Exploiting Accelerators for Efficient High Dimensional Similarity Search

Sandeep R. Agrawal
Oracle Labs
sandeep.r.agrawal@oracle.com

Christopher M. Dee
Duke University
cmd45@duke.edu

Alvin R. Lebeck
Duke University
alvy@cs.duke.edu

Abstract

Similarity search finds the most similar matches in an object collection for a given query; making it an important problem across a wide range of disciplines such as web search, image recognition and protein sequencing. Practical implementations of High Dimensional Similarity Search (HDSS) search across billions of possible solutions for multiple queries in real time, making its performance and efficiency a significant challenge. Existing clusters and datacenters use commercial multicore hardware to perform search, which may not provide the optimal performance and performance per Watt.

This work explores the performance, power and cost benefits of using throughput accelerators like GPUs to perform similarity search for query cohorts even under tight deadlines. We propose optimized implementations of similarity search for both the host and the accelerator. Augmenting existing Xeon servers with accelerators results in a 3× improvement in throughput per machine, resulting in a more than 2.5× reduction in cost of ownership, even for discounted Xeon servers. Replacing a Xeon based cluster with an accelerator based cluster for similarity search reduces the total cost of ownership by more than 6× to 16× while consuming significantly less power than an ARM based cluster.

Categories and Subject Descriptors C.5.5 [*Computer System Implementation*]: Servers

Keywords high throughput, energy efficiency, total cost of ownership, GPGPU

1. Introduction

Similarity search or nearest neighbor search is a classical problem with many applications across many disciplines. First referred to as the post-office problem by Knuth in his comprehensive work "The Art of Computer Programming", it finds extensive applicability for a wide range of real world problems. In text processing it can be used to identify the most similar documents in a collection of given documents [28]. In web search it can be used to identify the matching pages to a given search query [5]. In image processing, similarity search is used in reverse image search or Content Based Image Retrieval applications to yield the most similar images to a given image [4, 32]. It can be used in protein sequencing to identify the similarity between a new sequence and a database of known proteins [12, 25].

Datacenters currently handle arrival rates of the order of thousands of queries/s [14], but these are expected to increase by several orders of magnitude with the advent of more intelligent services, wearables, connected cars and the Internet of Things [35]. The number of connected sessions will not be limited by the number of connected people, rather by the number of connected devices. According to a recent report by Verizon, the number of connections is expected to increase fourfold to 5.4 billion by 2020 [36]. Much further into the future, the number of devices connected to web services in expected to reach hundreds of billions [35], and the question naturally arises as to what is the best way to satisfy this demand. Server designs based on commodity multicore processors may not be the most cost effective and energy efficient for all workloads, and there is ongoing debate over which architecture is best suited for specific workloads [2, 10, 19, 22, 27, 29, 31].

Throughput accelerators such as NVIDIA's Kepler, and Intel Xeon Phi achieve efficiencies >5 gigaflops/W, by exploiting SIMD and SIMT based hardware to amortize fetch and decode overheads, and executing a large number of threads simultaneously to improve throughput within a limited power envelope. These accelerators are generally considered suitable for scientific or HPC workloads, and too expensive or power hungry for web service workloads like search.

The last decade was an era of cheap off the shelf hardware being used to create large pools of machines to scale to user demand. With the end of Dennard scaling and the creep of dark silicon [9], general purpose cores are no longer viable alternatives for scaleout while operating within a fixed power budget. Even with the release of the Broadwell architecture, Intel is primarily focusing on efficiency improvements, and devoting larger areas of the die to the GPU, with a modest 5% CPU IPC increase [18], providing further evidence that we need to find an alternative for datacenters.

This work argues for the throughput and efficiency benefits of using accelerators for a similarity search workload and searching across billions of documents under deadlines as tight as few tens of milliseconds. This efficiency arises from two observations, 1) even under tight deadlines, for a high enough query arrival rate, a server has the opportunity to delay some requests in order to form *cohorts* of similar requests, and 2) these cohorts can be scheduled on conventional multi-threaded hardware or throughput accelerators to improve throughput and efficiency. Cohort scheduling has been shown to improve cache locality on general purpose multicores [24] and throughput/Watt on accelerators [2]. For the purposes of this work, we define two search queries to be *similar* if they search across the same set of objects.

The goal of this work is to look at high dimensional similarity search from a systems perspective, and provide a methodology to identify the optimal server design to satisfy latency constraints

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.
PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851144

for an incoming query stream by minimizing the total cost of ownership. The primary contributions of this paper are:

- Algorithms for similarity search based on Sparse Matrix Matrix Multiplication for both the host and the accelerator that perform better than existing MKL and cuSPARSE libraries for text based similarity search.
- An implementation of text search on NVIDIA GPUs that is twice as efficient and delivers more than twice the throughput of the corresponding implementation on Xeon and ARM cores.
- An SoC + Accelerator approach that delivers 2.3× the throughput of a Xeon server and is 75% more energy efficient than an ARM server for an arrival rate of 25,000 queries/s and a deadline of 50 ms.
- An evaluation of text search under deadline constraints that shows that scaling out using servers augmented with accelerators is more than 60% cheaper than scaling out with conventional Xeon chips for high arrival rates and tight deadlines, even for half priced Xeon servers.

The remainder of this paper is organized as follows. Section 2 defines the similarity search problem and its contemporary solutions. We then describe the baseline architecture of our system, and our approach to search on the host and the accelerator in Section 3. Section 4 briefly describes the model we use for computing cluster cost of ownership. Sections 5 and 6 describe our experiments, and evaluate our accelerator based implementation in comparison to the host processors. Related work is discussed in Section 7, and we conclude in Section 8.

2. Similarity Search

Similarity search is a general mechanism that identifies a group of *similar* objects based on some definition of similarity between any two objects in the given space. More formally, we are given a D dimensional space containing N *input points*. We are given a *query point* in this space and wish to find the k ($k << N$) most similar points to this query point from the collection of N points. These points can be objects such as documents or images characterized by some features and represented as vectors in this space. We assume that both N and D are very large, and aim for an exact solution to this problem by exhaustively searching through all input objects.

An essential factor in similarity search performance is the way the input objects are stored and accessed. For many practical problems, N and D can be very large, and an *index* structure is used to *shard* these objects across a cluster. For example, for document retrieval in an application like web search using a bag of words model, N is in the order of billions, and D is in the order of millions (unique words, n-grams). As the number of unique words per document is usually $<< D$, the document vectors are sparse; and an inverted list is used to efficiently store and access the index [5].

In many cases, an approximate solution to the nearest neighbor problem is equally useful [17]. Locality Sensitive Hashing (LSH) is a more recent approach for approximate similarity search [11, 34]. LSH is essentially a clustering technique that partitions the input space into buckets based on a family of hash functions. Approximate similarity search can give higher performance gains at the expense of quality [8], and we leave that exploration to future work. We discuss our work in the context of exhaustive search, but the ideas are broadly applicable and used for approximate search as well.

Many of these methods add significant complexity to index construction and querying, and do not map well to throughput accelerators. Adding a datacenter or cluster perspective significantly constrains the time and power we have to perform the search operation. We assume an incoming stream of *query points* in this space with a constant arrival rate of α and find the $k(k << N)$ nearest neighbors for each of the query points within a deadline of T(seconds). We aim to identify the processor or platform giving the highest throughput in terms of Queries × Documents per second while at the same time looking for higher throughput per Watt. For the purposes of this work, we analyze similarity search in the context of document retrieval, which makes the input and query vectors sparse. The ideas presented in this paper can be easily applied to any similarity search problem with the above properties.

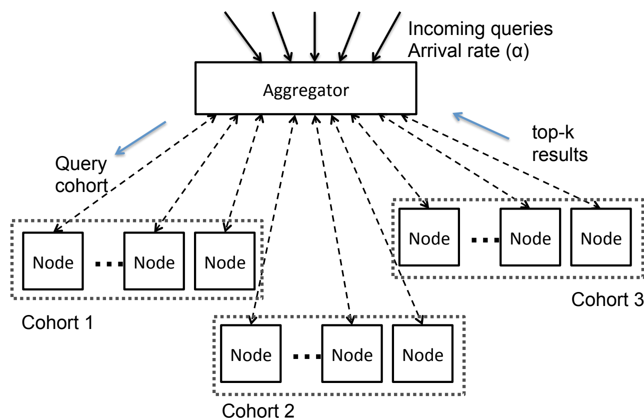


Figure 1: Baseline Search Architecture.

3. Mapping Similarity Search to Different Architectures

In its simplest form, similarity search involves ranking the input documents on the basis of their similarity with a given query. We first look at the baseline architecture of our cluster, and the algorithms we adopt to perform search on the host and the accelerator. We then briefly describe the optimizations we perform on both platforms, in order to fully utilize the capabilities of each architecture.

3.1 Baseline Architecture

We assume a baseline architecture similar to contemporary designs [20], consisting of an aggregator node [1] and multiple service nodes (Figure 1). The aggregator is responsible for parsing the query stream and generating the respective query vectors. A *query vector* is defined as a vector in the given input space for which we wish to compute the nearest neighbors. The aggregator waits for C query vectors to arrive based on the query arrival rate (α), where C is the $cohort\ size$ and this cohort is broadcast to each of the service nodes. A service node holds a *shard* of the input data set, and is responsible for generating the k highest similarity matches for each query in the cohort from its shard. The aggregator receives these matches from each service node, and generates the k highest similarity results from these matches.

A set of service nodes is needed to store all the N documents, and a given cohort runs on this set. A node processes a single cohort at a time across its shard. While a cohort is being serviced, more queries arrive at the aggregator, forming additional query cohorts, and more nodes are needed to service these queries. Our cluster design holds multiple copies of the document collection N, allowing multiple cohorts to be serviced at a time. In this work we focus on the design and capabilities of the service nodes which do most of the work, and assume the aggregator has the capacity to handle the incoming query arrival rate.

[1] We use the terms node and machine synonymously in this work.

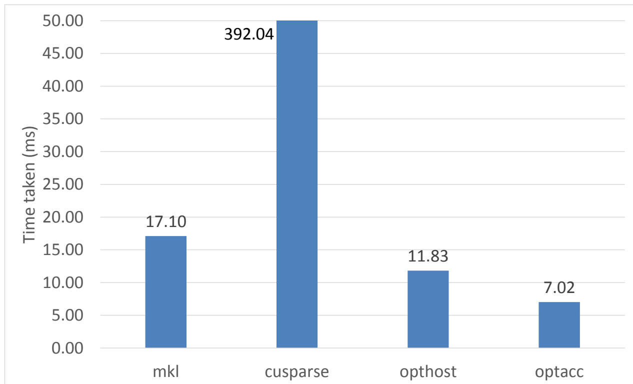


Figure 2: Time taken for an SpMM operation by MKL (16C/32T Xeon) and cuSPARSE (GTX Titan) for $\mu = 500000$, $C = 256$ queries. *opthost* and *optacc* are the times for our algorithm.

3.2 Algorithms and Implementation

Each service node contains a shard of the document collection, based on the node's compute capabilities and the number of documents it can process within the given deadline. A machine holds $n_{machine}$ documents that we wish to search across (similar to a *shard* in a cluster setup), and we batch queries into *cohorts* of size C each. The machine is then responsible for generating the k highest similarity matches for each query in the cohort across these $n_{machine}$ documents.

Let A^T be the $n_{machine} \times D$ matrix composed from $n_{machine}$ input data vectors, and Q be the $C \times D$ query matrix composed from a cohort of query vectors. We assume that both A and Q are sparse and stored in the Compressed Sparse Row (CSR) format and use cosine similarity as a measure of similarity between two documents in the given space [28]. We note that A stored in the CSR format is similar to an inverted list [5], as each row of A represents a term, and holds the list of documents (columns + values) that contain that term. Computing the similarity between the input vectors and a query cohort then becomes a Sparse Matrix Matrix Multiplication (SpMM) operation to produce a $C \times n_{machine}$ similarity matrix S,

$$S = Q \times A \tag{1}$$

such that $S_{i,j}$ is the similarity between query i and document j. We evaluated an implementation of our SpMM approach using commercial BLAS libraries such as Intel's MKL and NVIDIA's cuSPARSE, however, we found their performance to be unsuitable for realtime deadlines. We develop our own implementations for SpMM and Top-k on the host and the accelerator which deliver higher performance than these commercial libraries (Figure 2), and briefly describe them now.

Sparse Matrix Matrix Multiplication Sparse matrices consist primarily of zeros, and are therefore stored in special formats for efficient storage utilization. *Compressed Sparse Row* (CSR) stores a sparse matrix in the form of 3 one-dimensional arrays; the values array which holds all nonzeros, the columns array which holds column indices for these nonzeros, and the rows array which holds indices into the column array for the start of each row. SpMM is a harder problem than dense matrix multiplication, as different sparsity patterns make coalesced memory accesses for both matrices difficult.

Algorithm 1 shows the pseudocode for our approach. The outer loop iterates over all the rows of the query matrix Q in parallel. The middle loop iterates over each nonzero in a particular row of Q. The

Algorithm 1 Sparse Matrix Matrix Multiplication for $S = Q \times A$

```
1:  ▷ clear all values in S
2:  S ← 0
3:  ▷ in parallel
4:  for i ← 1 to C do
5:      for j where Q(i, j) ≠ 0 do
6:          ▷ in parallel on accelerator
7:          for k where A(j, k) ≠ 0 do
8:              S(i, k) ← S(i, k) + Q(i, j) * A(j, k)
9:          end for
10:     end for
11: end for
```

inner loop accumulates the rows of A corresponding to nonzeros in the current row of Q in parallel on the accelerator.

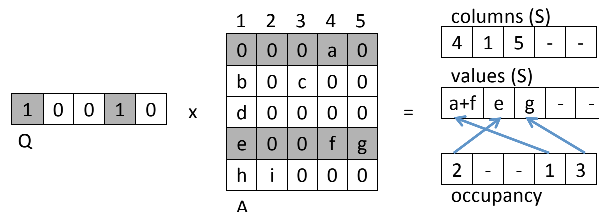


Figure 3: Our SpMM approach for S = Q x A. The shaded rows of A are accumulated into the values(S).

We merge rows of A corresponding to nonzeros in a row of Q into a row of S. S is stored in a sparse format similar to CSR, but the columns are not sorted as our final goal is to compute the top-k results. Accumulating values from rows of A is usually done by using a hash table to map columns of S to indexes into the values array of S. However, standard hash tables can become arbitrarily complex, due to lack of size guarantees and handling of collisions. A key aspect of our approach is the use of a $C \times n_{machine}$ *occupancy array* to keep track of the location of each nonzero of S in the values array. The occupancy array maps a column index in A to an index in the values array of S. For every nonzero in a row of A, the occupancy array is checked to see if that column index already exists in the values array (*accumulation*) or it needs to be appended (*insertion*). For an *insertion*, the occupancy array is updated to reflect the location of the new inserted value in the values array, essentially creating a perfect hash function at runtime. The columns array of S is updated with the column index of the corresponding value in the values array.

Figure 3 shows a toy example to demonstrate our approach. When the first nonzero of Q is processed, a is *inserted* into the values array, and occupancy(4) is updated to the position of a in the values array. When the next nonzero of Q is processed, e and g are *insertions*, and f is an *accumulation* into values[occupancy[4]]. Our final goal is to compute the highest k results for each row of the similarity matrix, therefore, we do not sort the columns array. This improves overall time taken by the algorithm, and makes the output arrays from our SpMM implementation different from the traditional CSR format.

Top-k We compute the top-k results in parallel for a query cohort. Our top-k algorithm relies on maintaining a small per query *cache* of the current top-k results while traversing the list of similarity scores. The cache is a list of the current k highest similarity scores stored in memory. The value under inspection is inserted into the cache if it is greater than the current minimum in the cache. The number of insertions into the cache reduces as we iterate through

the list and the cache gains a better idea of the actual results. For $k << n_{machine}$, we found that the top-k latency is primarily governed by the time to load and inspect the values from memory rather than the insertions, and we use a simple butterfly reduction to calculate the minimum in parallel on the accelerator rather than a binary heap.

We emphasize that our SpMM algorithm is designed specifically for text search, in that we do not output the S matrix, it lives entirely inside fast scratchpad storage. After we extract the top k values from each sub-matrix, S is discarded, making our approach dissimilar to existing SpMM implementations.

3.3 Architecture specific modifications

We performed a large number of optimizations for our SpMM and top-k implementations by analyzing program performance using VTune on the host, and the NVIDIA Visual Profiler on the accelerator. We use the well known technique of tiling to divide the $n_{machine}$ documents or shard into smaller tiles of size μ each, performing SpMM and top-k operations for each tile and merging the Top-k results into a set of k most similar documents.

Tiling the input set is more efficient because it improves cache locality and decreases cache contention on the host across a cohort of queries, as the working set can fit in the LLC. It enables us to work with limited memory on the accelerator and allows for early termination of the search, since remaining tiles can be ignored under latency constraints to give partial results. We now focus on the broad changes adopted for each device.

3.3.1 Host specific optimizations

We partition a query cohort uniformly across the various cores on the host multiprocessor, with all cores working on the same tile but for different queries. Processing the same tile across all cores allows smaller tile sizes to fit into the last level cache shared across cores, improving locality for a cohort. Processing different queries on different cores also avoids the need for atomics, as there is no shared state. Within a core, a group of queries is processed sequentially, essentially performing a Sparse Vector Matrix (SpVM) multiplication operation on a tile. For each query, the SpVM operation is followed by a top-k computation, taking advantage of cache locality for the value and column arrays of S.

We perform experiments for a different number of threads for each cohort size. Larger number of threads per tile allows for higher instruction level and memory level parallelism, however, more threads mean more queries in flight and more state, which creates cache contention. Memory bandwidth on the host is limited, and increasing ILP beyond that decreases performance.

For evaluating single queries (or $C = 1$), we enable parallelism by processing different tiles across different cores. This improves memory bandwidth utilization at the expense of cache pollution, giving better performance than a single threaded implementation. We also use the Intel ICC compiler for auto-vectorization.

3.3.2 Accelerator specific optimizations

Given the large number of compute cores on the accelerator, we modify our algorithm to exploit both inter-query and intra-query parallelism. An easy way to exploit intra query parallelism would be to partition the input tile into smaller tiles (*microtiles*) based on document indexes, and launching a thread per query per microtile. The columns array is stored in the compressed sparse format, making this partitioning hard as we do not know where each document starts and each ends in a particular tile.

An obvious choice would be to just decrease the tile size to the microtile size and perform the SpMM operation. However, transferring these arrays over the PCIE bus is not feasible for our deadlines and we need to store the rows array for each tile in accelerator memory, which is of size $4D$ bytes. For $\mu = 1,000,000$ and a microtile size of 1,000, we would need 1,000 microtiles, increasing the overhead of the rows array to $4,000D$ bytes. D can be in millions, it can be the number of n-grams or unique words for text search, requiring several gigabytes just for the rows array. For large D, this becomes prohibitive.

Figure 4 shows our approach on the accelerator. A static partitioning is not feasible due to space constraints and for each query, we dynamically generate microtiles for rows of A indexed by the query terms. This partitioning can be done on the host or the accelerator. If done on the host, the partitioning information would have to be copied over to the accelerator, if done on the accelerator, the partition function needs to be parallelized. To solve this, we use a multi-step partitioning approach. The first partitioning operation is done on the host to generate fewer and larger *subtiles* to enable parallelism for the second step on the accelerator. The second partition operation runs on the accelerator, and generates the final microtiles in accelerator memory.

To summarize our partitioning, we divide a *shard* into multiple *tiles*, and divide a tile into multiple *subtiles* on the host. Each subtile is partitioned into smaller *microtiles* on the accelerator. An SpMM operation is performed for each query cohort and microtile, giving us the result values and columns arrays. The Top-k kernel then computes the final k values and columns for each query in the query cohort for a given tile.

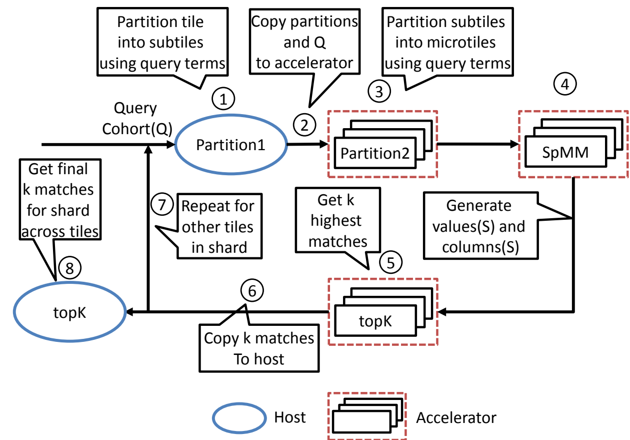


Figure 4: Implementation of similarity search on the accelerator

We divide an input tile into 10 subtiles, and choose a microtile size of 1000 documents to allow the occupancy array to fit into the smaller higher bandwidth scratchpad memory of the accelerator (similar to a self-managed cache). Multiple threads work on the same query and microtile at a time, so accumulations into values(S) and updates to the occupancy array are done atomically. Each microtile based SpMM operation generates its part of the values(S) and columns(S) array. We compute top-k results for each query across a tile, then merge these results and perform a final top-k operation across tiles to get k matches for a shard. We optimize our top-k implementation as well, by first performing top-k across subsets of the values(S) array, and then merging these results.

Computation is primarily performed on the accelerator, and no data is transferred across the PCIe bus except for the query cohort, the partition information at step 2 and the top-k results since an entire shard resides in GPU memory. Each of these data structures is a maximum of several MB in size, allowing us to avoid PCIe bandwidth constraints under tight deadlines. The host can also work on its tiles in parallel with the accelerator, and perform a final

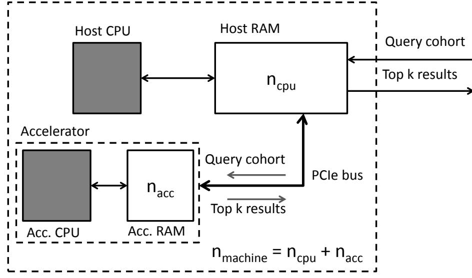


Figure 5: Block diagram of a node in our cluster design.

top-k computation between the top-k values generated from the host and those from the accelerator. These optimizations allow both the host and the accelerator to optimally utilize their underlying architectures, and each device can process millions of documents in sub-millisecond times for a query.

4. A Model for Cost Efficient Cluster Design

This work presents a framework to implement high dimensional similarity search across billions of points in a scalable and cost efficient manner. We propose a model that can be used to estimate the cost of performing similarity search across a stream of queries for a given server platform. The model computes the capital cost and operational cost of a system to run a workload to satisfy a given arrival rate (α) and deadline (T).

To calculate the cost of performing the search for a given α and T, we estimate two metrics: 1) the number of machines required, and 2) the energy consumed by these machines over their lifetime. The first parameter is used to estimate the total capital cost based on the cost of each machine, and the second parameter gives us the total operational/electricity cost of running the workload. The total cost of ownership ($\$_{total}$) is

$$\$_{total} = \$_{capital} + \$_{operational} \tag{2}$$

Given that both D and N are very large, and the embarrassingly parallel nature of similarity search, we divide the problem into shards across a cluster. We batch the incoming query stream into *cohorts* of size C each. Assuming a homogeneous distribution of machines in the cluster, we allocate an equal number of documents to each machine. We model an exhaustive search, thus each query is processed across all documents. We use \$ to refer to cost and n to refer to the number of documents. For example, $n_{machine}$ refers to the number of documents processed on a machine. The number of machines needed to process a query cohort is

$$machinesPerCohort = \frac{N}{n_{machine}} \tag{3}$$

The *cohort arrival rate* is $\frac{\alpha}{C}$. The number of cohorts that arrive while a cohort is being serviced for a *cohort service time* of T is

$$cohorts = \frac{\alpha}{C} \times T \tag{4}$$

To meet deadlines for all queries, all these cohorts need to be processed. Using Equation 3, the capital cost to process all pending cohorts is

$$\$_{capital} = cohorts \times N \times \frac{\$_{machine}}{n_{machine}} \tag{5}$$

where $\$_{machine}$ is the cost per machine. Each machine consists of a host processor and an accelerator (Figure 5). Breaking down the cost per machine and the documents per machine, (*cpu* refers to the general purpose cores in the machine, *acc* refers to the accelerator), the capital cost for a given configuration can be written as,

$$\$_{capital} = \frac{\alpha NT}{C} \times \frac{\$_{fixed} + \$_{cpu} + \$_{acc}}{n_{cpu} + n_{acc}} \tag{6}$$

where $\frac{\alpha NT}{C}$ is the total number of documents that need to be searched for all cohorts arriving in time T, $\$_{fixed}$ is the fixed cost per machine (including disk, RAM, etc.), $\$_{cpu}$ is the cost of the host processor, and $\$_{acc}$ is the cost of the accelerator.

Due to the homogeneous nature of the cluster, the energy consumed to process a cohort can simply be written as,

$$energyPerCohort = machinesPerCohort \times T \times (P_{fixed} + U_{cpu}P_{cpu} + U_{acc}P_{acc}) \tag{7}$$

where P_{fixed} is the fixed or idle power consumption of a machine, P_{cpu} is the power consumed by the host to process n_{cpu} documents, and P_{acc} is the power consumed by the accelerator to process n_{acc} documents. U_{cpu} and U_{acc} denote the fraction of the deadline when the processor is doing work, or the utilization factors of the host and the accelerator respectively. The time a processor is actually performing the search may be less than the deadline T. n_{acc} is limited by the memory on the accelerator, and the if the deadline is higher than the time it takes to process n_{acc} documents, the accelerator remains idle. Both the host and accelerator also remain idle while waiting for the cohort to form. The operational cost of the workload across the lifetime of the machines becomes

$$\$_{operational} = lifetimeCohorts \times energyPerCohort \times energyCost \tag{8}$$

where $lifetimeCohorts$ is the number of cohorts processed over the lifetime of a machine. Assuming the average lifetime of a machine as 4 years, for an arrival rate of α, the number of cohorts processed in 4 years is $\frac{\alpha}{C} \times 4 \times 365 \times 24 \times 3600$. The U.S. industrial average electricity rate is 6.7 cents/kWh [3]. Using Equations 3, 7 and 8, the total operational cost of the workload is,

$$\$_{operational} = \frac{\alpha NT}{C} \times 2.35 \times \frac{P_{fixed} + U_{cpu}P_{cpu} + U_{acc}P_{acc}}{n_{cpu} + n_{acc}} \tag{9}$$

All of the parameters in Equations 6 and 9 can be easily measured for a given server platform, with or without an accelerator (set corresponding accelerator values to 0). The capital cost for a given configuration is essentially the *dollars/workDone* (inverse of the performance per dollar), and the operational cost is the *Watts/workDone* (inverse of the performance per watt) metric. We define the work done per machine as the number of queries processed on a machine across its shard within the given deadline, or,

$$W = C \times (n_{cpu} + n_{acc}) \tag{10}$$

We also define,

$$\$_{system} = \$_{fixed} + \$_{cpu} \tag{11}$$

and,

$$P_{system} = P_{fixed} + U_{cpu}P_{cpu} \tag{12}$$

Using these parameters, the total cost incurred can be written as,

$$\$_{total} = \frac{\alpha NT}{W}(\$_{system} + \$_{acc} + 2.35(P_{system} + U_{acc}P_{acc}) \tag{13}$$

This cost can be easily broken down and understood. By application of Little's Law, αNT is the total amount of work that needs to be done, therefore, $\frac{\alpha NT}{W}$ is the number of machines needed. $\$_{system} + \$_{acc}$ is the cost per machine. 2.35 is the dollars per watt spent over the lifetime of a machine. $P_{system} + U_{acc}P_{acc}$ is the

power consumed per machine. This model allows us to compute of the cost of ownership of a similarity search platform for a given arrival rate, deadline, corpus size and hardware configuration. The model also allows us to compare different cluster designs. For a given arrival rate and deadline, designating the two configurations as 1 and 2, configuration 1 is cheaper than configuration 2 if,

$$\frac{\$_{total}(1)}{\$_{total}(2)} < 1 \tag{14}$$

We pick the highest Work done for each configuration, and this inequality gives us a linear design space that can be explored for different values of the cost of each system. We can compare two CPU configurations, a CPU configuration with an CPU + Accelerator configuration, or a CPU system with an accelerator only system, for different arrival rates and service times.

5. Methodology

Table 1 shows the different platforms we use for evaluation. The PowerEdge R720 represents a state of the art enterprise class server, and the Titan and Maxwell represent different points in the accelerator design space. The GTX Titan is based on the Kepler architecture, and represents a power hungry high throughput accelerator. The GTX 750Ti is based on the more energy efficient Maxwell architecture, and represents a low cost accelerator. We use the quad-core Tegra K1 based on the ARM A15 architecture, representative of high energy efficiency. The Xeon + Titan and Xeon + Maxwell represent our augmented configurations.

We use the English Wikipedia consisting of 4M pages as our input document set. We identify distinct words containing 3 or more letters from across the entire set giving around 3.5 million unique words/dimensions. We use the popular *tf-idf* method to score documents [28], which reflects the importance/weight of a word in a given document. There are many different techniques for document scoring, e.g. PageRank [5] for web search, color histograms for images [32], and other machine learning based ranking models. The scoring methodology is used to populate the document and query matrices used in our algorithms. We assume the document and query matrices are given to us, and our results are independent of the scoring methodology used.

We perform our experiments for a collection of N = 1 billion documents. To simulate this collection, we assume multiple copies of Wikipedia, specifically 250 copies of 4M pages each. We use boolean values for the query vectors, denoting presence (1) or absence (0) of a term. Due to a lack of a public database of queries, we generate queries by randomly picking titles of Wikipedia pages. Correctness and quality is guaranteed by ensuring the results from the search contain the page matching the respective title. We use single precision floating point for our computations for all platforms.

We implement C versions of our SpMM and Top-k algorithms for the x86 platforms and C+CUDA versions for the GPU platforms. The x86 code is compiled using the Intel Compiler Collection (ICC) 15.0.0 with vectorization and -Ofast enabled. The GPU code is compiled with gcc + CUDA 6.5 with -O3. We measure real values for power using a Watts up? PRO meter.

A global scale search engine like Google processes over 100 billion search queries a month, or 38K search queries/second. Assuming these are distributed to 20+ datacenters around the world, the maximum arrival rate of queries at a location can be estimated as 1000 queries/second. We evaluate our search platform for an arrival rate (α) of 1000 queries/s modeling present datacenters. We also evaluate a high arrival rate of 25000 queries/s to demonstrate the scalability of our approach, and to model future datacenters. We constrain our deadline to 50 ms, leaving time for other aspects of search such as parsing, network overhead and ranking.

We vary the tile size μ from 100K documents to 2M documents by randomly partitioning the Wikipedia document set. The cohort size (C) varies from 1 to 1024 for each value of μ, as 1024 is the largest cohort size feasible for an arrival rate of 25000 queries/s and a deadline of 50 ms. For a cohort size of 1, we run experiments for 1K queries, allowing the experiments to finish in a reasonable amount of time. For larger cohort sizes, we evaluate 8K random queries. We use a value of $k = 32$ for our top-k computations. For our experiments with the Xeon processor, we vary the number of threads from 1 to 128, and pick the number of threads giving the highest throughput. For the ARM experiments, we vary the number of threads from 1 to 8.

We note that our design space is essentially 7 dimensional, where each value of $(platform, \alpha, T, C, \mu, threads)$ represents a different design point and cost value. We evaluate a platform on the basis of its most cost effective configuration. We run experiments for each value of these parameters and pick the value of μ, C and $threads$ which gives the highest Work done (W) and reduce the design space to 4 dimensions $(platform, \alpha, T, W)$ for our analysis.

We model a higher memory variant of the GTX 750Ti, and increase its memory to 6 GB, while retaining existing compute capabilities (5 SMs for the Maxwell). We assume that the DRAM accesses and signaling consume 20% of the accelerator's power [21]. A 6 GB Titan consumes 150 W of power for our workload, and we estimate that 6GB of GDDR5 memory consumes 30W of dynamic power. We add 30W to the Maxwell's power consumption to model this variant.

To take advantage of the accelerators cost efficiency, we model an *accelerator only* cluster design where the primary function of the host processor is to drive the accelerator, and search is performed only on the accelerator. This allows us to significantly reduce the cost of the host platform. We choose the J1800 SoC based on the Bay Trail-D processor. We refer to these *accelerator only* configurations as *SoC + Titan* and *SoC + Maxwell* in our experiments.

5.1 Computing n_{cpu} and n_{acc}

The number of documents that can be processed on the host or the accelerator for a cohort is bounded by the deadline T. The available service time is further reduced by the time it takes to form a cohort. For an arrival rate of α and a cohort size of C, the available processing time is $T' = T - \frac{C}{\alpha}$, since the first query in the cohort must finish before the deadline. We determine the number of documents that can be processed on the host or the accelerator within this time so that at least 95% of queries satisfy the deadline by using a Monte Carlo simulation for the tiles processed per machine.

We assume the host has a large enough memory pool such that it can accommodate the maximum number of documents that can be processed within the available deadline. For the accelerator, we bound the number of documents that can be processed by the available memory on the accelerator, to avoid PCIe bottlenecks.

6. Evaluation

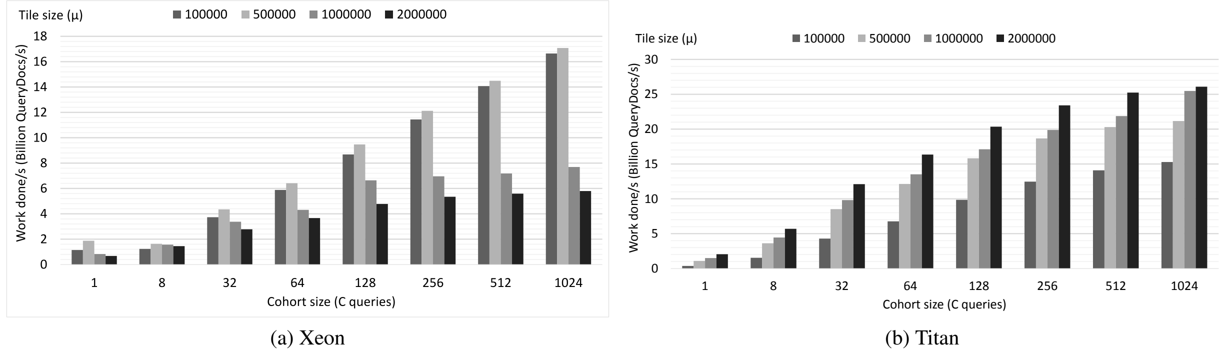
This section evaluates conventional hardware platforms and compares them to accelerator based platforms. We present some preliminary results that confirm the applicability of commercial accelerators to similarity search, and show that these provide much more efficient performance than conventional processors even under very tight deadlines.

6.1 Impact of the Tile Size μ and Cohort Size C

A simple measure of the work done in similarity search can be the number of queries a device can process over a collection of documents. Performance of a configuration becomes the

Table 1: Experimental and Modeled(*) System Platforms

Platform	Description
Xeon	PowerEdge R720, 2 x Xeon E5-2650v2, 22 nm, 16C/32T, 8x8GB 1866MHz RDIMMs
ARM	Tegra K1 SoC, 28 nm, Jetson TK1, ARM A15, 4 cores, 2GB DDR3L RAM
Xeon + Titan	PowerEdge R720, 2 x Xeon E5-2650v2, GTX Titan, 28 nm, 2688 CUDA cores, 14 SMs, 6GB GDDR5 Memory
Xeon + Maxwell	PowerEdge R720, 2 x Xeon E5-2650v2, GTX 750Ti, 28 nm, 640 CUDA cores, 5 SMs, 6GB GDDR5 Memory
SoC + Titan*	J1800 SoC, 22 nm, 4GB DDR3L RAM, GTX Titan
SoC + Maxwell*	J1800 SoC, 22 nm, 4GB DDR3L RAM, GTX 750Ti (6GB variant)



(a) Xeon

(b) Titan

Figure 6: Performance of a processor for different cohort sizes and tile sizes.

Work done/second, and the energy efficiency becomes the Work done/Joule. Figure 6a and 6b show the performance for the Xeon and the Titan as a function of the tile size μ and the cohort size C. Focusing on the Xeon, we can see that larger cohort sizes give higher performance, since more queries can work on the same tile at a given time, which fits in the last level cache. On the other hand, larger tile sizes don't give the highest performance. Even if the tile is shared across queries, each query maintains its own occupancy, values and columns array, and large tile sizes result in a degenerate cache access pattern, causing misses in the top-k segment.

For the Titan, both larger cohort sizes and larger tile sizes result in higher performance. The partition function needs to be run for each tile, and larger tile sizes create fewer overall tiles, reducing partitioning overheads. Larger cohort sizes enable more memory level parallelism and higher utilization of the accelerator.

For the rest of our analysis, we pick the tile size giving the highest performance/throughput for a given processor at a given cohort size. The power consumption of a platform remains relatively uniform across different cohort sizes and tile sizes, therefore higher throughput gives both higher Work done/s and Work done/J.

6.2 Peak Performance and Efficiency

Similarity search involves processing a set of queries over the given documents using the most efficient processor available. Batching queries together is intuitively a good idea, as it increases ILP, and allows queries to share resources amortizing fixed costs. We first determine the raw throughput and efficiency of the host and accelerator processors. This is shown in Figure 7a and 7b for different cohort sizes. Even for single queries ($C = 1$), the accelerators outperform the Xeon, with the Maxwell delivering 1.5$\times$ more performance than the Xeon and being 3$\times$ more energy efficient. The Titan remains underutilized for single queries, and its lower clock speeds compared to the Maxwell reduce performance, delivering 9% more performance than the Xeon, at 21% more energy efficieny.

For larger cohort sizes, the Titan delivers more performance, while the Maxwell is more energy efficient. For a cohort size of 64 (Figure 7b), the Titan delivers more than 2.5$\times$ the performance of the Xeon, while the Maxwell delivers more than 3.5$\times$ the work per Joule. This can be partially attributed to the more energy efficient Maxwell architecture in comparison to Kepler. The Maxwell card also provides more available scratchpad memory per core, resulting in more parallelism for our algorithm. However, the Maxwell saturates in terms of both throughput and efficiency for cohort sizes larger than 64. This is expected, as our variant of the Maxwell has just 5 SMs. The Titan becomes more energy efficient than the Maxwell for cohort sizes larger than 256, as there is enough instruction and memory level parallelism to fully utilize the accelerator.

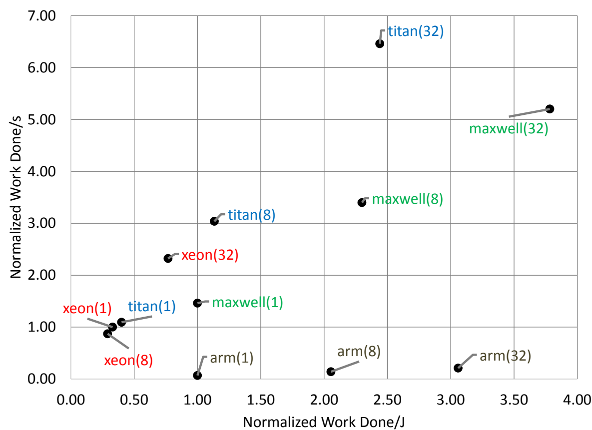
The quad-core ARM is more energy efficient than the Xeon for all cohort sizes, however, its throughput is significantly lesser, delivering 7% of the throughput of the Xeon at 3$\times$ the efficiency for a cohort size of 64. We can see that current accelerators are a better choice than the Xeon and ARM in terms of both performance and efficiency, providing ample motivation to consider accelerators for more efficient cluster configurations. This analysis ignores the memory limitations of the accelerator and the delay incurred while forming a cohort, and looks at the peak performance and efficiency figures for each platform. We next look at a practical implementation with a stream of incoming queries and constrained deadlines.

6.3 Adding Constraints

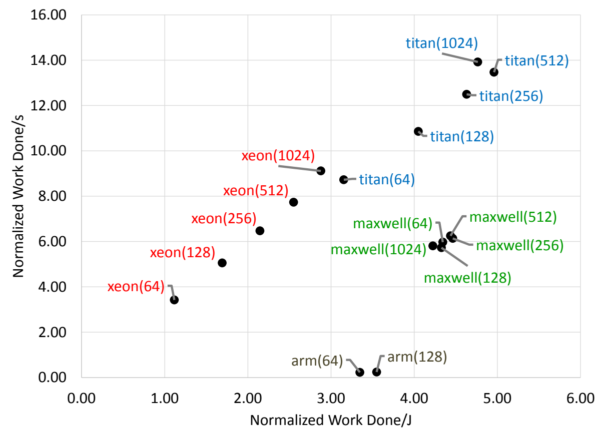
We now model a more realistic search platform by adding several constraints:

- The search operation must finish within T ms, and 95% of queries finish within this time.
- The number of tiles processed on the accelerator is limited by available memory as PCI-E transfers are infeasible within these deadlines.
- Queries arrive with a rate of α queries/s, therefore forming a cohort incurs batching delay, and the cohort must still finish within T ms.

Figure 8a shows the throughput and efficiency of the host and accelerator platforms for an arrival rate of 1000 queries/s and a deadline of 50 ms. Larger cohorts require more time to create,

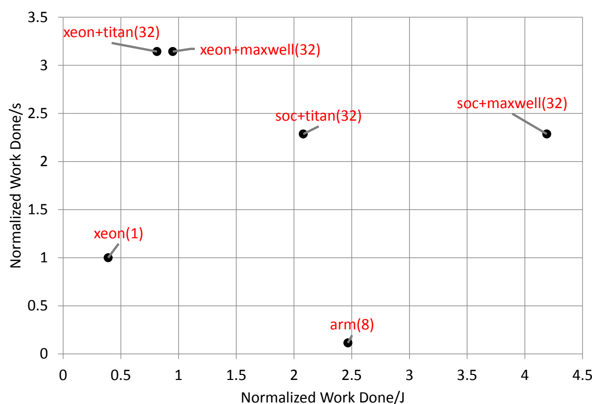


(a) C = 1-32

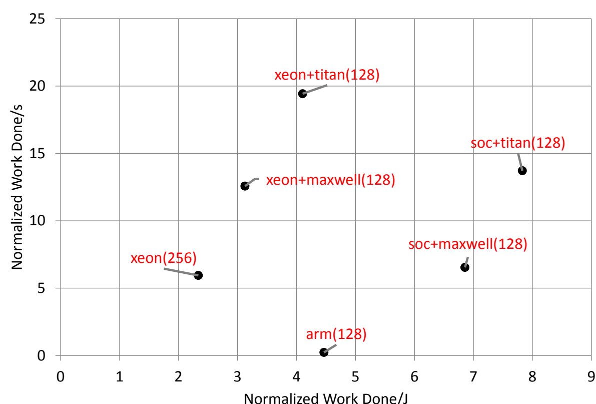


(b) C = 64-1024

Figure 7: Throughput-efficiency of text search for different processors for different cohort sizes (in brackets). The throughput (y-axis) has been normalized to Xeon(1) and efficiency (x-axis) have been normalized to arm(1).



(a) α = 1000 queries/s



(b) α = 25000 queries/s

Figure 8: Optimal platforms for text search for a deadline of 50 ms. The numbers in brackets are cohort sizes. The throughput (y-axis) has been normalized to Xeon(1) and efficiency (x-axis) have been normalized to arm(1).

and cohort sizes larger than 32 become infeasible in 50 ms. The Maxwell based accelerator only configuration (soc+maxwell) is the most efficient at a cohort size of 32, doing 70% more Work/J, while delivering 2.2× of the throughput of the Xeon. The augmented accelerator based configurations deliver the highest throughput, which is expected as each machine carries two processors.

The accelerators have 6 GB of memory, and can hold a maximum of 8M documents. For a cohort size of 1, they process these 8M documents in < 5 ms, and remain idle for 45 ms, leading to low performance and energy efficiency. Due to memory capacity constraints, accelerators opt for the highest cohort size to maximize their Work done. For a cohort size of 32, the soc+accelerator platforms remain idle for 32 ms while batching the requests, yet do more than 2× the work of the Xeon, while being more energy efficient than the ARM.

The host platforms are not bound by memory limitations, and both the Xeon and the ARM processors achieve their highest throughput and efficiency for smaller cohort sizes, as they have more time to satisfy 95% of the queries giving them more flexibility in tile management. For a cohort size of 32, they have to wait 32 ms for the cohort to form, and have only 18 ms to process documents. Each tile takes longer to process as well for larger cohort sizes, and even though larger cohort sizes are more efficient (Figure 7), they might not be the optimal strategy under tight deadlines.

Intuitively, increasing the arrival rate of queries (lower cohort formation times and higher cohort sizes) should increase both throughput and efficiency for all the configurations. This can be confirmed from Figure 8b, which shows the throughput and efficiency of our evaluated platforms for an arrival rate of 25000 queries/s. Both the accelerator only configurations deliver higher throughput and are more energy efficient than the ARM and Xeon designs. The SoC + Titan approach delivers 2.3× the throughput of the Xeon while being 75% more energy efficient than the ARM.

Tables 2a and 2b show the optimal parameters and results of our model for different experimental configurations. Each set of listed values gives the highest throughput for that particular cluster platform. The number of machines represent the machines needed to process a billion documents for the given arrival rate such that

Platform	C	μ (Kdocs)		Power (Watts)			n (Kdocs)		No. of machines	Total Cluster Power (kW)
		cpu	acc	fixed	cpu	acc	cpu	acc		
(a) α = 1000 queries/s										
Xeon	1	500	-	144	33	-	56000	-	893	146
Arm	8	100	-	2	2	-	800	-	7813	23
Xeon+Maxwell	32	100	2000	156	32	68	1500	4000	285	52
Xeon+Titan	32	100	1000	193	32	62	1500	4000	285	62
SoC+Maxwell	32	-	2000	17	-	68	-	4000	391	14
SoC+Titan	32	-	1000	54	-	62	-	4000	391	27
(b) α = 25000 queries/s										
Xeon	256	100	-	144	32	-	1300	-	3757	611
Arm	128	100	-	2	2	-	100	-	97657	320
Xeon+Maxwell	128	100	500	156	41	65	2500	3000	1776	419
Xeon+Titan	128	100	2000	193	41	77	2500	6000	1149	323
SoC+Maxwell	128	-	500	17	-	65	-	3000	3256	218
SoC+Titan	128	-	2000	54	-	77	-	6000	1628	183

Table 2: Configurations for $T = 50$ ms. Each configuration is optimal for that platform giving highest Work done/s.

95% of queries finish within 50 ms. The accelerator augment based configurations require less than half the machines to satisfy the given arrival rate and deadline compared to a Xeon only cluster.

The ARM based cluster requires almost 100K quad-core processors to handle an arrival rate of 25000 queries/s, making it impractical due to scale-out/uncore overheads. Although accelerators require additional power per machine, the reduction in number of machines reduces overall fixed costs, reducing total power required.

Accelerators do more work than the host processors for the same arrival rate and deadline constraints, translating into higher throughput and higher throughput/Watt. A natural question for any datacenter architect would be if these improvements translate into actual cost savings, or if the cost of the accelerator becomes prohibitive. We next look at the cost of different cluster configurations based on the host processor and accelerator used.

6.4 Cost of ownership Analysis

We now compare the different platforms based on cost of ownership from models created in Section 4. We use values of power, utilization and work done from our experiments, and identify optimal configurations for each platform type based on highest throughput. Identifying the cheapest configuration is essential for minimizing total cost of ownership.

We pessimistically assume retail/sticker prices for the ARM system and the accelerators. We use a value of $\$_{system}(arm) = \$100$, and $\$_{acc} = \$1000$ for the Titan, and \$300 for the 6GB Maxwell. Lower values for these prices would give higher cost gains for the accelerator based designs. Figure 9a shows the ratio of the cost of ownership for different configurations normalized to the cost of the Xeon platform, for values of $\$_{system}(xeon)$ ranging from \$500 to \$6000 (the retail price for a PowerEdge R720) for an arrival rate of 1000 queries/s. Even for a half priced Xeon system, the ARM cluster is 73% cheaper, and the SoC + Maxwell configuration is more than 15× cheaper than the Xeon cluster.

If a 16 cores/32 threads Xeon system can be acquired for $1/12th$ its retail price at \$500 (including fixed costs), both accelerator configurations are still significantly cheaper, with the Xeon + Maxwell being 56% cheaper, and the SoC + Maxwell more than 3× cheaper. The cost benefits of the Xeon + Maxwell system remain constant at higher values of $\$_{system}(xeon)$, as the ratio of total costs approaches the ratio of the work done by the two systems.

Increasing the arrival rate to 25000 queries/s (Figure 9b) makes the Xeon more efficient due to larger cohort sizes, and the Xeon cluster is cheaper than the ARM one for $\$_{system}(xeon) < \$2400$. We choose the Titan as the accelerator at these arrival rates due to its higher throughput and efficiency. At half the retail price of the Xeon, the SoC + Titan system is more than 5× cheaper, and the Xeon + Titan configuration is 2× cheaper. Even for $\$_{system}(xeon) = \$500$, the accelerator platforms are 30% cheaper than the Xeon cluster. For capital and operation costs running into millions of dollars, these are still significant savings.

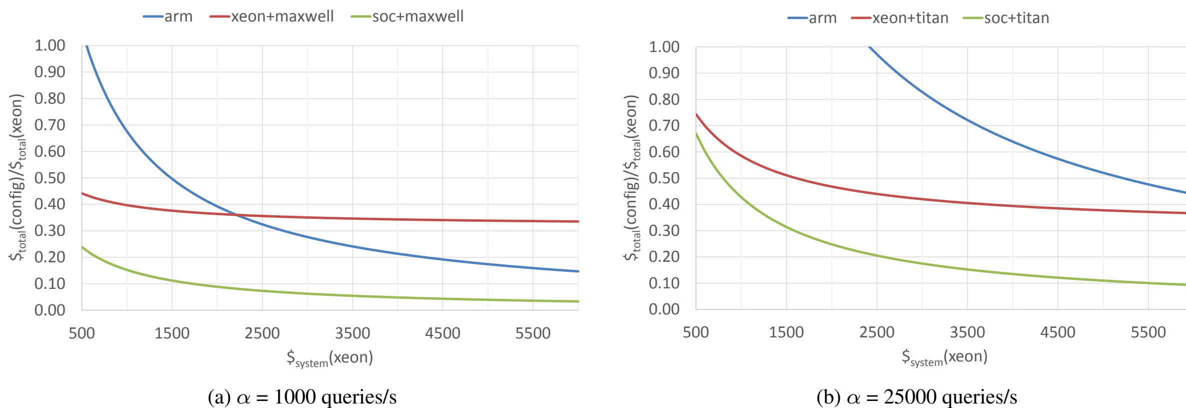
This analysis fixes the price of the accelerators and looks at the gains in the total cost of ownership across a range of Xeon cost values. These results show significant margins for the accelerators, even with present sticker prices for accelerators and cheap Xeons, providing ample proof for the applicability of accelerators to text search. Both the Titan and the Maxwell are manufactured at 28 nm, in comparison to the Xeon which is at 22 nm, providing even more potential for these cards in the future.

6.5 Miscellaneous

Network Bandwidth in Scale-Out Due to its embarrassingly parallel nature, text search is not a network intensive workload. The only data transferred over the network is the query cohort from the aggregator to the service nodes, and the final top-k results from the service nodes back to the aggregator. For an application like web search, assuming an average of 2.4 terms per query [33], requires (4B for term weight + 4B for term identifier) × 2.4 = 20B per query. Assuming a node receives a cohort of size 256 queries every 50 ms, processes its shard, and sends out the top 32 results (values + document ids) per cohort, the total bandwidth requirement at a node is $\frac{(256\times20)+(256\times32\times8)}{0.05}$, or 11 Mbps, which is ∼1% of a standard Gbps link. Using accelerators also reduces the overall number of machines required (Table 2), further reducing network overhead by requiring fewer connections and switches.

Changing the Arrival rate and Deadline Our conclusions would not change with variations in the arrival rate. Changing the arrival rate translates to changing the cohort size, and even at the smallest cohort size of 1, the Maxwell card is as efficient as the Xeon and ARM processors (Figure 7). However, the 6GB memory limit of the GPU prevents it from attaining that efficiency, and higher memory GPUs would be desirable. For higher arrival rates, larger cohort sizes become feasible, and we have already demonstrated the applicability of accelerators to those.

The primary bottleneck for longer deadlines would be the memory and PCIE limitations of current GPUs. Due to low PCIE bandwidth, it is not possible to transfer documents over to GPU memory, and the GPU will sit idle for the remainder of the time, leading to lower efficiencies. For higher arrival rates, it can compensate by forming larger cohorts within the deadline. Therefore, accelerators are not a good choice at lower arrival rates and longer deadlines, but in that case the query can be easily processed on the host processor.



(a) α = 1000 queries/s

(b) α = 25000 queries/s

Figure 9: Ratio of cost of ownership for different platforms as a function of cost of the Xeon system for T = 50 ms

6.6 Limitations

Indexing and Query Parsing We assume that the document index and the query matrices are given to us, and do not consider the cost of constructing these. We set our deadlines to 50 ms, and a web search takes anywhere from 200 - 300 ms for fluid response times. We assume the remaining time as sufficient to do parsing and ranking. There is a significant body of work in constructing the index offline, and we believe this to be complementary to our work.

Scale out overhead We do not consider additional overhead while scaling out, such as the network and fixed hardware, cooling costs and PUE factors. Adding accelerators to machines would increase cooling costs per machine, however, it requires fewer machines to do the same task. These overheads can also be folded into the fixed costs per machine, and higher system costs further amplify the benefits of adding accelerators to existing hardware.

Additional workloads Our work identifies the most efficient hardware platform for similarity search. A datacenter server runs multiple workloads, and these might result in conflicting platform designs. There is a growing body of work optimizing more and more applications for accelerators, and these devices can be utilized for other workloads as well.

Corpus size We use the English Wikipedia for our experiments, which consists of just 4M pages. The number of documents indexed by industrial search engines is many orders of magnitude larger than that. However, our study only uses Wikipedia as a reference for scaling out to a larger document collection with similar attributes as Wikipedia. Wikipedia is sufficiently dense, averaging to 100 nonzeros per row of the document term matrix. A larger corpus consisting of random web pages would contain fewer distinct words than Wikipedia, speeding up the SpMM and top-k operations even more.

7. Related Work

This work spans several research areas in a large number of topics namely similarity search, k Nearest Neighbors, datacenters, total cost of ownership models, GPGPU implementations, Internet Search Engines and Sparse Matrices. In the interest of brevity, we focus on research most closely related to our work.

Total cost of ownership models for datacenters exist in many different forms, focusing on the relationship between capital cost and operational cost [3, 23]. Sparse matrices have been used for text search before by Goharian et al. [13]. Their work focuses on using Sparse Matrix Vector Multiplication to process one query at a time, and on the storage benefits of a sparse matrix approach as compared to a more traditional inverted index.

Both the MKL and cuSPARSE libraries are closed-source, making it difficult to algorithmically compare our approach to theirs. We note that the transpose of the document-term matrix (A^T) is very similar to the inverted index when stored in the CSR format, and a search on all documents becomes a SpMM operation. We reiterate that our algorithm works better than these commercial libraries for text based similarity search, and leave a more generic evaluation to future work.

Dalton et al. [6] present the ESC algorithm for GPUs, which breaks down the SpMM operation into parallelizable primitives, however, the algorithm requires significant intermediate state, and several passes over this data, making it computationally expensive. Liu et al [26] focus on irregular sparse matrices, and allocation schemes for the output matrix S. In contrast, we maintain the matrix S entirely in temporary scratchpad storage, and only output the top k results for each row.

Patwary et al. [30] present a similar algorithm that uses a dense matrix to store the results of SpMM. They focus on improving the performance of SpMM on Xeon multicores up to 28 threads. We demonstrate scalability on GPUs that have hundreds of thousands of threads, and focus on the TCO benefits of using GPUs for text based similarity search.

Recent work by Sundaram et al. [34] uses LSH to quickly search across a billion tweets using 100 machines. However, LSH requires massive amounts of state to achieve reasonable quality, using 4000GB (40GB per machine) of state to index just 28GB of data (average 28 chars per tweet), causing scalability issues.

Memcached on GPUs [15, 16] evaluates the popular Memcached distributed workload on GPUs. They perform request batching as well, trading request latency for higher request throughput and energy efficiency. This further validates our findings on the applicability of accelerators to scale-out workloads in datacenters.

Reddi el al. [19] evaluate the efficiency and QoS characteristics of the Bing search engine on Xeon and Atom cores. Text search on GPUs was studied by Ding et al. [7] using a traditional compressed inverted list implementation. Jeon et al. [20] focus on the adaptive parallelization of a query based on the system load and parallelization efficiency. There is a large body of work on search engine quality and scoring techniques such as PageRank [5], which are complementary to our work. Catapult [31] is a recent design that uses FPGAs to improve the ranking throughput of servers, while our work focuses on the document selection problem, which precedes document ranking. Catapult also highlights the increasing interest

in using accelerators to improve performance and cost efficiency in commercial datacenters, and is complementary to our work.

8. Conclusion

Similarity search is a well known problem with a wide range of real world applications across many disciplines. Searching for similar matches to a stream of queries across billions of objects requires thousands of machines, and identifying the right server platform and sharding strategy are crucial for minimizing total cost of ownership.

We evaluate the potential of accelerators to improve the cost effectiveness of cluster designs for similarity search. While general purpose server processors offer higher utility and ease of programmability, accelerators deliver higher throughputs, and result in significant cost and energy savings. We present optimized algorithms for similarity search and show improvements in both throughput and efficiency as compared to Xeon and ARM based systems. Higher throughput per machine results in a reduction in the number of machines needed to perform search, significantly lowering cost of ownership.

Acknowledgments

This work is supported in part by the National Science Foundation (CCF-1335443), an NVIDIA Graduate Fellowship, and equipment donations from NVIDIA. We thank Victor Orlikowski and Jeff Chase for letting us install power meters on their servers.

References

[1] S. R. Agrawal. *Harnessing Data Parallel Hardware for Server Workloads*. PhD thesis, Duke University, 2015.

[2] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 19–34, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541956.

[3] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013. doi: 10.2200/S00516ED2V01Y201306CAC024.

[4] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, Sept. 2001. ISSN 0360-0300. doi: 10.1145/502807.502809.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V. URL `http://dl.acm.org/citation.cfm?id=297805.297827`.

[6] S. Dalton, L. Olson, and N. Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Trans. Math. Softw.*, 41(4):25:1–25:20, Oct. 2015. ISSN 0098-3500. doi: 10.1145/2699470. URL `http://doi.acm.org/10.1145/2699470`.

[7] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance IR query processing. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 421–430, New York, USA, 2009. ACM. ISBN 978-1-60558-487-4. doi: 10.1145/1526709.1526766.

[8] W. Dong. *High-dimensional Similarity Search for Large Datasets*. PhD thesis, Princeton, NJ, USA, 2011. AAI3481579.

[9] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108. URL `http://doi.acm.org/10.1145/2000064.2000108`.

[10] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150982.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-615-7.

[12] W. Gish and D. J. States. Identification of protein coding regions by database similarity search. *Nat Genet*, 3(3):266–272, Mar. 1993. doi: 10.1038/ng0393-266.

[13] N. Goharian, T. El-Ghazawi, and D. Grossman. Enterprise text processing: a sparse matrix approach. In *Information Technology: Coding and Computing, 2001. Proceedings. International Conference on*, pages 71–75, Apr. 2001. doi: 10.1109/ITCC.2001.918768.

[14] Google Zeitgeist 2012. Google zeitgeist 2012. `http://www.google.com/zeitgeist/2012/#the-world`.

[15] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS '12, pages 88–98, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1143-4. doi: 10.1109/ISPASS.2012.6189209.

[16] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 43–57, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806836. URL `http://doi.acm.org/10.1145/2806777.2806836`.

[17] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM. ISBN 0-89791-962-9. doi: 10.1145/276698.276876.

[18] Intel. Advancing moore's law in 2014the road to 14 nm. 2014. URL `http://www.intel.com/content/www/us/en/silicon-innovations/advancing-moores-law-in-2014-presentation.html`.

[19] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1816002.

[20] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Adaptive parallelism for web search. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 155–168, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465367.

[21] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, Sept. 2011. ISSN 0272-1732. doi: 10.1109/MM.2011.89.

[22] T. Kgil, A. Saidi, N. Binkert, S. Reinhardt, K. Flautner, and T. Mudge. Picoserver: Using 3D stacking technology to build energy efficient servers. *J. Emerg. Technol. Comput. Syst.*, 4(4):16:1–16:34, Nov. 2008. ISSN 1550-4832. doi: 10.1145/1412587.1412589.

[23] J. Koomey. A simple model for determining true total cost of ownership for data centers.

[24] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance (Extended Abstract). In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools*

for embedded systems, pages 182–187, New York, NY, USA, 2001. ACM. ISBN 1-58113-425-8. doi: 10.1145/384197.384222.

[25] D. Lipman and W. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985. doi: 10.1126/science.2983426. URL `http://www.sciencemag.org/content/227/4693/1435.abstract`.

[26] W. Liu and B. Vinter. A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors. *J. Parallel Distrib. Comput.*, 85(C):47–61, Nov. 2015. ISSN 0743-7315. doi: 10.1016/j.jpdc.2015.06.010. URL `http://dx.doi.org/10.1016/j.jpdc.2015.06.010`.

[27] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages –, Washington DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4503-1642-2. doi: 10.1145/2337159.2337217.

[28] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

[29] T. Mudge and U. Holzle. Challenges and opportunities for extremely energy-efficient processors. *IEEE Micro*, 30(4):20–24, July 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.61.

[30] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *High Performance Computing*, pages 48–57. Springer, 2015.

[31] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.

[32] A. W. M. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain. Content-based image retrieval at the end of the early years. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(12): 1349–1380, Dec. 2000. ISSN 0162-8828. doi: 10.1109/34.895972.

[33] A. Spink, D. Wolfram, M. B. J. Jansen, and T. Saracevic. Searching the web: The public and their queries. *J. Am. Soc. Inf. Sci. Technol.*, 52(3):226–234, Feb. 2001. ISSN 1532-2882. doi: 10.1002/1097-4571(2000)9999:9999⟨::AID-ASI1591⟩3.3.CO;2-I.

[34] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, Sept. 2013. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=2556549.2556574`.

[35] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé. Vision and challenges for realising the internet of things. *Cluster of European Research Projects on the Internet of Things, European Commision*, 2010.

[36] Verizon. State of the market the internet of things 2015. 2015. URL `http://www.verizonenterprise.com/resources/reports/rp_state-of-market-the-market-the-internet-of-things-2015_en_xg.pdf`.