Harnessing Data Parallel Hardware for Server Workloads

by

Sandeep R Agrawal

Department of Computer Science
Duke University

Date: _____________________

Approved:

__________________________
Alvin R Lebeck, Supervisor

__________________________
Benjamin C Lee

__________________________
Daniel J Sorin

__________________________
Landon P Cox

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2015
ABSTRACT

Harnessing Data Parallel Hardware for Server Workloads

by

Sandeep R Agrawal

Department of Computer Science
Duke University

Date: ____________________________

Approved:

______________________________
Alvin R Lebeck, Supervisor

______________________________
Benjamin C Lee

______________________________
Daniel J Sorin

______________________________
Landon P Cox

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2015
Abstract

Trends in increasing web traffic demand an increase in server throughput while preserving energy efficiency and total cost of ownership. Present work in optimizing data center efficiency primarily focuses on using general purpose processors, however these might not be the most efficient platforms for server workloads. Data parallel hardware achieves high energy efficiency by amortizing instruction costs across multiple data streams, and high throughput by enabling massive parallelism across independent threads. These benefits are considered traditionally applicable to scientific workloads, and common server tasks like page serving or search are considered unsuitable for a data parallel execution model.

Our work builds on the observation that server workload execution patterns are not completely unique across multiple requests. For a high enough arrival rate, a server has the opportunity to launch cohorts of similar requests on data parallel hardware, improving server performance and power/energy efficiency. We present a framework—called Rhythm—for high throughput servers that can exploit similarity across requests to improve server performance and power/energy efficiency by launching data parallel executions for request cohorts. An implementation of the SPECWeb Banking workload using Rhythm on NVIDIA GPUs provides a basis for evaluation.

Similarity search is another ubiquitous server workload that involves identifying the nearest neighbors to a given query across a large number of points. We explore
the performance, power and dollar benefits of using accelerators to perform similarity search for query cohorts in very high dimensions under tight deadlines, and demonstrate an implementation on GPUs that searches across a corpus of billions of documents and is significantly cheaper than commercial deployments. We show that with software and system modifications, data parallel designs can greatly outperform common task parallel implementations.
For Savani - my anchor to me, and my family, the people who created me.
Contents

Abstract iv

List of Tables xi

List of Figures xii

Acknowledgements xv

1 Introduction 1

1.1 Motivation 2

1.2 Thesis Statement 5

1.3 Contributions 9

1.4 Summary 11

2 Rhythm: High Throughput Server Design 12

2.1 Server Software Architecture 14

2.2 The Rhythm Pipeline 17

2.3 Rhythm Server Implementation 19

2.3.1 Pipeline Control 19

2.3.2 Pipeline Stages 21

2.4 Data Structures 22

2.4.1 HTTP Session Array/Cookies 22

2.4.2 Request and Response Buffer Layout 23

2.4.3 Buffer Transpose 23
2.4.4 Error Handling ................................................. 25
2.4.5 Request Flow in Rhythm ................................. 26
2.4.6 Improving Programmability ............................. 26
2.4.7 CUDA Specifics ............................................. 27
2.5 Summary ......................................................... 28

3 Evaluating a Banking server on Rhythm ............ 29
  3.1 Methodology ...................................................... 31
    3.1.1 Platforms ................................................. 31
    3.1.2 Metrics .................................................... 33
    3.1.3 Modeling Future Systems ................................ 33
  3.2 Tesla A: Evaluating the Network ...................... 36
  3.3 Titan: Overall Results ..................................... 40
  3.4 Titan A: Emulated Remote Backend .................... 43
  3.5 Titan B: Integrated NIC and Device Backend .......... 44
  3.6 Titan C: Increasing Device Utilization .................. 46
  3.7 Scaling Many Core Processors ............................ 46
  3.8 System Resource Requirements ............................. 47
  3.9 Miscellaneous ................................................. 48
  3.10 Related Work .................................................. 49
  3.11 Conclusion .................................................... 50

4 Exploiting Accelerators for Efficient High Dimensional Similarity Search ...... 52
  4.1 Similarity Search ............................................. 54
  4.2 Baseline Architecture ....................................... 57
  4.3 Algorithms and Implementation ........................... 57
    4.3.1 Sparse Matrix Matrix Multiplication ................. 59
### 6 Conclusions

6.1 Key Contributions ................................................. 102

6.1.1 Server Design ................................................. 103

6.1.2 Similarity Search .............................................. 103

6.2 Directions for Future Work ........................................ 105

6.3 Summary .......................................................... 107

Bibliography .......................................................... 108

Biography .............................................................. 117
List of Tables

3.1 Experimental System Platforms ............................................. 31
3.2 SPECWeb Banking Workload .................................................. 32
3.3 Rhythm Banking PCIE transactions ......................................... 38
3.4 SPECWeb Banking Experimental Results. *Titan C Latency is for transposed response ......................................................... 41
5.1 Experimental System Platforms ................................................ 78
5.2 Utilization for $T = 50$ ms, (a) $\alpha = 1000$ queries/s and (b) $\alpha = 25000$ queries/s. The power is the total power needed to support the cluster. 87
5.3 Configurations for $T = 50$ ms, (a) $\alpha = 1000$ queries/s and (b) $\alpha = 25000$ queries/s. Each configuration is optimal for that platform giving the highest Work done/second. The total power is the peak power needed to support the cluster. ......................... 89
List of Figures

1.1 Server Design Space ................................................. 3

2.1 Typical Web Server Request Processing Overview ................. 13

2.2 Rhythm Pipeline (Each stage can have multiple instances) ....... 14

2.3 Rhythm Event-based Server: Each Finite State Machine (FSM) is
associated with a cohort of requests. ................................. 15

2.4 Rhythm Pipeline (Detailed) ........................................... 17

2.5 Request Buffer Layout (for illustrative purposes) .................. 24

2.6 Rhythm Web Server Request Processing Overview ................. 25

3.1 Throughput for different cohort sizes and different degrees of paral-
lelism for the CPU and the GPU. Parallelism for the GPU indicates
the width of the Rhythm pipeline and denotes number of threads for
the CPU. ................................................................. 37

3.2 PCIe 2.0 Limitations in Tesla A for various request types .......... 39

3.3 95% latency for different request types for Tesla A for different cohort
sizes, for 8 cohorts in flight. ........................................... 39

3.4 Throughput-Efficiency for Wall Power. Throughput (y-axis) is nor-
malized to Core i7 8 workers and efficiency (x-axis) to ARM A9 2
workers. The shaded region represents the desired operating range. ....... 42

3.5 Throughput-Efficiency for Dynamic Power. Throughput (y-axis) is
normalized to Core i7 8 workers and efficiency (x-axis) to ARM A9 2
workers. The shaded region represents the desired operating range. ....... 43

3.6 PCIe 3.0 Limitations in Titan A for various request types .......... 44
3.7 Throughput-Efficiency for different request types on Titan B (dynamic power). Rhythm buffer sizes that are close to required sizes perform well (shaded area represents the desired operating range).

4.1 Baseline Search Architecture. Each query cohort requires a set of machines to process it.

4.2 Our basic approach for Similarity Search

4.3 Time taken for an SpMM operation by MKL 11.2 and cuSPARSE 6.5 for $\mu = 500000$ and $C = 256$ queries. The MKL measurements were performed on a 16C/32T Xeon, and the cuSPARSE measurements were performed on a GTX Titan. $\text{opthost}$ and $\text{optacc}$ are the times for our algorithm on the host and accelerator respectively.

4.4 Our SpMM approach for $S = Q \times A$. The shaded rows of $A$ are accumulated into the values array of $S$.

4.5 Implementation of similarity search on the accelerator

4.6 Block diagram of a node in our cluster design.

5.1 Performance of the Xeon for different cohort sizes and tile sizes.

5.2 Performance of the Titan for different cohort sizes and tile sizes.

5.3 Throughput-efficiency of text search for different processors for cohort sizes ranging from 1-32. 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).

5.4 Throughput-efficiency of text search for different processors for cohort sizes ranging from 64-1024. 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).

5.5 Optimal platforms for text search for a deadline of 50 ms and $\alpha = 1000$ queries/s. 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).

5.6 Optimal platforms for text search for a deadline of 50 ms and $\alpha = 25000$ queries/s. 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).
5.7 Ratio of the cost of ownership for different platforms as a function of the cost of the Xeon system for $T = 50$ ms, and $\alpha = 1000$ queries/s. . 92

5.8 Ratio of the cost of ownership for different platforms as a function of the cost of the Xeon system for $T = 50$ ms, and $\alpha = 25000$ queries/s. 93

5.9 Comparison of dollars values for different systems for equal cost of ownership. The shaded region represents values for which the ARM clusters are cheaper than Xeon only clusters. . . . . . . . . . . . . . . 94

5.10 Comparison of dollars values for different systems for equal cost of ownership. The shaded region represents values for which the Xeon + Accelerator clusters are cheaper than Xeon only clusters. . . . . . . . . . 95

5.11 Comparison of dollars values for different systems for equal cost of ownership. The shaded region represents values for which the SoC + Accelerator clusters are cheaper than Xeon only clusters. . . . . . . . . . 96
Acknowledgements

Writing this thesis is both a happy and sad occasion for me, happiness because of a sense of achievement, and sadness because of the end of an amazing journey. People look for advisors based on their academic merit, and their awesome papers, but for me, the definition of an advisor has grown far beyond that. An advisor is a mentor, a confidant, a friend, a parent, a teacher, a guru. I am not sure how to even begin to express my gratitude to Alvy, but in his words, his fulfillment is seeing his students succeed. I would like to thank Alvy for keeping his faith in me all these years, allowing me those months of inactivity when I probably seemed like a pointless exercise, keeping my occasionally grey morals and greed in check, allowing me to interrupt him and ramble on and on and on, giving me the freedom of thought and experiment, guiding me along the way avoiding rat holes, for teaching me how to think, how to do research, and in general be a better person.

I would also like to thank Savani, my best friend, fiance and soon to be wife, for enduring me all these years, she is probably the singular reason why I was able to do this. She is my anchor to reality and sanity, my hope for a life beyond my degree, and the answer to my happiness. I hope that for the rest of my life, I can partly return the favor, and give her the same happiness.

My life at Duke would not be complete without Rahul and Sai, friends and fellow gamers at heart. Midnight conversations over a cup of hot tea, insane Borderlands, Dirt and Diablo gaming marathons, these are memories I will cherish for a lifetime. I
express my gratitude to Rahul using a line from one of his favorite gaming moments, 
*Nice shot! Captain Macmillan would be proud.*

I would also like to express my gratitude to the faculty at Duke, Prof. Trivedi for making sense of probability, Pankaj for teaching me algorithms and how to apply them and Landon for his insight into Operating and Distributed Systems. Ben taught me how to think about power in systems and why it matters, and Dan for teaching me the importance of a good context, story and a solid evaluation. I also benefited a great deal from my interactions with my colleagues and office mates, especially Vali, Jun, Jonathan, Chris and Alex. I also want to thank Jeff and Victor, for providing me the Xeon infrastructure for my search experiments. This research would not have been possible without support from John Tran, and NVIDIA, and I am deeply indebted to them.

Finally, I want to thank my family, for inspiring me, thank you Didi, Manav jiju, Anu, Samip jiju, Payal and Bhavani jiju, for believing in me. I would like to thank my parents, for being there and for believing in me. I hope I can repay some of their sacrifices with this.
Data centers provide the computational and storage infrastructure required to meet today’s ever increasing demand for Internet content. For example, Facebook utilizes more than 100,000 servers to provide approximately one trillion page views per month (about 350K per second) and 1.2 million photo views per second. Google processes more than 100 billion queries per month (about 38,000 per second) across billions of pages [37]. Meeting the increasing demand for content often requires adding more machines to existing data centers and building more data centers. Although content distribution networks can offload some of the demand for static content, state-of-the-art data centers house vast numbers of servers and require 2-6 Mega Watts of power. Therefore server performance, scaling and energy efficiency (throughput/Watt) are crucial factors in reducing total cost of ownership (TCO) in today’s server-based industries [4, 15, 24, 25, 67, 70].

Most modern datacenter servers are based on general purpose server class processors like the Intel Xeon line and AMD’s Opteron chips. These powerful chips concentrate significant compute power in a single node, and therefore amortize fixed costs by reducing the number of machines needed for a workload. On the other
hand, there is an increased interest in ARM-based servers [44, 31], which reduce per server capital and operational costs, however they require more machines for the same performance. These 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 [25, 53, 56, 67, 72, 80, 2].

1.1 Motivation

Datacenters currently handle arrival rates of the order of thousands of queries/s [37], 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 [90]. 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 to fourfold to 5.4 billion by 2020 [95]. Much further into the future, the number of devices connected to web services in expected to reach hundreds of billions [90], and the question naturally arises as to what is the best way to satisfy this demand. Conventional wisdom relies on adding more machines and more datacenters [8], which is cheaper and simpler to do in the short term, however, it is inefficient in terms of area. Larger number of machines also increase design complexity, as more infrastructure like network switches, racks, power distribution, etc. is needed to support them.

An alternate approach can be to improve both the throughput and efficiency (throughput/Watt) of existing machines, allowing the same infrastructure to handle larger loads. Improving throughput per machine is essential in order to reduce the number of machines needed, thereby amortizing fixed costs. This throughput increase should not come at the cost of lower efficiency, as an overall increase in workload power requirements is not desirable. In this work, we designate both throughput and
throughput/Watt as first order design constraints. Figure 1.1 shows an overview of the server design space based on throughput normalized to an x86 core versus energy efficiency (performance/Watt) normalized to an ARM core. An ideal design would achieve throughput at or above an x86 core with energy efficiency at or above an ARM core.

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 [22], 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 [49], providing further evidence that we need to find an alternative for datacenters.

Recent highly-threaded accelerators (i.e., NVIDIA Kepler [76], Intel Xeon Phi [50]) achieve >1 teraflop of performance within a 225W power envelope, achieving >5 gigaflops/W. This efficiency arises from several factors, including the amortization
of overheads (e.g., instruction fetch) due to extensive use of vector (SIMD) and multi-threaded (SIMT) hardware, and simpler designs enabled by supporting a more restricted programming model; these accelerators excel at executing regular data-parallel programs, but are often considered unsuitable for arbitrary multi-threaded applications.

GPU-style accelerators achieve efficiency through two primary mechanisms: hardware multithreading and SIMD execution. First, for high throughput computing, hardware multithreading enables significant latency tolerance by overlapping the execution of multiple threads. Second, by utilizing a single instruction multiple thread (SIMT) model, properly aligned threads exploit data parallel execution and amortize instruction fetch and decode overhead. Recent research indicates instruction fetch and decode can be as high as 40%-50% of overall core power [84]. Amortizing this power over more operations can provide benefits across nearly all types of microarchitectures, from complex out-of-order cores to simple in-order cores. Although in this work we focus on GPU-style accelerators, our ideas are broadly applicable to data parallel architectures.

Nonetheless, as GPU-like hardware becomes increasingly general purpose, the potential advantages of highly-threaded, yet somewhat restricted hardware, are compelling. Even servers based on low power cores (e.g., ARM, Atom) could further benefit from amortizing microarchitecture power overheads (e.g., instruction fetch) over multiple requests, or utilizing heterogeneous System-on-Chip solutions (e.g., Tegra [75], AMD Fusion [11], or many core systems [62]) for servers. The challenge is to determine if traditional task parallel request-based server workloads can exploit the efficiencies of highly threaded data-parallel hardware in future server platforms.

This work takes the first steps toward meeting this challenge by building on several observations about technology trends and server workloads. In a typical data center server environment, requests may be distributed to a set of servers based on
information within the request (e.g., based on a hash of the user ID, URL), on the
type of service requested (e.g., login, static image, database query), or other sharding
methods. In this scenario, many of the requests perform the same task(s), such as a
login or search query.

The key insight is, given an incoming stream of requests, a server could
delay some requests in order to align the execution of similar requests—
allowing them to execute concurrently, leading us to our proposed hypothesis.

1.2 Thesis Statement

An acceptable increase in response time can be traded for an improvement
in server throughput per Watt by exploiting similarity across requests us-
ing cohort scheduling to launch data parallel executions on SIMD hard-
ware.

Testing this hypothesis requires an understanding of several constraints and en-
abling trends:

- **Request Completion Deadline** is the time within which at least 95% of
requests are processed. Deadlines within 100 ms are acceptable for servers
handling user facing requests, to ensure fluid responsiveness [18]. Assuming
service time of a single request is far lesser than this, a server has opportunity
to wait for similar requests to batch them together. On the other hand, a
violation of this deadline can lead to a degradation in user experience, and all
optimizations to the system must satisfy this constraint.

- **Similarity across requests** Similar requests when scheduled together can
take advantage of shared resources such as the instruction and data caches,
fetch and decode units, etc. amortizing per instruction costs. Control flow
similarity amongst co-scheduled tasks is essential to achieve high efficiency on
SIMD hardware. Mapping traditional request parallel workloads on Single Instruction Multiple Thread (SIMT) hardware raises several interesting questions about request/thread mapping. Based on similarity characteristics, it might be more suitable to map a single thread to a request, or multiple threads to a request. Similar control flow is just one aspect required to efficiently utilize GPU-style compute accelerators. Other additional challenges include, but are not limited to: 1) scheduling, 2) data divergence, 3) copy overheads, and 4) current platform limitations.

- **Request Arrival Rate** Request arrival rates in datacenters vary with time of day, the workload running and many other parameters. We batch requests together into *cohorts* [61] to take advantages of co-scheduling. As completion deadlines for user facing requests are usually bounded, higher arrival rates are needed to form larger cohorts. The cohort size chosen for a workload is a trade-off between cohort formation time, and efficiency gained from larger cohorts.

- **SIMD hardware** There is a wide variety of SIMD hardware available today, FPGAs, NVIDIA GPUs, AMD APUs, Intel SSE/AVX, etc each with its own advantages. We choose NVIDIA GPUs as our platform for evaluation due to the maturity of the CUDA framework, and large opportunity for optimization. Our ideas are broadly applicable, and can be applied to any SIMD platform.

- **Server workloads** With the advent of cloud computing, many services are being moved to datacenters today, and this trend is expected to escalate further into the future. EMail, Chat, Search, Page serving, Social Networking, all of these are request parallel workloads, with services used by billions of people across the globe. EMail is not latency critical, and Chat has no server process-
ing involved. A majority of the internet is primary web pages, and a majority of access to these pages happens via search engines. We present our ideas by targeting two of these workloads, Text Search and HTTP Page Serving, and evaluate our hypothesis for them.

- **I/O Bandwidth** Current systems may have bandwidth bottlenecks at either the network or the PCIe bus that limit overall throughput. Today’s systems utilize 10 Gbps - 40 Gbps networking infrastructure and PCIe 3.0 for the system interconnect. A single 10 Gbps link limits throughput to approximately 1M req/sec for 1KB requests, and is lower for larger messages. However, efforts are underway to create 100 Gbps and 400 Gbps network standards [47] and announced systems claim to sustain these bandwidths [74]. Furthermore, research is exploring techniques to achieve $10^{12}$ bps to $10^{15}$ bps using novel multicore fiber optics [82].

Once network bandwidth increases, the PCIe bus may become the bottleneck. Fortunately, PCIe 4.0 doubles bandwidth to 16G Transfers/sec [79] and as discussed below, system on chip architectural changes can eliminate/reduce system bus bandwidth limitations. We believe that future server platforms can exploit these progressive enhancements in network and system bandwidth to sustain significant throughput per node.

- **System on Chip** Another trend in processor and system architecture is heterogeneous computing where specialized accelerators and general purpose cores are combined in a system-on-chip (SoC) design (e.g., Tegra [75], AMD Fusion [11]). SoCs can reduce latency, improve bandwidth and reduce power by avoiding off-chip latency and exploiting on-chip density for a high bandwidth interconnect. Recent analysis [62, 63] shows overall benefits to data center design by using SoC architectures to reduce TCO, and mention the possibility of
integrated accelerators for servers.

- **Operating System and Backend Services** Increasing throughput per machine would also require support for high throughput access to OS services. Recent research is exploring both high throughput and clean abstractions for parallel access to OS services. GPUfs [85] provides a filesystem abstraction that enables access from the GPU. GPUNet enables the use of sockets on the GPU [57]. Similar in spirit is the recent development of vector interfaces [94] that can provide 1M input/output operations per second to a high performance solid state disk (SSD). There is also recent work on providing high throughput backend services by exploiting GPUs [6, 43, 99] or constructing specialized hardware for key-value (e.g., memcached [27]) servers [14, 63].

- **Algorithmic complexity and Programmer effort** Current web services are designed for swift deployment and uptime, and a lot of existing server code is written in PHP and Java for rapid prototyping. High level languages hide significant optimization potential behind abstractions, and there is recent work on supporting language abstractions on GPUs [58][46]. We choose C as our language for accelerator programming, and ensure that our comparisons are made to baseline multithreaded C implementations as well.

Existing designs and algorithms for server workloads might not be amenable to a data parallel implementation. Evaluating this hypothesis requires us to come up with new frameworks and algorithms that work best with the underlying hardware. The restrictive programming environment of accelerators makes writing server applications challenging, and requires significant innovations in algorithms to efficiently support cohort scheduling for web services.
1.3 Contributions

We aim to validate our hypothesis by analyzing and evaluating two ubiquitous data-center workloads, a web server and a search engine. This work aims to establish that request parallel server workloads are amenable to data parallel hardware. We summarize our key contributions.

Chapter 2 describes our design of *Rhythm*, a software architecture for high throughput SIMT-based servers using cohort scheduling. (*Rhythm* derives from the Greek word *rhythmos*, meaning any regular recurring motion or symmetry) (Section 2.1). The *Rhythm* pipeline achieves high efficiency by maximally utilizing the accelerator, with resource shortage as the only constraint. A prototype implementation of *Rhythm* and the SPECWeb Banking service on NVIDIA GPUs demonstrates that it is possible to run server workloads on a GPU.\(^1\) (Section 2.3).

The SPECWeb workload is written in PHP making it unsuitable for running on accelerators due to the lack of an available runtime. We implement standalone C and C+CUDA versions of the SPECWeb2009 Banking workload. Chapter 3 evaluates these implementations on future server platform architectures. We evaluate an *rsockets* based version of our server running on Infiniband hardware, and demonstrate that *Rhythm* achieves throughput higher than a quad-core Xeon processor even under PCIE bandwidth limitations.

Our prototype achieves 1.5M requests/sec on an NVIDIA GTX Titan GPU card for an idealized environment that removes network and storage I/O limitations. This is 4\(\times\) the throughput of a Core i7 running 8 threads at efficiencies comparable to a dual core ARM Cortex A9. Furthermore, approximately 192 1.2GHz, 1W ARM cores are required to achieve the same throughput with less than 40 Watts (21% overall) in available power to support the massively scaled system. We also demonstrate

\(^1\) Our initial prototype targets GPUs, but our ideas are applicable to a broad class of accelerators. Without loss of generality, we use GPU or device to refer to this broad class.
that array transpose offload can increase *Rhythm* throughput to over 3M reqs/sec (Section 2.3).

We then look at the high dimensional similarity search problem with applications in text search engines, recommender systems, behavioral ad targeting, image search and retrieval, etc. Chapter 4 provides 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 (Section 4.3). We also look at cost of ownership models for evaluating accelerator based clusters in Section 4.5. These models enable us to compare the cost of ownership of two different CPU based configurations, or a mix of CPU and accelerator based cluster configurations for similarity search.

Chapter 5 demonstrates cohort scheduling for text search under constrained deadlines. Query cohorts reduce cache contention on general purpose cores, and allow for higher occupancy and amortization of fixed costs on the accelerator. An implementation of text search on NVIDIA GPUs is more than twice as efficient and delivers more than twice the throughput of the corresponding implementation on Xeon and ARM cores (Section 5.3). Using this implementation, we show an SoC + Accelerator approach that delivers $2.3 \times$ the throughput of a Xeon server while being 75% more energy efficient than an ARM server for an arrival rate of 25,000 queries/s and a deadline of 50 ms. (Section 5.4)

Using capital and operational cost models from Chapter 4, we evaluate text search under deadline constraints and show 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 (Section 5.5). An alternative cluster design using a low power SoC driving an accelerator is $6 \times$ cheaper and consumes less overall power than a commercial Xeon based server cluster, for an arrival rate of 25,000 queries/s such that 95% of queries finish within
50 ms. We prove that even with zero cost host processors, retail priced accelerators are still cheaper due to significant savings on operational costs from higher energy efficiency.

1.4 Summary

Cloud based services and products will be dominant in the coming decades with technology dominating every aspect of our lives from the way we communicate to the way we process information and visualize the world. These services require extensive computation capabilities. The end of Dennard scaling and dark silicon has saturated the processing capabilities of the personal computer, creating a paradigm shift for consumers from personal computers to thin clients. Datacenters lie at the heart of these services, making throughput and efficiency scaling for these warehouse-scale computers a fundamental challenge for the next generation. This work serves as a milestone that ties together traditional web services and SIMD accelerators, by providing large single server gains in throughput and throughput/Watt, allowing existing datacenters to scale without sacrificing user experience.
Rhythm: High Throughput Server Design

HTTP Web servers serve as the backbone of the Internet, more intelligent services like search and social networking are crucial applications, however a majority of Internet activity is still serving pages. The Wikimedia sites alone stack upto 20B pageviews per month, and popular sites like Facebook serve millions of requests per second to a billion users. At Facebook, the majority of the traffic goes through front-end web servers and the majority of the data center servers (and thus power) are devoted to this front-end [39]. Reducing CPU load inspired the design of HipHop, which translates PHP to native code thereby increasing throughput per server by $> 5 \times$ [23].

Usage patterns found in web servers are generally viewed as transactional. In its simplest form, a client issues a request to the server and receives a response in return. Figure 2.1 shows the typical pattern for processing requests in web servers. Requests arrive over the network and are dispatched for processing, generally based on the type of request (e.g., a particular PHP file or query type). Request processing may be followed by access to a backend database (e.g., SQL or memcached), and the results from that query can lead to further processing before a response is sent back.
to the client. Conventional servers process requests individually, using a thread per request [26] or a staged event-based server [98, 61, 91].

We propose to improve efficiency based on the observation of similarity between requests. Intelligently grouping and scheduling incoming requests that execute the same code can be used to reduce overheads. Previous research demonstrated that request batching increases server throughput by using cohort scheduling [61]. The idea is to delay servicing a request until additional requests arrive at a similar point in their processing, and then execute the threads in this group (cohort) consecutively. This approach improves instruction and data locality of the server workload, improving overall performance. We believe that similarity of requests can be further exploited by the hardware to improve efficiency.

*Rhythm* is a software architecture that extends event-based staged servers and cohort scheduling [98, 61, 91] to support high throughput servers on emerging highly threaded data parallel accelerators. Conceptually, *Rhythm* pipelines the processing of request *cohorts*—a set of requests that require similar computation. In this section we first provide an overview of the *Rhythm* design, including our core design goals. This is followed by a more detailed description of the *Rhythm* pipeline and the requisite data structures. *Rhythm* is a general architecture that could be implemented in many
ways; Section 2.3 describes a specific implementation. Furthermore, *Rhythm* can be used to implement a variety of services, but in this work we focus on its use for web servers.

The key contributions in this chapter are:

- *Rhythm*, a software architecture for high throughput SIMT-based servers. (*Rhythm* derives from the Greek word *rhythmos*, meaning any regular recurring motion or symmetry) (Section 2.1)

- A prototype implementation of *Rhythm* on NVIDIA GPUs that demonstrates it is possible to run server workloads on a GPU.¹ (Section 2.3)

### 2.1 Server Software Architecture

*Rhythm* extends event-based staged servers by providing a pipelined architecture for processing cohorts of requests on data parallel hardware. The *Rhythm* pipeline is composed of five stages, as shown in Figure 2.2: 1) Reader, 2) Parser, 3) Dispatch, 4) Process, 5) Response. For each of these stages there may be one or more instances, allowing for parallelism across and within stages. For a given service, the

¹ Our initial prototype targets GPUs, but our ideas are applicable to a broad class of accelerators. Without loss of generality, we use GPU or device to refer to this broad class.
Figure 2.3: Rhythm Event-based Server: Each Finite State Machine (FSM) is associated with a cohort of requests.

The overall goal of Rhythm is to enable high throughput server implementations that can exploit the efficiencies of GPU-style accelerators. To achieve this, our design goals for Rhythm include: 1) Asynchronous, 2) Event driven, 3) Lock free & wait free, and 4) Utilize the most efficient computational resource (general purpose core or accelerator). The first three guidelines are well-known for high throughput server design on commodity general purpose processors (e.g., nginx [91]). The last design guideline is unique to an accelerator-based design and reflects our desire to exploit the efficiency of the accelerator as much as possible, but some requests that do not conform to a data parallel model may be executed more efficiently on a general purpose CPU.

The Rhythm pipeline is controlled by an event-based server with a single thread (see Figure 2.3) that provides cohort scheduling [61]. The server thread is generally responsible for delaying requests an appropriate amount of time to form cohorts, determining when a cohort is ready to launch on the accelerator, launching cohorts
onto the accelerator, managing cohort context transitions, and sending responses back to the originating clients.

Requests can be delayed for a limited amount of time and still achieve acceptable response times [70]. Rhythm includes a timeout so that requests are not delayed indefinitely during cohort formation. Setting a specific timeout value is a policy decision that depends on particular service level agreements, Rhythm simply provides the mechanism. A similar timeout mechanism could be used to ensure that stragglers (e.g., long backend accesses) do not delay other requests in a cohort during execution. Furthermore, since cohort formation can occur after each stage, the set of requests in a cohort may change. Straggler responses from the backend can either be executed on the host CPU or added to a subsequent cohort.

The Rhythm pipeline stalls only when insufficient resources are available (i.e., structural hazards such as memory buffers or device execution units). The single threaded control avoids thread switching overheads, is lock-free, wait-free and fully asynchronous. The design also allows for multiple instances of each stage (reader, parser, process and response) and each of the stages can be tuned for optimal concurrency. Rhythm maintains state that allows it to efficiently schedule cohorts on the host or the accelerator based on the current state of the system.

Cohort Management A cohort context contains information necessary to identify the specific request type and other properties of a cohort. Rhythm uses a cohort pool to track the availability of cohort contexts and manages their allocation throughout the pipeline. A cohort context can be Free, Partially_Full, Full or Busy. A Free context can be used to form a new cohort. Requests are added to a context by either the Reader or the Parser. The first request added to a Free cohort context transitions the context to Partially_Full where it can continue to accumulate requests until it becomes Full. When a cohort context begins execution, either because it was Full or a timeout occurred, it becomes Busy. A cohort context is Busy while the requests
transition through the various process stages. A cohort context is Freed after the responses are sent to their respective clients.

2.2 The *Rhythm* Pipeline

We now elaborate more on the design and function of the individual stages of the *Rhythm* pipeline (Figure 2.4). An implementation would provide resources (execution units and memory) to support one or more instances of each stage.

**Reader** The reader accumulates requests from the network to form a cohort, based purely on the request order. When a sufficient number of requests arrive, the reader passes the cohort to the next stage of the pipeline—the parser. The reader latency is primarily limited by the request arrival rate or network bandwidth. If all parsers are busy, the reader stalls.

**Parser** Request parsing follows a standard protocol defined by the HTTP Specification [28], making it an ideal candidate for SIMT execution. The parser extracts the method (GET or POST), the request type (PHP file or image), the content length, cookie information, the client file descriptors and the query string parame-
ters for each request in the cohort. This information is then composed into a request structure and added to the cohort. Cohorts are formed based on the file accessed or any other metric of similarity. The parser then signals dispatch if a cohort becomes Full.

**Dispatch** Dispatch is performed on the host and determines if a request should be executed on the host or the device. Some requests that simply access the file system are best processed on the host rather than the device. GPU access to the file system (e.g., GPUfs [85]) would enable dispatch execution on the device and decrease transfer overheads. A Full cohort context is ready for dispatch if the requisite resources on the device are available. Based on the resource requested (e.g., specific PHP file), the appropriate process stage is executed and the cohort context is updated to Busy to prevent redundant dispatch.

**Process** The process phase is defined by the different request types supported by the server and is generally composed of \( n \) backend stages and \( n + 1 \) process stages. Typically, a web service response contains the HTTP header, static HTML content, database content, and dynamic HTML generated based on the database content. The overall process phase alternates between content generation and backend access. For a typical remote backend, individual threads in a given process stage generate request strings which are sent to the backend, and the backend response is then passed on to the next stage of content generation (i.e., another process stage). We note that it is easy to incorporate a portion of the backend, such as a cache lookup, as part of a process stage. When the response is ready, an event is raised to signal process completion.

**Response** The response stage sends the responses to the respective clients and frees the associated cohort context. Its latency is primarily limited by available network bandwidth.

The *Rhythm* pipeline is general and could be implemented entirely on a single
machine or distributed across several machines. For example, read and parse could be implemented on a front-end machine, processing of specific request types on separate machines, and response formation on yet another machine. The next section presents an implementation on a single machine.

2.3 Rhythm Server Implementation

We implement a prototype version of Rhythm to evaluate its potential on existing NVIDIA GPU hardware. We apply several implementation specific optimizations to the Rhythm design. Many of these optimizations are well known software constructions and Rhythm exploits them for its goals of maximum throughput on accelerator hardware.

2.3.1 Pipeline Control

Event loop Rhythm is single threaded by design, therefore it uses a central event loop based on epoll to process all I/O events and device interactions. These include requests for new connections, backend responses, service requests for existing clients, and file system responses. The event loop is set to continuously poll, as each iteration of the event loop also allows the reader to read requests off the client sockets. The server socket is set as non-blocking, so that accept calls immediately return control to the event loop. Asynchronous stage execution (i.e., transitions in the cohort FSM) is managed using callbacks and local unix pipes. Callbacks are maintained as a linked list that is traversed on each iteration of the event loop.

Callbacks We use callbacks to track stage completion and transitions. The pipeline stages execute asynchronously on the device, and we need a mechanism to detect completion. Unfortunately, our current platform does not support interrupts from the device, therefore we use callbacks to implement a polling mechanism in the event loop. At the beginning of each stage of the pipeline, a callback for an
instance of that stage is added to the callback list. We execute this callback to poll the stage and only remove it from the list if the stage is complete. When a stage completes, it adds another callback to start execution of an instance the next stage in the pipeline. Although this approach doesn’t appear to limit throughput, it introduces unnecessary power consumption for polling many in-flight stages.

**Cohort context synchronization** Our current implementation uses the host for cohort dispatch, and thus requires maintaining cohort context on both the host and the device. We synchronize these two copies of the context at the parser since it is the only stage that modifies device contexts (setting them to *Partially Full* and *Full*) by populating cohorts with requests. The host contexts are copied to the device at parser launch, and the device contexts are copied to the host at parser termination for use in dispatch. Dispatch and response both execute on the host and modify the host cohort context by setting it to *Busy* or *Free*, respectively.

**PCI-E transfers** As the *Rhythm* pipeline is interspersed between the host and the device, cohort context synchronization and inter-stage communication takes place over the PCI-E bus. The following transactions take place in the lifetime of a single cohort.

1. Client Request buffers from the Reader on the host to the Parser on the device.
2. Cohort state from the Reader on the host to the Parser on the device.
3. Cohort state from the Parser on the device to the Reader on the host.
4. Backend requests from the device to the host.
5. Backend requests from the host to the Network Card.
6. Backend responses from the Network Card to the host.
7. Backend responses from the host to the device.
8. Client responses from Process on the device to Response on the host.

The size of each of these transactions increases with the cohort size, and larger cohort sizes and wider pipeline designs require significant PCI-E bandwidth to maintain throughput. The backend request copies from the device to the host and and response copies from the host to the device can be eliminated by using an RDMA based implementation like GPUNet [57], reducing PCI-E bandwidth requirements at the expense of accelerator compute bandwidth, as the network stack processing now occurs on the GPU.

2.3.2 Pipeline Stages

Reader/Parser The reader is double buffered to overlap request processing and request accumulation. When a cohort becomes full, the reader swaps the front and back buffers, signals the parser to begin processing, and resumes reading requests into the new front buffer. If the back buffer is not free, the reader stalls, waiting for the parser. The reader and parser are low latency stages, and a single instance of each is sufficient to achieve high throughput in our experiments.

Process The overall processing of a request is divided into one or more process stages separated by backend accesses. In our current implementation we preallocate pipeline resources (i.e., memory) for all process stages (including the backend when appropriate) and the response stage at the first process stage launch. Each stage is a computational unit on the device, called a kernel. Kernel progress and termination is tracked by the event loop. The overall process phase is latency bound depending on the specific request type. We provide multiple instances of process and response resources to maintain as many cohorts in flight as possible, allowing us to hide the latency of individual kernels and increase throughput.

Backend We open multiple persistent connections to the backend to avoid connection setup and breakdown overheads. Multiple connections are necessary in order...
to hide backend latency and pipeline backend requests and responses from different process stages. These connections can be maintained on the host using the standard socket interface or on the device using GPUNet [57]. If the backend resides on the host, requests and responses need to be transferred over the PCI-E bus. The network transfer latency and backend processing time both increase with larger cohort sizes, and more in-flight cohorts are needed in order to hide backend latency.

**Response** The final process stage adds a callback upon termination, and the response stage is invoked at the next iteration of the event loop. The callback is removed after the responses for the cohort are sent to the respective clients. All resources used by this cohort become available for reuse by subsequent cohorts.

2.4 Data Structures

*Rhythm* uses four primary data structures to support the web server: a cohort pool and the associated cohort contexts, a session array, a request buffer and a response buffer, each optimized to enable efficient data parallel execution. The cohort pool and contexts are implemented as static arrays to avoid allocation and synchronization overheads.

2.4.1 HTTP Session Array/Cookies

The session array is a device only structure that stores HTTP session state for the clients handled by the server. Sessions are created at login and destroyed upon logout. Since the session array is accessed for every request, its performance is critical to server throughput. To ensure conflict-free access for a cohort, the session array is implemented as a hash table with the number of buckets equal to the cohort size. Each request thread accesses a unique bucket, and insertion into a bucket is performed randomly based on a hash of the userid. The session identifier is a hash of the node index and the bucket index, ensuring O(1) time lookup for a session node.
Collisions upon insertion are handled using a linear search for a free node, resulting in $O(1)$ for collision free insertion, and $O(n)$ time in the case of a collision. Since lookups are constant time and the array is static, deletions are $O(1)$ time.

### 2.4.2 Request and Response Buffer Layout

In a typical server application each request is allocated contiguous buffers for incoming and outgoing data. Unfortunately, when executing on a GPU this data layout can undermine the potential benefits of executing multiple requests simultaneously. Specifically, GPU memory systems work best when memory references of the co-scheduled threads (referred to as a *warp*) exhibit good spatial locality (often called *coalesced* memory accesses). While a single thread has good spatial locality, across threads the memory locations accessed are separated by large distances.

To overcome this challenge we explore several methods. One approach utilizes the GPU threads to cooperatively perform the operations of a single request (intra-request concurrency). Unfortunately, this approach does not exploit the similarity in instruction control flow across requests (inter-request concurrency) and performs poorly. Therefore, we use a second approach that performs a data transformation on the buffers by transposing them to improve spatial locality. We also insert whitespaces in the generated HTML content to tolerate control divergence in the response generation stages.

### 2.4.3 Buffer Transpose

We view the buffers per cohort as a 2D array with each row representing the contiguous buffer for a given request, as shown in Figure 2.5. Initially these buffers are in row-major layout. To improve spatial locality within a cohort, we need the array in column-major layout so that thread buffers are interleaved in the sequential address space. To achieve this, we perform a simple array transpose operation and
leverage existing techniques to optimize the transpose [83]. When the requests are finished processing we perform an additional transpose to convert the responses back to row-major layout, with each buffer occupying contiguous locations in the linear address space. Other server workloads may require similar data structure design/transformations to fully utilize the hardware capabilities available. Our ongoing work is exploring other server workloads.

**Whitespace Padding in HTML Content** Transposing buffers can provide coalesced memory accesses if the individual thread buffer pointers are aligned (i.e., each thread uses the same row index value). However, the web pages in our system are dynamically generated with data returned from the backend database; differences in returned data (i.e., string lengths) can result in unaligned buffer pointers. Fortunately, we can exploit the HTML specification, which allows an arbitrary number of linear white spaces in the response body, to embed the appropriate number of whitespace characters after newline characters for each buffer to realign the buffer pointers.
**Whitespace Padding in HTML Headers** The HTTP response header requires the *Content-length* field whose value can only be known after the response is generated. Conventional servers can generate the header after the entire response content is created and use separate `send()` system calls for the header and response. For our *Rhythm* implementation, we avoid the overhead of an additional header generation stage by integrating header creation with response content creation. This creates an issue since the header is located near the beginning of the response buffer. To overcome this and ensure buffer pointer alignment we again exploit the HTML specification which allows white spaces after a header field. We reserve a fixed amount of space in the buffer by inserting white space characters (10 for a 32-bit content length), and replace whitespace with the actual content length value after the response is generated.

### 2.4.4 Error Handling

*Rhythm* maintains per request error state information to guarantee correctness. Request errors create control divergence among threads in a cohort; however, we assume that these scenarios are rare and do not impact throughput.
2.4.5 Request Flow in Rhythm

Figure 2.6 shows an overview of the request flow through the Rhythm pipeline. The ovals represent execution on the host and the boxes represent execution on the accelerator. The event-based server on the CPU copies request information into buffers on the device (step 1). When a sufficient number of requests arrive or the oldest request reaches a preset timeout, the parser is launched (steps 2-3) to identify and sort requests so that requests of the same type (PHP file) are contiguous in memory. The next stage (step 4) dispatches cohorts (and possibly processes some requests on the host CPU). Subsequent stages in the server include accessing backend storage services (step 7), handling backend responses (steps 8-10), and generating final HTML responses (steps 11-13).

2.4.6 Improving Programmability

A very common code block where SIMD code differs from non-SIMD code is memory accesses. We use a thread per request in Rhythm, and memory accesses of threads in a warp follow a strided pattern. In order to simplify programming for SIMD hardware, we rewrite all memory loads and stores using a transpose macro defined as

\[
\text{TRANSPOSE}(x) = (\text{COHORT\_SIZE} \times x)
\]

where \(x\) is a memory offset. HTTP processing and page generation is primarily string operations, and this allows us to rewrite all string functions on the device such as strcpy(), strstr(), strtok(), strcmp(), atoi(), etc using this definition. For example, the strcpy() routine can be written as

```c
char* device_strcpy(char* s1, const char* s2)
{
    char *dst = s1;
    const char *src = s2;
    while ((dst = *src) != '\0') {
```

26
This greatly simplifies writing device side code, and eases code portability between the device and the host.

2.4.7 CUDA Specifics

Our Rhythm prototype is implemented using CUDA on NVIDIA GPUs. A stream is defined as a sequence of dependent requests (memory copies or kernels) to the device, and different streams can execute concurrently on the device. We use asynchronous streams to implement the parser, the various process stages and the response stage of the Rhythm pipeline. Memory pools are created at startup to avoid allocation and synchronization overheads, and memory is recycled.

We use atomics to perform lock-free insertion and deletion into the session and cohort pools. For padding in HTML content we perform a max butterfly reduction across a warp that uses CUDA shared memory to calculate the padding amount for each thread. We use CUDA constant memory where possible to store static HTML content for pages. We store frequently used pointers in CUDA constant memory instead of local memory to optimize register usage and enable more inflight Rhythm cohorts.

Rhythm is designed from the ground up keeping high throughput in mind, ideally with request arrival rate as the only limiter. Our implementation is guided by our experience with SPECWeb Banking, and there are nearly endless opportunities for continued optimization.
2.5 Summary

A dramatic increase in the number of data centers in the last decade and their multi-megawatt power budgets has sharply brought into focus the energy economics of web services. We propose the use of data parallel accelerators and a software architecture called *Rhythm* to address throughput and efficiency demands of future server workloads. *Rhythm* is based on the insight that maximal power efficiency of an accelerator comes from maximizing utilization since it amortizes fixed system costs.

Data parallel hardware is primarily used to accelerate scientific workloads, however *Rhythm* shows that even traditional request parallel workloads like web services can benefit from SIMD/SIMT execution. There exists sufficient similarity between incoming requests to a web server, allowing *Rhythm* to take advantage of amortized fetch and decode across multiple requests, improving energy efficiency. Using a more restricted programming model and amortized fetch and decode overheads, SIMT accelerators are an attractive option, and *Rhythm* provides a way for server workloads to take advantage of this programming paradigm.
Evaluating a Banking server on Rhythm

*Rhythm* shows that traditional web service workloads can be run on accelerators with tolerable latencies. It serves as an important milestone in bridging the gap between datacenters relying on traditional general purpose cores to cheaply scale out, and commercial throughput accelerators which are primarily used to accelerate scientific functions. Recent programmability improvements for accelerators (CUDA, OpenCL) and the *Rhythm* framework enable the use of accelerators for server workloads, creating a potential paradigm shift in the architecture for web servers.

Traditionally, running a high throughput web server involved investing in expensive server class systems like the Xeon, but with *Rhythm* it now becomes possible for individuals and companies to run servers cheaply using commodity accelerators, potentially scaling out to thousands of users. Web servers and commercial datacenters represent a large chunk of the hardware commodities market, and this new demand for accelerators provides ample opportunity and profitability for industry, spurring innovation.

We now evaluate SPECWeb Banking as a potential workload for data parallel execution. The request parallel nature of Banking and control-flow similarity between
requests of the same type allow for improvements in both throughput and energy efficiency [2]. Many web service workloads are primarily request parallel, and this work opens up the way to exploit commercial accelerators to improve throughput/Watt for a wide range of traditional server applications like web search, media streaming, recommendation systems, news/opinion aggregators etc, to name a few. Many of these workloads currently suffer from scaleout challenges [25], and Rhythm provides a way to scale to hundreds of thousands of concurrent requests on a single server, increasing both throughput/machine, and at the same time reducing energy/request.

The key contributions in this chapter are:

- Standalone C and C+CUDA implementations of the SPECWeb2009 Banking workload.(Section 3.1)

- An evaluation of Rhythm using rsockets and Infiniband that shows that even with PCIE limitations, Rhythm outperforms a quad core Xeon processor.(Section 3.2)

- Evaluation of future server platform architectures. Our prototype achieves 1.5M requests/sec on an NVIDIA GTX Titan GPU card for an idealized environment that removes network and storage I/O limitations. This is 4× the throughput of a Core i7 running 8 threads at efficiencies comparable to a dual core ARM Cortex A9. Furthermore, approximately 192 1.2GHz, 1W ARM cores are required to achieve the same throughput with less than 40 Watts (21% overall) in available power to support the massively scaled system. We also demonstrate that array transpose offload can increase Rhythm throughput to over 3M reqs/sec.(Section 3.3)
Table 3.1: Experimental System Platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>GHz</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i5</td>
<td>3.4</td>
<td>Core i5 3570, 22 nm, 4 cores (4 threads), 8GB DDR3 RAM, 1Gbps NIC</td>
</tr>
<tr>
<td>Core i7</td>
<td>3.4</td>
<td>Core i7 3770, 22 nm, 4 cores (8 threads), 16GB DDR3 RAM, 1Gbps NIC</td>
</tr>
<tr>
<td>ARM A9</td>
<td>1.2</td>
<td>OMAP 4460, 45 nm, Panda board, 2 cores, 1GB LPDDR2 RAM</td>
</tr>
<tr>
<td>Titan</td>
<td>0.8</td>
<td>GTX Titan, 28 nm, 14 Streaming Multiprocessors, 6GB GDDR5 Memory</td>
</tr>
<tr>
<td>Tesla</td>
<td>0.7</td>
<td>Tesla K20c, 28 nm, 13 Streaming Multiprocessors, 5GB GDDR5 Memory</td>
</tr>
<tr>
<td>Xeon</td>
<td>3.1</td>
<td>Xeon E3-1220v3, 22 nm, 4 cores (4 threads), 32GB DDR3 RAM, QDR 40Gbps Infiniband NIC</td>
</tr>
</tbody>
</table>

3.1 Methodology

3.1.1 Platforms

Table 3.1 shows the various platforms we use to test workload performance. We use the quad-core Core i5 and Core i7 to represent the x86 family, and the dual-core Cortex A9 to represent the ARM family. The NVIDIA GTX Titan is used for our GPU measurements.

We use the SPECWeb Banking benchmark [88] for our studies. For general purpose processors we implement a standalone event-based C version and for the GPU we implement a Rhythm C+CUDA version. We remove HTTP headers from BESIM requests and responses in our port, as these would waste bandwidth inside a real datacenter. We implement 14 out of 16 Banking requests, and normalize the request percentages to sum to 100%. We skip the quick pay and check detail images benchmarks. Quick pay uses a variable number of kernel launches based on backend data, making it difficult to implement, and check detail images is completely disk bound, requiring GPUs integration to allow us to process it on the GPU. Table 3.2 summarizes characteristics about the Banking workload. The second column is the dynamic instruction count for our standalone C implementation and is the average
Table 3.2: SPECWeb Banking Workload

<table>
<thead>
<tr>
<th>Request Type</th>
<th>x86 Insns per Request</th>
<th>Response Size (KB)</th>
<th>Fraction of Requests (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SPECWeb</td>
<td>Rhythm</td>
</tr>
<tr>
<td>login</td>
<td>132,401</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>account summary</td>
<td>392,243</td>
<td>17</td>
<td>32</td>
</tr>
<tr>
<td>add payee</td>
<td>335,605</td>
<td>18</td>
<td>32</td>
</tr>
<tr>
<td>bill pay</td>
<td>334,105</td>
<td>15</td>
<td>32</td>
</tr>
<tr>
<td>bill pay status output</td>
<td>485,176</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>change profile</td>
<td>560,505</td>
<td>29</td>
<td>32</td>
</tr>
<tr>
<td>check detail html</td>
<td>240,615</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>order check</td>
<td>433,352</td>
<td>21</td>
<td>32</td>
</tr>
<tr>
<td>place check order</td>
<td>466,283</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>post payee</td>
<td>638,598</td>
<td>34</td>
<td>64</td>
</tr>
<tr>
<td>post transfer</td>
<td>334,267</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>profile</td>
<td>590,816</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>transfer</td>
<td>277,235</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>logout</td>
<td>792,684</td>
<td>46</td>
<td>64</td>
</tr>
<tr>
<td>Average</td>
<td>429,563</td>
<td>15.5</td>
<td>26.4</td>
</tr>
</tbody>
</table>

across 100 random requests, and when combined with the third column we see a diverse mix of requests with varying compute/response byte ratios.

We use Ubuntu 12.04 with CUDA 5.5RC on our x86 platform and Linaro 13.01 for our ARM platform. All code is compiled using gcc with -O3 enabled. We test our server against the SPECWeb client validator to guarantee correctness. The X server is shut down for all our benchmark runs. We unplug the GPU for the baseline x86 test runs. For the CUDA version, we allocate 1KB per backend request and 4KB per backend response. We use the next higher power of two for the HTML response size (Table 3.2), since powers of two allow us to easily divide work on the hardware for the response transpose.

We implement support for static images, however, image throughput is primarily dictated by network bandwidth since there is no processing involved. The parser groups image requests into an image cohort, these cohorts bypass the process stage and the image responses are sent to the respective clients. Image cohorts can be
processed on the device as well using GPUs [85]. Static images can also be served at high throughputs via Content Delivery Networks (CDNs) like Akamai [78]. We do not evaluate image throughput for our prototype.

3.1.2 Metrics

Our metrics of interest include 1) throughput, 2) power, 3) latency, and 4) throughput/watt. We obtain throughput using the unix `clock_gettime()` interface to measure end-to-end time to process a set number of requests. Latency is calculated by logging the time that a request arrives and subtracting the request completion time, and we compute an average latency over all requests. We also compute the 95% latency for individual request types. Power is measured at the wall outlet using a Kill-A-Watt meter. We measure the idle and test power for each of our runs. Subtracting the two gives us the dynamic\(^1\) power consumed by the workload. We examine throughput/Watt for both wall power and dynamic power as both of these represent different viewpoints on system efficiency. A system’s cost of ownership is effectively based on wall power, whereas dynamic power measures the marginal costs incurred due to load.

3.1.3 Modeling Future Systems

SPECWeb’s default test harness is based on Java and is quite slow, rendering it unusable for our system. We create our own test harness and use various optimizations to allow us to efficiently model future high bandwidth networks and datacenter-level throughputs.

\(^1\) We use dynamic to represent the non-idle power under load. This is distinct from dynamic/static power used for circuit analysis.
Input Generation

We use the C `rand()` function to randomly generate input request data. For request types other than `login`, we randomly generate session identifiers and populate the session array with random user ids. We test each request type in isolation and process 16M requests of each type on the Titan, and 1M requests of each type on the Tesla due to infrastructure constraints. Using the request distributions from Table 3.2, we compute a weighted harmonic mean of request efficiency (throughput/watt) to obtain the efficiency for the entire workload.

Measuring the Impact of the Network

To measure the performance of Rhythm with the network in place between the Rhythm server and the backend, we use a different infrastructure provided by the GPUNet team at UT Austin [57]. We use two Xeon E3-1220v3 machines equipped with Tesla K20c GPUs and connected via a QDR 40Gbps Infiniband interconnect, using Mellanox HCA cards with the MT26428 chipset. The network can sustain a maximum of 3.44 GB/s bandwidth in any given direction. We could not measure power on this infrastructure, so we only perform throughput and latency measurements.

We measure a configuration in which the backend resides on the host on a separate machine, and backend requests and responses are copied between the accelerator memory and the host memory and then between host memory and the network card over the PCIE bus. We use the Tesla K20c accelerator for these measurements, and call this configuration Tesla A. The baseline x86 version for this variation uses the Xeon processor.

The backend implementations on the two machines use the `rsockets` [41] library to communicate over the Infiniband network. We optimize network bandwidth utilization for the baseline by first transmitting the length of the message, and then the
actual message instead of the entire 4KB. This is not feasible for the accelerator, as we transmit a cohort of requests, and all messages in the cohort are part of a larger buffer.

*Emulation*

We look at the raw performance achieved by the host and accelerator platforms without the network, as a proof of concept of the applicability of SIMD accelerators to HTTP servers. Similarly, PCIE bandwidth may artificially limit throughput. To explore future system architectures, we model three different *Rhythm* systems that progressively add capabilities: Titan A, Titan B, and Titan C.

**x86 and ARM platforms** run our C version of the Banking workload. We use the i7 and i5 to represent the x86 platform, as we are unable to measure power on the Xeon. For maximum throughput, we eliminate network and PCI limitations by generating requests from and copying responses to main memory and implement the backend as a function call.

**Titan A** models a high bandwidth network by generating requests locally and not sending responses across the network. We run the backend locally as one or more threads on the host to emulate the requisite backend throughput. We pre-generate requests into a buffer, and read them from memory on the fly to emulate high arrival rates.

**Titan B** extends the above design to eliminate the PCIE bandwidth bottleneck by implementing the SPECWeb Besim backend on the GPU. This emulates the effect of an SoC style approach with an integrated general purpose core and NIC. A local device backend also avoids the need to transpose the backend request and response data, potentially further improving performance.

**Titan C** further extends our system to emulate specialized hardware that performs the final transpose on the device after response generation, just prior to sending
the response on the network. The response transpose could be performed on the host, on the NIC while reading data to send to the clients, or by a specialized logic unit associated with the memory controller (e.g., the logic layer of a 3D DRAM [45]). The latter is a general approach that could be used for other transpose operations in the Rhythm pipeline.

All of our optimizations are validated for correctness since we run on real hardware, and we can examine the output. A local or device backend emulates a high throughput key-value store [94], or the use of a database cache [27] on the local machine, which is commonly used to tolerate backend latencies.

3.2 Tesla A: Evaluating the Network

Rhythm enables a high throughput server design by having hundreds of thousands of requests in flight keeping the accelerator fully utilized for higher energy efficiency. This leads to an obvious question, Can the network sustain these throughputs? Or would the network stack overheads eclipse the gains from Rhythm?. We answer these questions by evaluating the Rhythm pipeline with a real network stack, and examine performance.

A standard 1Gbps NIC can only sustain 31K 4KB requests in flight, and would bottleneck the system. The accelerator would remain idle most of the time, as the system is network bound degrading energy efficiency. We utilize a 40Gbps Infiniband network between the server and the backend, which would allow 900K 4KB requests in flight (achieved peak Infiniband throughput with rsockets is only 28Gbps instead of the full 40Gbps).

The addition of the network would increase per cohort latency, as each cohort would now wait for the backend to respond creating bubbles in the Rhythm pipeline. Assuming a cohort of size 4096 requests each of size 4K, the time taken to transfer 16MB of data at peak Infiniband throughput is 5 ms. This latency can easily be
Figure 3.1: Throughput for different cohort sizes and different degrees of parallelism for the CPU and the GPU. Parallelism for the GPU indicates the width of the Rhythm pipeline and denotes number of threads for the CPU.

hidden by having a wider pipeline, or having more cohorts in flight. Figure 3.1 shows the measured throughput of the Tesla A configuration for different degrees of parallelism.

Increasing the width of the pipeline increases throughput, and larger cohort sizes provide higher throughputs. This is expected as for smaller cohort sizes, there are not enough threads running to hide memory latency, and the GPU remains underutilized. The throughput saturates beyond 8 request cohorts in flight, because the system becomes bound by the PCIE 2.0 bandwidth. We examine the PCIE bus transactions in more detail.

Table 3.3 shows the KB/request transferred in each direction over the PCIE bus. Our experimental platform connects the Infiniband card and the Tesla accelerator over the same PCIE 2.0 port using a PLX switch, therefore, both the network and
Table 3.3: Rhythm Banking PCIE transactions

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Backend Requests</th>
<th>Dev to Host KB/req</th>
<th>Host to Dev KB/req</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>2</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>account summary</td>
<td>1</td>
<td>10</td>
<td>34</td>
</tr>
<tr>
<td>add payee</td>
<td>0</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>bill pay</td>
<td>1</td>
<td>10</td>
<td>34</td>
</tr>
<tr>
<td>bill pay status output</td>
<td>1</td>
<td>10</td>
<td>34</td>
</tr>
<tr>
<td>change profile</td>
<td>1</td>
<td>10</td>
<td>34</td>
</tr>
<tr>
<td>check detail html</td>
<td>1</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>order check</td>
<td>1</td>
<td>10</td>
<td>34</td>
</tr>
<tr>
<td>place check order</td>
<td>1</td>
<td>10</td>
<td>34</td>
</tr>
<tr>
<td>post payee</td>
<td>1</td>
<td>10</td>
<td>66</td>
</tr>
<tr>
<td>post transfer</td>
<td>1</td>
<td>10</td>
<td>34</td>
</tr>
<tr>
<td>profile</td>
<td>1</td>
<td>10</td>
<td>66</td>
</tr>
<tr>
<td>transfer</td>
<td>1</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>logout</td>
<td>0</td>
<td>2</td>
<td>64</td>
</tr>
</tbody>
</table>

GPU share the same 16 lanes. PCIE 2.0 x16 supports a peak throughput of 6.4 GBps in either direction. The Device to Host transactions are computed as the sum of the backend request from the accelerator to the host (1 KB), the host to the NIC (1 KB) and the final response (8 KB - 64 KB). The Host to Device transactions are computed as the sum of request buffers (2 KB), the backend response from the NIC to the host (4 KB), the backend response from the host to the accelerator (4 KB). Login has multiple requests to the backend, so Host to Device bandwidth consumption dominates Device to Host bandwidth.

Figure 3.2 shows the achieved throughput for the different request types for Tesla A, and the throughput bounded by the PCIE 2.0 bus for a cohort size of 2048 and 16 cohorts in flight. We can see that most of the requests achieve within 80% to 95% of peak PCIE 2.0 bandwidth. The minor difference arises because many PCIE transactions do not use the full 6.4 GBps bus bandwidth.

We measure the 95% percentile latency across request types for different cohort sizes for 8 cohorts in flight, since the system essentially becomes PCIE bound beyond
that, increasing latency without any gains in throughput. Looking at Figure 3.3, the 95% latency for a cohort size of 4096 is not acceptable. A cohort size of 2048 is more acceptable, with only profile and logout having 95% latencies over 100 ms. From Figure 3.1, we can see that the Tesla card achieves higher throughputs than the quad-core Xeon for cohort sizes of 2048 and 1024. These latencies were computed by taking

**Figure 3.2:** PCIe 2.0 Limitations in Tesla A for various request types

**Figure 3.3:** 95% latency for different request types for Tesla A for different cohort sizes, for 8 cohorts in flight.
the highest throughput for each request type feasible within accelerator memory limitations. This is why logout has a lower 95% latency for a cohort size of 4096 as compared to 2048, as lower cohort sizes can sustain higher parallelism. Even with PCIE 2.0 limitations, Rhythm on Tesla cards outperforms the Xeon, demonstrating the applicability of accelerators to our workload.

Platform limitations created by the PLX switch and the PCIE 2.0 interface of the Tesla card do not allow us to fully explore the potential of Rhythm. We next look at a more idealized prototype on a single machine, which removes network limitations and uses a more powerful PCIE 3.0 accelerator. We explore the server design space by progressively removing bottlenecks, and show the benefits on Rhythm for high throughput servers.

3.3 Titan: Overall Results

Rhythm is designed as a free-flowing pipeline that serves to maximally utilize the underlying accelerator hardware to achieve optimal efficiency. This section presents our experimental results that confirm the expected limitations of current system bottlenecks on throughput. We then show how removing these bottlenecks enables Rhythm on a GPU to operate at high throughput with high efficiency. We also demonstrate that replicating general purpose cores to achieve high throughput cannot match the efficiency of Rhythm on today’s GPUs. However, we caution that our results represent an initial exploration of the overall server design space primarily to gauge where Rhythm sits with respect to potential alternative platforms. A more comprehensive study would account for many differences such as technology node, different accelerators, etc. Nonetheless, our results point towards the design of future data parallel accelerators specialized for server workloads.

Table 3.4 shows the throughput, latency, wall and dynamic power of the banking workload for our various modeled platforms. More worker threads is always more
Table 3.4: SPECWeb Banking Experimental Results. *Titan C Latency is for transposed response

<table>
<thead>
<tr>
<th>Platform</th>
<th>Power (Watts)</th>
<th>Latency (ms)</th>
<th>Throughput (KReqs/s)</th>
<th>Reqs/Joule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Idle Wall Dynamic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i5 1 worker</td>
<td>47 76 20</td>
<td>0.016</td>
<td>75</td>
<td>972</td>
</tr>
<tr>
<td>Core i5 4 workers</td>
<td>47 98 51</td>
<td>0.016</td>
<td>282</td>
<td>2447</td>
</tr>
<tr>
<td>Core i7 4 workers</td>
<td>45 147 102</td>
<td>0.014</td>
<td>331</td>
<td>1901</td>
</tr>
<tr>
<td>Core i7 8 workers</td>
<td>45 156 111</td>
<td>0.014</td>
<td>377</td>
<td>2042</td>
</tr>
<tr>
<td>ARM a9 1 worker</td>
<td>2 3.4 1.4</td>
<td>0.176</td>
<td>8</td>
<td>1672</td>
</tr>
<tr>
<td>ARM a9 2 workers</td>
<td>2 4.5 2.5</td>
<td>0.176</td>
<td>16</td>
<td>2683</td>
</tr>
<tr>
<td>Titan A</td>
<td>74 226 152</td>
<td>86</td>
<td>398</td>
<td>1469</td>
</tr>
<tr>
<td>Titan B</td>
<td>74 306 232</td>
<td>24</td>
<td>1535</td>
<td>3329</td>
</tr>
<tr>
<td>Titan C</td>
<td>74 285 211</td>
<td>10*</td>
<td>3082</td>
<td>9070</td>
</tr>
</tbody>
</table>

beneficial for the general purpose cores since it amortizes the fixed costs associated with powering on the chip. For our evaluation we only consider the higher worker threads since they represent the best operating point for each platform.

To gain better insight into the results we utilize a throughput-efficiency plot that normalizes throughput to the Core i7 eight worker threads and efficiency to the ARM A9 two worker threads. Figure 3.4 shows the throughput-efficiency for the various platforms. Considering both wall power (Figure 3.4) and dynamic power (Figure 3.5), the Core i5 is more power efficient than the i7, with efficiencies comparable to the ARM, while delivering 75% of the i7’s throughput. On the other hand, the ARM achieves only 4% of the i7’s throughput, and 6% of the i5’s throughput. The results also show that Titan A performs poorly in terms of efficiency, and provides only marginal throughput improvements. In contrast, Titan C provides massive gains in both throughput and power efficiency, processing ~ 1.5× more requests per Joule compared to the ARM and delivering 7× more requests per second compared to the Core i7. Titan B provides more than 4× the throughput of the i7, though at 91% dynamic efficiency and 124% wall efficiency of the ARM.

The Core i7 and i5 both exhibit low response latency, and even the ARM chip manages latencies of a few hundred microseconds. Titan A exhibits high response
latencies, rendering it unusable for real server applications. Titan B and C perform relatively well, with latencies in 10s of milliseconds. We also measured the 99th percentile latency for our workloads, however it did not differ substantially from the average latency. The 99th percentile latency for different request types varies from 18%-40% of the average latency for Titan B, and varies from 7%-34% of the average for Titan C. These latencies are tolerable [18], and expected, since *Rhythm* sacrifices latency to achieve massive gains in throughput and efficiency. We now evaluate the nature of each Titan platform and their potential as exposed by *Rhythm*. 

*Figure 3.4: Throughput-Efficiency for Wall Power. Throughput (y-axis) is normalized to Core i7 8 workers and efficiency (x-axis) to ARM A9 2 workers. The shaded region represents the desired operating range.*
Figure 3.5: Throughput-Efficiency for Dynamic Power. Throughput (y-axis) is normalized to Core i7 8 workers and efficiency (x-axis) to ARM A9 2 workers. The shaded region represents the desired operating range.

3.4 Titan A: Emulated Remote Backend

Running with an emulated remote backend involves copies over the PCI Express bus for requests to, and responses from the backend. On current platforms, this limits overall throughput. *Rhythm* transfers 1KB for the request buffers, 1KB for the backend request, 4KB for the backend response and 26.4KB on an average for the response over the PCIe bus. We can calculate the throughput bound of the PCIe 3.0 bus by taking the ratio of the peak bandwidth (12GB/s) and the data transferred per request. Figure 3.6 shows the achieved throughput and throughput bounded by
the available PCIe 3.0 bandwidth for the different request types. We can see that all requests achieve throughputs ranging from 83% to 95% of the PCI bounds. The minor difference between the two is expected since we transfer data in smaller chunks, which does not allow us to reach peak PCIe bandwidth. We can see that Rhythm on Titan A is primarily limited by the PCIe 3.0 bandwidth, which creates a structural hazard in the Rhythm pipeline, leading to stalls and a loss in power efficiency.

A potential enabling trend is the PCIe 4.0 standard, which doubles usable bandwidth to 24 GB/s. This could increase Titan A’s throughput to 864K reqs/s and a commensurate increase in efficiency that may bring it near the ARM A9’s efficiency (depending on the PCIe 4.0 power). However, even at 25 GB/s, the PCIe bus is still a bottleneck for Rhythm on a Titan.

3.5 Titan B: Integrated NIC and Device Backend

Titan B increases Rhythm’s average throughput to more than 4× that of the core i7 achieving more than 1.5M reqs/sec, at 91% dynamic efficiency and 124% wall efficiency of the ARM chip.

We note that Titan B’s throughput and efficiency could improve if we used a
better response padding method. Many request types incur significant overhead due to excessive padding since their total response size is just beyond one power of two and we simply round up to the next power of two. This introduces exponentially more overhead for transposes for larger response sizes. Cross-referencing response sizes in Table 3.2 with dynamic efficiency in Figure 3.7 for different requests, we observe that for small responses (i.e., login) or where Rhythm uses a response buffer close to the size of the original response (e.g., change_profile and transfer), Titan B achieves throughput 3.5×-5× higher than the core i7, with dynamic efficiencies

**Figure 3.7:** Throughput-Efficiency for different request types on Titan B (dynamic power). Rhythm buffer sizes that are close to required sizes perform well (shaded area represents the desired operating range).
of 105% to 120% of the ARM, showing room for further optimization. We further analyze this difference and observe that the response transpose takes up a significant fraction of device time, and creates bubbles in the Rhythm pipeline. Titan C removes this limitation.

### 3.6 Titan C: Increasing Device Utilization

Titan C tries to push the Rhythm pipeline to its limits by increasing GPU utilization and offloading the response transpose. With these optimizations, Titan C achieves more than 3M reqs/sec on an average, or more than 8× the throughput of the Core i7. Increasing GPU utilization amortizes the fixed power overheads, achieving a dynamic efficiency of more than 2.5× that of the ARM, and a wall efficiency of more than 3.3×. These numbers ignore power to perform the transpose, and as we show in the next section, with an ample power budget for the transpose, Rhythm can still outperform the ARM in efficiency while achieving far greater throughputs.

### 3.7 Scaling Many Core Processors

A natural comparison for achieving high throughput is to scale the number of general purpose cores to match the throughput of Rhythm on both Titan B and Titan C. We use single thread throughput for the general purpose cores and idealistically assume linear performance scaling. The scaled systems incur additional uncore overhead to support scaling, such as additional chip I/Os, on chip interconnect, additional memory controllers and memory, etc. We use the i5 for scaling instead of the i7 due to its higher dynamic efficiency (Table 5.3). Based on our measurements, and other studies [62], we assume a dynamic power of 1W per ARM core and 10W per i5 core. The power available for the uncore overhead is the difference between the idealized scaled system’s power and the Titan platform’s power.
With respect to Titan B (dynamic power), we need 192 ARM cores and 21 i5 cores to match throughput, requiring 192W and 210W, respectively. Titan B uses 232W, leaving only 40W for the ARM (21%) or 22W (10%) for the i5 available for uncore scaling overhead. Compared to Titan C, we need 385 ARM cores and 41 i5 cores to match throughput. This results in 385W for the ARM system and 410W for the i5 system, and Titan C has more than 170W in which to implement the transpose operation and still outperform the scaled systems.

Given that the Titan-based systems also have room to reduce overall power consumption (e.g., lower power DRAM, eliminate GPU specific features, etc.) it appears difficult for simple replicated designs to match the overall throughput and throughput/watt of Rhythm on GPU-style accelerators.

3.8 System Resource Requirements

**Network Bandwidth** Using the average value of the response size of the SPECWeb Banking workload (Table 3.2), and the average number of backend responses, we can easily calculate the network bandwidth requirements of our Titan platforms. At an average throughput of 398K reqs/s, Titan A requires a network bandwidth of 67 Gbps, Titan B requires 258 Gbps and Titan C requires 517 Gbps. All of these numbers assume raw uncompressed data. Most modern web browsers support compression and research has demonstrated more than 80% compression of HTML content in pages for popular websites [65]. A compression ratio of 80% means that Titan C can easily be operated on a 100Gbps link, already defined by the IEEE 802.3bj standard [47].

**PCIE Bandwidth Optimizations** Currently, a majority of PCIE bandwidth consumption is wasted due to the uncompressed cohorts being transferred across the PCIE bus. We allocate 8KB-64KB for the client response buffers, 1KB for the backend request buffers, and 4KB for the backend response buffers, and for most
requests only a fraction of these buffers is utilized. On average only 59% of the response buffers are used (Table 3.2), 93% of the backend request buffers and 96% of the backend response buffers are empty. A cohort can easily be compressed by calculating the prefix sum of individual request lengths, giving locations for each request in the compressed buffer. A warp per request can then be used to copy over the request data into its corresponding location, achieving coalescing as well. Implementing this optimizations would allow the PCIE 4.0 bus to sustain 1.5M reqs/s, easily usable for Titan B.

**Memory Capacity** We are currently limited by the memory on the device. For our experiments, we emulate 16M active sessions on the Titan, and at 40B per session, this requires 640MB of memory. The session array is implemented as a hash table using random insertion, and we allocate memory for 64M sessions to reduce the chance of a collision to 25%, but this requires 2.5GB of device memory. Since all memory is preallocated in pools, we also need to allocate enough memory for the process phase, backend data, request buffers, response buffers and transpose buffers to avoid structural hazards in the pipeline. The memory required for buffers increases linearly with cohort size, therefore, we are limited to 8 cohorts in flight of size 4096 requests each on the GTX Titan.

3.9 Miscellaneous

**CPU based SIMD implementations** Web service workloads and data parallel hardware are a great match, and *Rhythm* provides a way to bring the two together by exploiting similarity amongst requests to enable SIMD processing. The GPU’s throughput oriented SIMT architecture provides a good platform to test this approach. A SIMD based implementation on current CPUs would provide a useful data point in this design space as well.

**Cohort Size sensitivity** We performed experiments on the Titan platform for
cohort sizes ranging from 256 to 8192. Larger cohort sizes are better for throughput since they allow more work to be launched on the GPU, however, they require more memory. Larger cohort sizes also impact response latency, since it takes more time to form a cohort. However, for arrival rates of the order of a million reqs/sec, cohort formation times are negligible.

**Parser divergence** Our current experiments run the same type of requests on the parser, however, this reduces control divergence in the parser. In a real world system, multiple request types would arrive at the parser, increasing control divergence and reducing single parser throughput. We measured parser latency for a real Specweb Banking Trace containing a mix of requests and images. On an average, the parser takes 556us including the request buffer transpose, giving a throughput of 7.4M reqs/sec for a cohort size of 4096. Therefore, the parser is fast enough even when it is processing cohorts of different types. The *Rhythm* design also allows for multiple parsers to be launched concurrently, and for higher throughputs, this would further help in hiding parser latency.

**HyperQ** We performed our experiments on a NVIDIA GTX690 as well, however, a single work queue between the host and device created false dependencies among process kernels, limiting throughput. The GTX Titan supports 32 simultaneous work queues (HyperQ), allowing for much higher throughput and GPU utilization. *Rhythm* can expose significant concurrency and the hardware must be capable of exploiting it. Emulating future platforms with integrated high bandwidth devices exposed the benefit of having HyperQ scheduling on the GPU.

### 3.10 Related Work

This work touches on topics across a broad spectrum of computer systems related topics, including, but not limited to: server design, operating system design and implementation, system and processor architecture, and data layout optimizations.
For brevity, we focus on some of the most closely related research.

At the application mapping level, several researchers have explored general purpose use of GPUs [13, 42, 16] and mapping non-traditional workloads onto GPUs, such as MIMD programs [19], database queries (e.g., [6, 38, 99], etc.) and memcached [43], and software routers [40]. Other work explores the potential opportunities created by delaying server requests for either improved memory hierarchy performance [61] or energy management [60, 70], and allocating compute resources dynamically based on load in staged servers [98].

Recent microarchitecture work [93] shows the benefits of data-triggered threads and methods for eliminating redundant computation, while other work (MMT [66] and Thread Fusion [35]) shows the benefit of exploiting identical instructions in SMT processors to remove redundant operations (i.e., fetch, execute, etc.). Our work focuses on coarser grain identification of cohorts and targets different applications, but may benefit from the addition of these methods. STREX [5] improves transaction processing by aligning instructions to reduce cache misses. Other work [9] explores data similarity and memory hierarchies that can merge identical content used by different cores into a single cache line. Thread scheduling on GPUs has been addressed in the context of reducing the impact of control divergence [92, 71, 29, 30]. More generally, several researchers are exploring processor design for data center workloads [14, 25, 53, 56, 62, 63, 67, 72].

3.11 Conclusion

A dramatic increase in the number of data centers in the last decade and their multi-megawatt power budgets has sharply brought into focus the energy economics of web services. We propose the use of data parallel accelerators and a software architecture called *Rhythm* to address throughput and efficiency demands of future server workloads. *Rhythm* is based on the insight that maximal power efficiency
of an accelerator comes from maximizing utilization since it amortizes fixed system costs. We show Rhythm achieves throughput 4× to 8× of an 8 thread core i7 at efficiencies (requests/joule) comparable to or higher than a dual core ARM Cortex A9. This work serves as a milestone that demonstrates that Web Servers and data parallel accelerators are a great match. Even on current generation GPU hardware, Rhythm outperforms both the x86 and ARM architectures, and is just a first step in exploring an exciting and varied design space.
Exploiting Accelerators for Efficient High Dimensional Similarity Search

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 [68]. In web search it can be used to identify the matching pages to a given search query [12]. 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 [86][10]. It can be used in protein sequencing to identify the similarity between a new sequence and a database of known proteins [33][64].

Similarity search is also used in a large number of web services, for example, movie recommendation systems (finding a movie similar to a list of already watched movies), behavioral ad targeting (displaying an ad similar to a user’s interests) and personalized news feeds (identifying news articles similar to previously read items)
These services are powered by datacenters; colossal clusters that have hundreds of thousands of machines. The Internet has more than a trillion pages [36], and a simple web search goes through billions of them in a few milliseconds. Google alone processes more than 100 billion queries per month (about 38,000 per second) across these pages [37]. Netflix identifies movie recommendations across a database of more than 5 billion ratings [73].

These massive scale out factors make single server capital and operational costs all the more important, and identifying the most suitable server architecture for performing similarity search essential. Increasing throughput per machine requires fewer machines, reducing capital costs, and improving efficiency per machine improves operational costs. Even small scale cluster deployments provisioned using a commercial vendor like Amazon’s EC2 [3] can significantly benefit from improved server designs, as fewer machines are needed for the same workload.

Accelerators are generally considered suitable for scientific or HPC workloads, and either too expensive or power hungry for traditional web service workloads like search. We argue 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 [61] 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 for an incoming query stream by minimizing the total cost of ownership. The primary contributions of this chapter 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. (Section 4.3)

- A model for evaluating the capital and operational costs of cluster design for similarity search using accelerators. (Section 4.5)

- A methodology for comparing the cost of ownership of two different CPU based configurations, or a mix of CPU and accelerator based cluster configurations for similarity search. (Section 4.5)

4.1 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.

Numerous solutions and methodologies exist for solving the similarity search problem. The simplest solution is a brute force linear search which searches through all points and takes $O(ND)$ time. There are several space partitioning methods such as kd-trees which bound the size of the problem by performing lookups over only a
subset of objects. However, these partitioning methods do not scale well to higher dimensions, and effectively deteriorate to linear search [97]. Searching through billions of points across thousands of queries requires significant computational horsepower, making the problem of efficient similarity search all the more challenging.

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/image retrieval in an application like web search using a bag of words model, $N$ is usually 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 $\ll D$, the document vectors are sparse; and an inverted list is used to efficiently store and access the index [12].

In many cases, an approximate solution to the nearest neighbor problem is equally useful [48]. Locality Sensitive Hashing (LSH) is a more recent approach for approximate similarity search [32][89]. LSH is essentially a clustering technique that partitions the input space into buckets based on a family of hash functions. Collisions represent points that lie closer to each other and therefore belong to the same cluster. A query is mapped to its cluster using the same hash functions, and a simple linear search across the collisions yields the most probable matches. Approximate similarity search can give higher performance gains at the expense of quality [21]. We discuss our work in the context of exhaustive search, but the ideas are broadly applicable and can be 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. LSH requires a significant amount of state to store the hash tables for acceptable search quality. We note that hashing a query cohort in LSH is a SpMM operation, and an optimized SpMM algorithm would be beneficial.
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 $\alpha$ and find the $k(k \ll N)$ 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 $\times$ 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 chapter can be easily applied to any similarity search problem with the above properties.
4.2 Baseline Architecture

We assume a baseline architecture similar to contemporary designs [54], consisting of an aggregator node \(^1\) and multiple service nodes (Figure 4.1). 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 (\(\alpha\)), 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.

4.3 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 \(n_{\text{machine}}\) 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

\(^1\) We use the terms node and machine synonymously in this work.
machine is then responsible for generating the $k$ highest similarity matches for each query in the cohort across these $n_{machine}$ documents (Figure 4.2).

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 [68]. We note that $A$ stored in the CSR format is similar to an inverted list [12], 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 $S$,

$$S = Q \times A$$

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 [51] and NVIDIA’s cuSPARSE [17], 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 4.3), and briefly describe them now.
4.3.1 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 inner loop accumulates the rows of $A$
Algorithm 1 Sparse Matrix Matrix Multiplication for $S = Q \times A$

1: $\leftarrow$ clear all values in $S$
2: $S \leftarrow 0$
3: $\leftarrow$ in parallel
4: $\textbf{for } i \leftarrow 1 \textbf{ to } C \textbf{ do }$
5: $\quad \textbf{for } j \textbf{ where } Q(i, j) \neq 0 \textbf{ do }$
6: $\quad \quad \textbf{in parallel on accelerator}$
7: $\quad \quad \textbf{for } k \textbf{ where } A(j, k) \neq 0 \textbf{ do }$
8: $\quad \quad \quad S(i, k) \leftarrow S(i, k) + Q(i, j) \times A(j, k)$
9: $\quad \textbf{end for}$
10: $\textbf{end for}$
11: $\textbf{end for}$

Figure 4.4: Our SpMM approach for $S = Q \times A$. The shaded rows of $A$ are accumulated into the values array of $S$.

We merge rows of $A$ corresponding to the 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 $C \times n_{\text{machine}}$ 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 \((\text{insertion})\). For an \textit{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 4.4 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 \(a\) (1) in the values array. When the next nonzero of \(Q\) is processed, \(e\) and \(g\) are insertions, and \(f\) is an \textit{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.

4.3.2 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 \textit{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 \(k << n_{\text{machine}}\), 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 \([81]\) to calculate the minimum in parallel on the accelerator rather than a binary heap.
4.4 Architecture specific modifications

We performed a large number of optimizations for our SpMM and top-k implementations by analyzing program performance using VTune [52] on the host, and the NVIDIA Visual Profiler [77] on the accelerator. Instead of processing all documents on a machine at once, we use the well known technique of tiling. We divide the documents or shard into smaller tiles of size $\mu$ each, perform 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:

1. It improves cache locality and decreases cache contention on the host across a cohort of queries, as tiling allows the working set to fit in the LLC.

2. It allows us to work with limited memory on the accelerator.

3. It 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.

4.4.1 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 parallelism and memory level parallelism, however, more threads mean more queries in flight and more state, which creates cache contention. Also, 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 attempted to optimize the host further by manually adding AVX instructions. However, the switch between sequential code based on conditionals and SIMD code negated the benefits of manual SIMD on the host. Instead, we rely on the Intel ICC compiler to perform auto-vectorization.

4.4.2 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 4.5 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 partitioning step on the accelerator. The second partition operation runs on the accelerator, and generates the final microtiles.

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.

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 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.
4.5 A Model for Cost Efficient Cluster Design

This work presents a framework to implement high dimensional similarity search across billions or trillions 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 ($\alpha$) and deadline ($T$).

To calculate the cost of performing the search for a given $\alpha$ 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

$$total = \$\text{capital} + \$\text{operational}$$

(4.2)

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, $n_{\text{machine}}$ refers to the number of documents processed on a machine. The number of machines needed to process a query cohort is

$$\text{machinesPerCohort} = \frac{N}{n_{\text{machine}}}$$

(4.3)

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

$$cohorts = \frac{\alpha}{C} \times T \quad (4.4)$$

To meet deadlines for all queries, all these cohorts need to be processed. Using Equation 4.3, the capital cost to process all pending cohorts is

$$\$capital = cohorts \times N \times \frac{\$machine}{n_{machine}} \quad (4.5)$$

where $\$machine$ is the cost per machine. Each machine consists of a host processor and an accelerator (Figure 4.6). 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,

$$\$capital = \frac{\alpha NT}{C} \times \frac{\$fixed + \$cpu + \$acc}{n_{cpu} + n_{acc}} \quad (4.6)$$

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

$$\text{energyPerCohort} = \text{machinesPerCohort} \times T \times (P_{\text{fixed}} + U_{\text{cpu}}P_{\text{cpu}} + U_{\text{acc}}P_{\text{acc}}) \quad (4.7)$$

where $P_{\text{fixed}}$ is the fixed or idle power consumption of a machine, $P_{\text{cpu}}$ is the power consumed by the host to process $n_{\text{cpu}}$ documents, and $P_{\text{acc}}$ is the power consumed by the accelerator to process $n_{\text{acc}}$ documents. $U_{\text{cpu}}$ and $U_{\text{acc}}$ 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 $T$. $n_{\text{acc}}$ is limited by the memory on the accelerator, and the if the deadline is higher than the time it takes to process $n_{\text{acc}}$ 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

$$\$_{\text{operational}} = \text{lifetimeCohorts} \times \text{energyPerCohort} \times \text{energyCost} \quad (4.8)$$

where $\text{lifetimeCohorts}$ 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 $\alpha$, 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 [7]. Using Equations 4.3, 4.7 and 4.8, the total operational cost of the workload is,

$$\$_{\text{operational}} = \frac{\alpha NT}{C} \times 2.35 \times \frac{P_{\text{fixed}} + U_{\text{cpu}}P_{\text{cpu}} + U_{\text{acc}}P_{\text{acc}}}{n_{\text{cpu}} + n_{\text{acc}}} \quad (4.9)$$

All of the parameters in Equations 4.6 and 4.9 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,

$$ W = C \times (n_{cpu} + n_{acc}) \quad (4.10) $$

We also define,

$$ \$system = \$fixed + \$cpu \quad (4.11) $$

and,

$$ P_{system} = P_{fixed} + U_{cpu}P_{cpu} \quad (4.12) $$

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

$$ \$total = \frac{\alpha NT}{W}(\$system + \$acc + 2.35(P_{system} + U_{acc}P_{acc})) \quad (4.13) $$

This cost can be easily broken down and understood. By application of Little’s Law, $\alpha NT$ 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 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, and we now look at specific examples of configurations and our approach for evaluating them.

### 4.5.1 Comparing two CPU configurations

A common use case can be to compare two different CPU configurations, for example, two different Xeon platforms, or comparison of a Xeon based cluster to an ARM based cluster. These are CPU only configurations, so $\$acc$, $P_{acc}$, and $n_{acc}$ are all zero. For
a given arrival rate and deadline, designating the two configurations as 1 and 2, the ratio of their total cost can be written as,

\[
\frac{\$_{\text{total}}(1)}{\$_{\text{total}}(2)} = \frac{W_2}{W_1} \times \frac{\$_{\text{system}}(1) + 2.35P_{\text{system}}(1)}{\$_{\text{system}}(2) + 2.35P_{\text{system}}(2)} 
\] (4.14)

where \(W_1\) and \(W_2\) are the work done per machine by Configuration 1 and 2 respectively. Our goal is to determine if configuration 1 is cheaper than configuration 2, or

\[
\frac{\$_{\text{total}}(1)}{\$_{\text{total}}(2)} < 1 
\] (4.15)

or,

\[
\$_{\text{system}}(2) - \frac{W_2}{W_1} \$_{\text{system}}(1) + 2.35(P_{\text{system}}(2) - \frac{W_2}{W_1}P_{\text{system}}(1)) > 0 
\] (4.16)

\(W\) for a system can be determined by picking the cohort size \(C\) and \(n_{\text{cpu}} + n_{\text{acc}}\) giving the highest work done (Equation 4.10) within the given deadline \(T\). We pick \(W_1\) and \(W_2\) based on the optimal configuration for each CPU type for the given arrival rate and deadline. This essentially gives us a 2-dimensional space for different values of \$_{\text{system}}(1)\) and \$_{\text{system}}(2)\), and this inequality gives us a region for which configuration 1 is cheaper than configuration 2.

4.5.2 Augmenting existing servers with accelerators

Augmenting existing servers with accelerators can be advantageous due to the accelerators limited applicability, and the higher utility of general purpose cores to run other workloads. Augmenting existing servers with accelerators allows each machine to perform more work, amortizing fixed costs per machine. Many server machines have free PCI-E slots, or more can be added using a simple PCIE switch (like the PLX PEX 8747). Assuming Configuration 1 is the accelerator augmented design,
and Configuration 2 is the CPU only cluster,

\[
\frac{\$_{\text{total}}(1)}{\$_{\text{total}}(2)} = \frac{W2 \$_{\text{system}} + \$_{\text{acc}} + 2.35(P_{\text{system}}(1) + U_{\text{acc}}P_{\text{acc}})}{\$_{\text{system}} + 2.35P_{\text{system}}(2)}
\]

(4.17)

where \(P_{\text{system}}(1)\) also includes the idle power consumed by the accelerator. We are adding an accelerator to the same machine, so \(\$_{\text{system}}\) remains the same in both the numerator and the denominator. Identifying the values of \(\$_{\text{system}}\) and \(\$_{\text{acc}}\) for which the accelerator augmented system is cheaper than a CPU only cluster gives us the inequality

\[
\$_{\text{system}}(1) - \frac{W2}{W1}\$_{\text{acc}} + 2.35(P_{\text{system}}(2) - \frac{W2}{W1}(P_{\text{system}}(1) + U_{\text{acc}}P_{\text{acc}})) > 0
\]

(4.18)

An interesting conclusion from Equation 4.18 is, even if the cost of the CPU cluster is 0, i.e. \(\$_{\text{system}} = 0\), augmenting is still a good idea if,

\[
\$_{\text{acc}} < 2.35\left(\frac{W1}{W2}P_{\text{system}}(2) - (P_{\text{system}}(1) + U_{\text{acc}} \times P_{\text{acc}})\right)
\]

(4.19)

or, the cost of the accelerator is less than power savings achieved from the reduced number of machines needed with the augment.

4.5.3 Accelerator only clusters

Adding accelerators to servers is a good idea, as it minimizes changes to current datacenters. However, if the accelerators deliver higher performance/$ than existing server class processors like the Xeon, it should be advantageous to reduce system costs by having an accelerator-only cluster, and the only function of the host is to drive the accelerator. The host can be a single core processor requiring minimal RAM and storage like an Atom based SoC , and simply provides a thin wrapper over the accelerator to communicate with the network. Assuming Configuration 1 is the SoC + accelerator cluster, and Configuration 2 is the CPU only cluster,

\[
\frac{\$_{\text{total}}(1)}{\$_{\text{total}}(2)} = \frac{W2}{W1} \times \frac{\$_{\text{system}}(1) + \$_{\text{acc}} + 2.35(P_{\text{system}}(1) + U_{\text{acc}}P_{\text{acc}})}{\$_{\text{system}}(2) + 2.35P_{\text{system}}(2)}
\]

(4.20)
The cost of the SoC, $s_{\text{system}}(1)$ can be fixed, as it is not involved in processing any documents. $P_{\text{system}}(1)$ is the sum of the idle power of the SoC and the accelerator. The range of dollars values of the CPU system, $s_{\text{system}}(2)$ and the accelerator, $s_{\text{acc}}$, can be determined from the inequality,

$$\frac{W_2}{W_1} s_{\text{system}}(2) - 2.35 \frac{W_2}{W_1} s_{\text{acc}} + 2.35 P_{\text{system}}(2) - \frac{W_2}{W_1} (s_{\text{system}}(1) + 2.35 (P_{\text{system}}(1) + U_{\text{acc}} P_{\text{acc}})) > 0$$

(4.21)

Similarly, by varying $s_{\text{system}}(2)$ and $s_{\text{acc}}$, we can explore the corresponding dollar savings given by $\frac{s_{\text{total}}(1)}{s_{\text{total}}(2)}$. An important aspect of this model is the distribution of documents between the host and the accelerator, and we now propose a method to determine this partitioning.

### 4.5.4 Computing $n_{\text{cpu}}$ and $n_{\text{acc}}$

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 $\alpha$ 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 using the following approach.

1. Experimentally measure the time taken to process a query cohort of size $C$ across a tile of size $\mu$ for all tiles in the document collection on a given processor.

2. Compute the number of tiles of size $\mu$ that can fit in available host or accelerator memory, and use this to compute an initial value of $n_{\text{machine}}$.

3. Compute the number of machines needed to process a cohort using Equation 4.3.
4. Distribute the tiles across these machines.

5. Compute the latency of the cohort on a machine as the sum of the latencies of
   the tiles on a machine using values measured in Step 1.

6. Compute the time per cohort as the maximum time taken across all the ma-
   chines the cohort runs on, since the aggregator waits for the results from each
   service node.

7. Calculate the percentage of cohorts that finish within the available processing
   time $T'$.

8. If this value is less than 95%, iteratively reduce the number of tiles processed
   per machine and go back to Step 3.

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 dead-
line. For the accelerator, we bound the number of documents that can be processed
by the available memory on the accelerator, to avoid PCIe transfers and bottlenecks.

We use the above model to compare the cost of performing similarity search
across different hardware platforms. We identify the values of $n_{cpu}$, $n_{acc}$, $\mu_{cpu}$, $\mu_{acc}$
and $C$ that minimize $\$_{total}$ for a given platform, and compare these costs for different
arrival rates and service times.

4.6 Related Work

Total cost of ownership models for datacenters exist in many different forms, focusing
on the relationship between capital cost and operational cost [7][59]. These models
focus on the datacenter as a whole, assuming a fixed number of machines and cost per
machine. To the best of our knowledge, this is the first work that includes workload
characteristics and constraints into creating a more balanced and energy efficient server design.

Sparse matrices have been used for text search before by Goharian et al. [34]. 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. 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. The values array essentially contains the list of documents for every term stored contiguously, and the columns array contains the indexes for these documents respectively. We argue for processing a query cohort at a time, generating more work for the hardware, amortizing fixed system costs, and resulting in higher energy efficiency. We also focus on identifying the best hardware platform and cohort size to sustain a given arrival rate under a given deadline, across a scaled out system.

LSH is one of the most popular approximate search techniques due to its low query latency and provable quality guarantees. However, LSH requires storing large amounts of state to achieve an acceptable quality of results. Our work focuses on exhaustive search, and our SpMM algorithms can easily be applied to accelerate LSH as well while projecting the query cohort onto a lower dimensional space. Techniques like PageRank [12] rely on a static ordering of documents prior to search based on the in-degree of a page, and do not provide strong quality guarantees.

4.7 Summary

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. This chapter presents optimized algorithms for both the host and the accelerator
which are faster than existing MKL and cuSPARSE routines. We also present a model for the computing the total cost of ownership for similarity search, and a methodology for comparing the cost of different configuration using accelerators.
An Analysis of Text Search on Accelerators

Text search is an example of similarity search, widely used due to its prevalence in Internet search engines. Applications like Google Search process thousands of queries across billions of pages, and text search still remains one of the fundamental ways to mine and seek information. Using the models and algorithms from Chapter 4, we evaluate the applicability of accelerators to a simple text search engine for both low and high arrival rates across billions of documents. Our contributions in this chapter are:

- A demonstration of cohort scheduling for text search under constrained deadlines. Query cohorts reduce cache contention on general purpose cores, and allow for higher occupancy and amortization of fixed costs on the accelerator. (Section 5.2)

- An implementation of text search on NVIDIA GPUs that is more than twice as efficient and delivers more than twice the throughput of the corresponding implementation on Xeon and ARM cores. (Section 5.3)
• An SoC + Accelerator approach that delivers 2.3\times the throughput of a Xeon server while being 75% more energy efficient than an ARM server for an arrival rate of 25,000 queries/s and a deadline of 50 ms. (Section 5.4)

• 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. (Section 5.5)

• A cost evaluation of an alternative cluster design using our SoC + Accelerator approach that is 6\times cheaper and consumes less overall power than a commercial Xeon based server cluster, for an arrival rate of 25,000 queries/s such that 95% of queries finish within 50 ms. (Section 5.5)

• Analysis of the design space comparing different prices of host processors and accelerators, that proves that even with zero cost host processors, retail priced accelerators are still cheaper due to savings on operational costs from higher energy efficiency. (Section 5.5)

We present some preliminary results that confirm the applicability of commercial accelerators to text search, and show that these provide much more cost effective performance than server class processors even under very tight deadlines.

5.1 Methodology

Table 5.1 shows the different platforms we model 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
Table 5.1: Experimental System Platforms

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

represents a low cost accelerator. We use the quad-core Tegra K1 to model 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 4 million 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 [68], which reflects the importance/weight of a word in a given document. There are many different techniques for document scoring, e.g. PageRank [12] for web search, color histograms for images [86], and other machine learning based ranking models. The scoring methodology is used to populate the document and query matrices used in our algorithms. Different scoring approaches would just change the values in these matrices, and the SpMM and Top-k algorithms used would remain the same.

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 4 million pages each. We assume the document and query matrices are given to us, and our results are independent of the scoring methodology used.

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 because the high accuracy of double precision is not needed for identifying similar documents.

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 [96].

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 ($\alpha$) 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 $\mu$ 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 $\mu$, 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 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, 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 pick the value of $\mu$, $C$, 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 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. A tile of 2M documents is around 1.5 GB in size, and transferring this over a PCIE 3.0 interface would take $>100$ ms. Even with PCIE 4.0 having twice the bandwidth, these times are prohibitive. For evaluating the accelerator, we therefore limit $n_{acc}$ only to the documents that can fit in available accelerator memory.

The host and accelerator platforms are not running at peak power for the entire duration of the deadline. To estimate total energy consumed during our workload, we also calculate average latency incurred to process a shard, and sum the energy cost incurred while idle and during processing time.

Our Maxwell card, the GTX 750Ti has just 2GB of available memory, which is insufficient for our purposes. We model a higher memory variant of the Maxwell, 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 [55]. 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
Figure 5.1: Performance of the Xeon for different cohort sizes and tile sizes.

SoC + Titan and SoC + Maxwell in our experiments.

We now look at the performance of the host and accelerator platforms for different cohort sizes and shard sizes. Picking the optimal configuration for each platform will allow us to compare the efficiency and cost of ownership of similarity search across different cluster designs.

5.2 Impact of the Tile Size \( \mu \) 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 Work done/second, and the energy efficiency becomes the Work done/Joule. Figure 5.1 and 5.2 show the performance for the Xeon and the Titan as a function of the tile size \( \mu \) 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 part of the algorithm.

In case of 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 result in fewer overall tiles, reducing partitioning overheads. Larger cohort sizes enable more memory level parallelism, resulting in 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.
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 5.3 and 5.4 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 with...
Figure 5.4: Throughput-efficiency of text search for different processors for cohort sizes ranging from 64-1024. 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).

to the Maxwell reduce performance, delivering 9% more performance than the Xeon, while being only 21% more energy efficient.

For larger cohort sizes, the Titan delivers more performance, while the Maxwell is more energy efficient. For a cohort size of 64 (Figure 5.4), 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 [69]. 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. 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.

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. This provides us ample motivation to consider accelerators for more efficient cluster configurations. However, this analysis is based on throughput calculations based on average query latency. We next look at a practical implementation with a stream of incoming queries and constrained deadlines.

5.4 Adding Constraints

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

- The search operation must finish within a deadline of $T$ ms, such that 95% of queries finish within this time.

- The number of tiles that can be processed on the accelerator is limited by available memory as PCI-E transfers are infeasible within these deadlines.

- Queries arrive as an incoming stream with a rate of $\alpha$ queries/s, therefore forming a cohort incurs batching delay, and the cohort must still finish within the deadline.
Figure 5.5: Optimal platforms for text search for a deadline of 50 ms and $\alpha = 1000$ queries/s. 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).

Figure 5.5 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, and cohort sizes larger than 32 become infeasible in 50 ms. The Maxwell based accelerator only configuration (soc+maxwell) is the most efficient configuration 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 instead of one.

Table 5.2 shows the utilization of the different processors for different arrival rates. The times shown are the fraction of the deadline spent forming cohorts and doing useful work. The fractions don’t add up to 1, as these utilizations are computed
Table 5.2: Utilization for $T = 50$ ms, (a) $\alpha = 1000$ queries/s and (b) $\alpha = 25000$ queries/s. The power is the total power needed to support the cluster.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Cohort formation</th>
<th>$U_{cpu}$</th>
<th>$U_{acc}$</th>
<th>Avg. Power (kW)</th>
<th>Peak Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) $\alpha = 1000$ queries/s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon</td>
<td>0.00</td>
<td>0.60</td>
<td>-</td>
<td>144</td>
<td>158</td>
</tr>
<tr>
<td>Arm</td>
<td>0.16</td>
<td>0.48</td>
<td>-</td>
<td>23</td>
<td>31</td>
</tr>
<tr>
<td>Xeon+Maxwell</td>
<td>0.64</td>
<td>0.26</td>
<td>0.26</td>
<td>52</td>
<td>73</td>
</tr>
<tr>
<td>Xeon+Titan</td>
<td>0.64</td>
<td>0.26</td>
<td>0.26</td>
<td>62</td>
<td>82</td>
</tr>
<tr>
<td>SoC+Maxwell</td>
<td>0.64</td>
<td>-</td>
<td>0.26</td>
<td>14</td>
<td>33</td>
</tr>
<tr>
<td>SoC+Titan</td>
<td>0.64</td>
<td>-</td>
<td>0.26</td>
<td>27</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>(b) $\alpha = 25000$ queries/s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon</td>
<td>0.20</td>
<td>0.58</td>
<td>-</td>
<td>611</td>
<td>661</td>
</tr>
<tr>
<td>Arm</td>
<td>0.10</td>
<td>0.64</td>
<td>-</td>
<td>320</td>
<td>391</td>
</tr>
<tr>
<td>Xeon+Maxwell</td>
<td>0.10</td>
<td>0.74</td>
<td>0.77</td>
<td>419</td>
<td>465</td>
</tr>
<tr>
<td>Xeon+Titan</td>
<td>0.10</td>
<td>0.74</td>
<td>0.75</td>
<td>323</td>
<td>357</td>
</tr>
<tr>
<td>SoC+Maxwell</td>
<td>0.10</td>
<td>-</td>
<td>0.77</td>
<td>218</td>
<td>267</td>
</tr>
<tr>
<td>SoC+Titan</td>
<td>0.10</td>
<td>-</td>
<td>0.75</td>
<td>183</td>
<td>213</td>
</tr>
</tbody>
</table>

based on average latency per cohort, and the remainder of the time is slack needed to sustain 95% latency for the queries.

For an arrival rate of 1000 queries/s, the average power is significantly lower than the peak power for the accelerator cluster due to low utilization of the processors. The accelerators have only 6 GB of memory, and can hold a maximum of 8M documents. For a cohort size of 1, they process these 8M documents in less than 5 ms, and remain idle for 45 ms, leading to low performance and energy efficiency. The accelerators have a limited number of documents to work with due to memory capacity constraints, and 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 5.3), 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 5.6, 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 $2.3 \times$
Table 5.3: Configurations for $T = 50$ ms, (a) $\alpha = 1000$ queries/s and (b) $\alpha = 25000$ queries/s. Each configuration is optimal for that platform giving the highest Work done/second. The total power is the peak power needed to support the cluster.

<table>
<thead>
<tr>
<th>Platform</th>
<th>C</th>
<th>$\mu$ (Kdocs)</th>
<th>Power (Watts)</th>
<th>n (Kdocs)</th>
<th>No. of machines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cpu</td>
<td>acc</td>
<td>fixed</td>
<td>cpu</td>
</tr>
<tr>
<td>Xeon</td>
<td>1</td>
<td>500</td>
<td>-</td>
<td>144</td>
<td>33</td>
</tr>
<tr>
<td>Arm</td>
<td>8</td>
<td>100</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Xeon+Maxwell</td>
<td>32</td>
<td>100</td>
<td>2000</td>
<td>156</td>
<td>32</td>
</tr>
<tr>
<td>Xeon+Titan</td>
<td>32</td>
<td>100</td>
<td>1000</td>
<td>193</td>
<td>32</td>
</tr>
<tr>
<td>SoC+Maxwell</td>
<td>32</td>
<td>-</td>
<td>2000</td>
<td>17</td>
<td>-</td>
</tr>
<tr>
<td>SoC+Titan</td>
<td>32</td>
<td>-</td>
<td>1000</td>
<td>54</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256</td>
<td>100</td>
<td>144</td>
<td>32</td>
</tr>
<tr>
<td>Arm</td>
<td>128</td>
<td>100</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Xeon+Maxwell</td>
<td>128</td>
<td>100</td>
<td>500</td>
<td>156</td>
<td>41</td>
</tr>
<tr>
<td>Xeon+Titan</td>
<td>128</td>
<td>100</td>
<td>2000</td>
<td>193</td>
<td>41</td>
</tr>
<tr>
<td>SoC+Maxwell</td>
<td>128</td>
<td>-</td>
<td>500</td>
<td>17</td>
<td>-</td>
</tr>
<tr>
<td>SoC+Titan</td>
<td>128</td>
<td>-</td>
<td>2000</td>
<td>54</td>
<td>-</td>
</tr>
</tbody>
</table>

the throughput of the Xeon while being 75% more energy efficient than the ARM.

Tables 5.3a and 5.3b 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 95% of queries finish within 50 ms. We observe the following from these values:

- For an arrival rate of 25,000 queries/sec it takes 5.2 ms to form a cohort of size 128 and 10.4 ms to form a cohort of size 256. The processors remain idle during this time, and even with the reduced service times, larger cohort sizes are more optimal.

- For higher arrival rates, the optimal cohort size for the Xeon is 256. Higher arrival rates allow for larger cohorts within the same deadline, and larger cohorts can take advantage of cache locality for the same tile.
• The host benefits from a smaller tile size \((\mu = 500K \text{ or } 100K)\) for its optimal configuration. Smaller tiles fit entirely within the last level cache, reducing the average memory access time.

• The accelerator (Maxwell or Titan) always processes more documents than the host \((n_{acc} > n_{cpu})\) within the given deadline, which shows that the accelerators deliver higher throughput than the host.

• The accelerator always chooses a large tile size \((\mu = 500K \text{ to } 2M)\) for optimal efficiency. Larger tiles allow the accelerator to be fully utilized, and amortize fixed costs by doing more work per kernel launch.

• 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 the number of machines reduces the overall fixed costs, reducing total power required.

• The accelerator based systems require higher peak power at an arrival rate of \(1000\) queries/s, however, due to their larger cohort sizes, their utilization is extremely low, causing much lower average power consumption.

• The accelerators deliver higher throughput than the Xeon and ARM processors (Figure 5.3 and 5.4), therefore, the SoC + accelerator platforms require fewer machines for the same workload.
Accelerators do more work than the host processors for the same arrival rate and within the same deadline constraints, translating into higher throughput and higher throughput/Watt. A natural question for any datacenter architect would be if these increases in throughput and efficiency translate into actual dollar 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.

5.5 Cost of ownership Analysis

We now compare the different platforms based on cost of ownership from models created in Chapter 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 dollars gains for the accelerator based designs. Figure 5.7 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 based cluster is 73\% cheaper, and the SoC + Maxwell based configuration is more than 15× cheaper than the Xeon cluster.

If a 16 cores/32 threads Xeon system can be acquired for $1\text{\(^{th}\)}$ its retail price at $500 (including fixed costs), both accelerator based configurations are still significantly cheaper, with the Xeon + Maxwell being 56\% cheaper, and the SoC + Maxwell more than 3× cheaper. The dollar benefits of the Xeon + Maxwell system essentially remain constant at higher values of $system(xeon)$, as $system(xeon) >> acc$, and the
Increasing the arrival rate to 25000 queries/s (Figure 5.8) makes the Xeon more efficient due to larger cohort sizes, and the Xeon based cluster is cheaper than the ARM based 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 based 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. We now vary the cost of the accelerator and ARM system as well, and identify dollar values for which the accelerator based platforms are a better choice than Xeon based platforms.
Figure 5.8: Ratio of the cost of ownership for different platforms as a function of the cost of the Xeon system for $T = 50 \text{ ms}$, and $\alpha = 25000$ queries/s.

5.5.1 Comparing two CPU configurations

We first compare the Xeon based cluster to the ARM based cluster (Figure 5.9). Assume $s_{system}(arm)$ as the cost of an ARM system, and $s_{system}(xeon)$ as the cost of a Xeon system (including fixed costs). Substituting values for the highest throughput configuration into Equation 4.16, for an arrival rate for 1000 queries/s and a deadline of 50 ms,

$$s_{system}(arm) < 0.114 s_{system}(xeon) + 37.015 \quad (5.1)$$

Even if a Xeon based system costs 0 dollars, if the ARM based system costs less than $37, its still a good idea to go for the ARM based cluster (Figure 5.9a). The reason for this margin is that we are paying $37 dollars extra per machine for the operational cost of the Xeon cluster, which the ARM system makes up for by being more energy efficient. The gap in throughput between the two systems is reflected
Figure 5.9: Comparison of dollars values for different systems for equal cost of ownership. The shaded region represents values for which the ARM clusters are cheaper than Xeon only clusters.

in the slope of this line (0.114), for every $100 spent on the Xeon system, the ARM system can only tolerate an increase of $11 if it is to remain more efficient.

For a higher arrival rate of 25000 queries/s (Figure 5.9b), the Xeon becomes much more efficient, and the equation becomes,

$$\$ \text{system}(\text{arm}) < 0.038 \$ \text{system}(\text{xeon}) + 6.985$$

The efficiency gap between the Xeon and the ARM decreases, and the constant factor becomes just $7. The slope also decreases to just 0.038, or, for a Xeon system costing $1000, each ARM machine should not be more than $45 to stay more efficient than the Xeon.

5.5.2 Augmenting existing servers with accelerators

We now look at augmenting a Xeon server with an accelerator (Figure 5.10). For an arrival rate of 1000 queries/s, we pick the Maxwell card. Substituting highest throughput values from Table 5.3 in Equation 4.18,

$$\$ \text{acc}(\text{maxwell}) < 2.143 \$ \text{system}(\text{xeon}) + 781.797$$

The constant dollar value of $781.797 is much higher due to the significantly higher power savings per machine from the accelerator. We can also see that the slope of
Figure 5.10: Comparison of dollars values for different systems for equal cost of ownership. The shaded region represents values for which the Xeon + Accelerator clusters are cheaper than Xeon only clusters.

This line is greater than 1, i.e. for every $100 dollar increase in the price of the Xeon, the Maxwell card can actually be more than twice as expensive, and the overall cost of ownership would still be lower. Even if the Xeon system was free, the Maxwell card retails for $300, and the accelerator augment cluster would still be cheaper.

For an arrival rate of 25000 queries/s, we use the Titan as our accelerator due to its higher throughput at larger cohort sizes, and the equation becomes,

$$S_{\text{acc}}(\text{titan}) < 2.269S_{\text{system}}(\text{xeon}) + 604.86$$ (5.4)

The power savings per machine are lower at higher arrival rates, however, the slope of the line is higher, giving greater margins for the accelerator.

5.5.3 Accelerator only clusters

Augmenting servers with accelerators for text search results in fewer machines and more energy efficient clusters. However, the host Xeon processor does not provide the optimal performance/$ for our workload, and the SoC based design minimizes the host’s overheads. We use a value of $S_{\text{system}} = $100 for the SoC. The SoC + Accelerator configurations provide significant cost savings (Figure 5.7 and 5.8), and
an accelerator based cluster is more cost effective when (Figure 5.11),

\[
\$_{\text{acc}}(maxwell) < 2.286\$_{\text{system}}(xeon) + 697.642
\]  

for an arrival rate of 1000 queries/s. For a higher arrival rate,

\[
\$_{\text{acc}}(titan) < 2.308\$_{\text{system}}(xeon) + 517.745
\]  

All of these equations show significant margins for the accelerators, even with present sticker prices for accelerators and free 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.

5.6 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 [87], requires (4B for term
weight + 4B for term identifier) \times 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 \times 20) + (256 \times 32 \times 8)}{0.05}, or 11 Mbps, which is \sim 1\% of a standard Gbps link. Using accelerators also reduces the overall number of machines required (Table 5.3), 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 5.3). 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 solution for lower arrival rates and longer deadlines, however, in that case, the query can be easily processed on the host processor.

5.7 Limitations

This work compares the cost effectiveness of similarity search across different platforms, however there are a few limitations in our analysis. We briefly summarize these below.

**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 (Equations 5.3 - 5.6).

**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, it has > 100 distinct words per page on an average, which translates 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.
5.8 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.

Reddi el al. [53] evaluate the efficiency and Quality of Service characteristics of
the Bing search engine on Xeon and Atom cores. They focus on the microarchi-
tectural differences between the two x86 platforms, and conclude that commercially
available Xeon chips are more cost effective than the Atom when scaling out within
a fixed power budget. We focus on a much wider comparison across different archi-
tectures and platforms.

Text search on GPUs was studied by Ding et al. [20]. Their work focused on a
more traditional compressed inverted list implementation, and argued for a combined
CPU+GPU implementation. Jeon et al. [54] 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 [12],
which are complementary to our work. We focus on the implementation and efficiency
analysis of the search and assume that the query and document-term matrices are
given.

Catapult [80] is a recent design that uses FPGAs to improve the ranking through-
put 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.

Recent work by Sundaram et al. [89] uses LSH to quickly search across a billion
tweets using 100 machines. However, LSH requires massive amounts of state for
storing multiple hash functions to achieve reasonable quality, they use 4000GB (40GB per machine) of state to index just 28GB of data (average 28 chars per tweet), causing scalability issues.

5.9 Summary

We evaluate the potential of accelerators to improve the performance and efficiency of similarity search. While general purpose server processors offer higher utility and ease of programmability, accelerators deliver higher throughputs, and result in significant performance benefits and energy savings. We evaluate optimized algorithms for similarity search and show 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. We also evaluate an alternative cluster design using a low power SoC driving an accelerator that is 6× cheaper and consumes less overall power than a commercial Xeon based server cluster, for an arrival rate of 25,000 queries/s such that 95% of queries finish within 50 ms.
Increasing server throughput and energy efficiency is critical in today’s Internet and cloud dependent lifestyles. Our reliance on an increasing plethora of online services to intelligently improve the quality of our lives places an enormous demand on datacenters, and methods to efficiently scale to this demand become necessary. These services manage our interaction with friends and the people around us (email, chat, social networking), our way to access and mine information (text search, image search, voice search, cloud storage and databases), our consumption habits (movie and music streaming, recommendation engines), the way we manage the environment around us (sensor networks, weather services, news services) and the way we travel (maps, car and bike rentals, traffic monitoring systems, photo services, airline and hotel bookings), just to name a few.

Traditional datacenter infrastructure has relied on Moore’s law to manage this increasing demand for scale, however, with the end of Dennard scaling and lack of an alternative to silicon scaling, CPU performance increases have largely been stagnant for the last few years. The request parallel nature of these services has provided a simple solution to scaleout, add more machines and datacenters. With
server throughput demands expected to go up more than 100× over the next several decades [90], increasing machine count is not a long term solution.

A majority of work in the last decade has been focused on improving server efficiency, or requests per Joule. Server efficiency is important, however it still does not solve the problem of increasing datacenter throughput. Power is just one of the design constraints in server management, throughput/machine is equally important as well. Reducing the number of machines needed to handle a given arrival rate is essential, as it reduces the network, brick and mortar, and mechanical infrastructure overheads.

This work argues for augmenting existing servers with data parallel accelerators as a means to address datacenter scalability. The request parallel nature of server workloads allows us to group similar requests into a cohort, and schedule this cohort on data parallel hardware, trading an acceptable increase in response time for improvements in server throughput and efficiency. We examine two common datacenter workloads, page serving and similarity search, and design algorithms and methodologies to evaluate their applicability to SIMT hardware.

6.1 Key Contributions

Traditional server workloads like HTTP page serving and text search are amenable to data parallel hardware, and this work serves as proof of that. Using intelligent algorithmic and system modifications, it is possible to harness the efficiency gains of SIMD execution for request parallel server workloads, by grouping similar requests into cohorts, and amortizing fixed instruction costs. We summarize our key contributions.
6.1.1 Server Design

We design a high throughput event based server pipeline, Rhythm, that uses cohort scheduling to schedule request cohorts on data parallel hardware. Rhythm extends cohort scheduling and event based staged servers, and maximizes throughput by stalling only on resource shortage. A cohort is defined a group of similar requests, and as we are targeting SIMT hardware, this similarity is essentially control-flow similarity. The Rhythm pipeline is applicable to any SIMD hardware, and aims to utilize the most efficient computational resource for a given cohort, which can be the host or the accelerator.

A prototype implementation of the SPECWeb Banking service using Rhythm on NVIDIA GPUs achieves more than 1.5M reqs/s without network or storage limitations, which is more than 4× the throughput of a Core i7 running at efficiencies higher than an ARM A9. We also perform a simple equal throughput scale-out analysis, that shows that more than 192 1.2GHz, 1W ARM cores are required to achieve the same throughput as Rhythm with less than 40 Watts (21% overall) in available power to support the massively scaled system. An array transpose offload can further improve Rhythm throughput to over 3M reqs/s.

6.1.2 Similarity Search

The similarity search problem involves searching for the most similar matches to a given query from a document collection based on a similarity metric. We present optimized algorithms for the High Dimensional Similarity Search problem on the host and the accelerator, which perform better than existing cuSPARSE and MKL implementations. Documents are divided into multiple tiles, search queries are grouped into cohorts, and we perform a Sparse Matrix Matrix Multiplication operation across each tile and cohort to compute the cosine similarity between each query and document. The resultant matrix is then run through our optimized Top-k implementation
to give the highest similarity matches for each query.

We also present a methodology for evaluating the cost of running the similarity search workload on a cluster of machines, with the document collection *sharded* across the cluster and query cohorts broadcast to each machine. For a given arrival rate and deadline, we compute the capital and operational cost of computing the highest similarity matches such that 95% of queries finish within the deadline. This model allows us to compare different configurations based on the host and accelerator platforms used, and the prices of these processors.

We use these algorithms to implement a prototype of a text search engine on NVIDIA GPUs and Intel Xeon processors, and compare the throughput and energy efficiency achieved by them. Our implementation on the accelerators is more than twice as efficient and delivers more than twice the throughput of the corresponding implementation on Xeon and ARM cores. Using this implementation, we show an SoC + Accelerator approach that delivers $2.3\times$ the throughput of a Xeon server while being $75\%$ more energy efficient than an ARM server for an arrival rate of 25,000 queries/s and a deadline of 50 ms.

Using our cost models, we show 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. Our SoC+Accelerator design is $6\times$ cheaper and consumes less overall power than a commercial Xeon based server cluster, for an arrival rate of 25,000 queries/s and a deadline of 50 ms. We prove that even with zero cost host processors, retail priced accelerators are still cheaper due to significant savings on operational costs from higher energy efficiency.
6.2 Directions for Future Work

This work addresses a large number of potential directions and challenges for running traditional request parallel workloads on data parallel hardware. However, research is a self-sustaining life form, and for every direction taken, there are several others that present themselves. We look at several exciting possibilities that stem from this work.

**Server Design** *Rhythm* provides a design framework for a high throughput pipeline that runs in synchronization between the host and the accelerator. We implement one variation of *Rhythm* for SPECWeb Banking using a thread per request. However, many such variations are possible, for example using multiple threads or a warp per request to trade off throughput for improved request latency. It would also be interesting to look at a *Rhythm* implementation on SoCs like the Tegra K1, where it becomes easier to synchronize memory accesses and pipeline transitions between the host and the accelerator.

Another possible variation would be one that runs entirely on the accelerator, removing host processor dependencies. Dispatch can run on the accelerator, using Dynamic Parallelism to launch pipeline stages, and GPUNet and GPUFS for network and I/O functionality. This would reduce PCIE bandwidth requirements, and would also enable more power savings by using a low power CPU core to support the CUDA runtime.

**Programming languages and Runtime** The Internet is a rapidly changing and evolving landscape, and web developers require tools that enable swift deployment and low programmer effort. *Rhythm* provides a high throughput software architecture that enables web servers to run on accelerators, however, programming for accelerators is still a challenge. Web developers still prefer to work on high level languages like PHP, and a runtime that enables cohort scheduling on request parallel
PHP pages to run them on SIMT hardware would be an important second step to Rhythm.

The design challenges lie in determining whether to support an existing language like PHP for server workloads on accelerators, or to create our own language framework for web applications. Our work hints at some of these challenges via the TRANSPOSE routine and modified string libraries, however, a significant body of work is needed before it becomes possible to write web applications on accelerators as easily as general purpose processors.

**Similarity Search** Approximate search approaches like LSH also use SpMM to compute hashes for each document using multiple hash functions, as well as Top-k to find the highest similarity matches from a hash bucket. Our evaluation of similarity search uses text search, which assumes sparse input matrices, exhaustive search and a document index based on Wikipedia. A comprehensive search system would incorporate our SpMM and Top-k algorithms, together with Dense Matrix Multiplication algorithms and would allow many variations of similarity search like image search, music search and movie search to run on accelerators. These algorithms can also be used to accelerate approximate search, allowing for gains across many classes of applications.

**Other server workloads** We look at two commonly used task parallel workloads, HTTP Page Serving and Similarity Search on accelerators. Datacenters run many applications, and a common argument against accelerators is that datacenter clusters usually run many workloads in parallel for resource consolidation to reduce costs, and accelerators can only run a specific workload. GPUs provide a middle ground in programmability between general purpose cores and FPGAs, and there has been an immense body of work in the last decade enabling many applications to run on GPUs.

An interesting future direction can be server workload consolidation on GPUs to
enable higher accelerator utilization and efficiency. Kernels from multiple applications can be simultaneously run on the device, however not all applications would be amenable to this. For example, Page Serving and Text Search can easily be scheduled together, as Page Serving is more PCIE and N/W bound, whereas Search has minimal network and PCIE interaction and is more Compute and Memory bound.

6.3 Summary

An acceptable increase in response time can be traded for an improvement in server throughput per Watt by exploiting similarity across requests using cohort scheduling to launch data parallel executions on SIMD hardware.

This work validates this hypothesis by evaluating two ubiquitous server workloads, page serving and search, and demonstrates significant increases in both throughput and throughput/Watt, while keeping 95% request latency within fluid response bounds. We perform comprehensive evaluations for these workloads, by looking at performance, energy, latency for both workloads, and dollar cost of a cluster for search, and show significant savings across all metrics thereby proving that traditional request parallel server workloads are amenable to data parallel accelerators.
Bibliography


[36] We knew the web was big... http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html.


Biography

Sandeep R Agrawal was born on 29th April 1985 in Mumbai, then called as Bombay, Maharashtra, India. He earned his Bachelors in Electronics and Communication Engineering from the Indian Institute of Technology Roorkee in 2006, a beautiful city in the Himalayas. After Roorkee, he spent four years at Samsung India working on their native Linux platform, where his work was primarily aimed towards graphics optimizations for the various chipsets the platform ran on. He got his Masters degree in Electrical and Computer Engineering in 2012, followed by his PhD degree in Computer Science from Duke University in 2015.

He won several awards in Samsung India, notably the Spot Award for 2D graphics optimizations in 2009. In his PhD, he received the Outstanding Research Initiation Project (RIP) Award by the Computer Science Dept. in 2013. His work related to high throughput server design on accelerators was published in ASPLOS 2014 [2]. He was also a recipient of the NVIDIA Graduate Fellowship for 2014. After his PhD, he will be joining the RAPID group at Oracle Labs in May 2015.