-core Processors", Intel Labs Whitepaper, 2011.

[48] Kang, U., Papalexakis, E., et al., "GigaTensor: Scaling Tensor Analysis up by 100 Times - Algorithms and Discoveries". Proc. 18th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, Beijing, China, 2012 pp. 316-324.

[49] HP, "HP Project Moonshot", http://www.hp.com/go/moonshot, 2013, Accessed: 03-11-2013.

[50] Dell, "Dell Copper", http://www.dell.com/learn//campaigns/project-copper, 2013, Accessed: 03-05-2013.

[51] AMD, "AMD SeaMicro", http://www.seamicro.com/, 2013, Accessed: 03-05-2013.

[52] Koren, Y., Bell, R., et al., "Matrix Factorization Techniques for Recommender Systems", Computer, 2009, 42, (8), pp. 30-37.