Rhythm: Harnessing Data Parallel Hardware for Server Workloads

Sandeep R Agrawal
Duke University
sandeep@cs.duke.edu

Valentin Pistol
Duke University
pistol@cs.duke.edu

Jun Pang
Duke University
pangjun@cs.duke.edu

John Tran
NVIDIA
johntran@nvidia.com

David Tarjan *
NVIDIA

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

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 the data center as a whole, using off-the-shelf hardware for individual servers. Server capacity is typically increased by adding more machines, which is cheap, though inefficient in the long run in terms of energy and area.

Our work builds on the observation that server workload execution patterns are not completely unique across multiple requests. 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 evaluating both software and hardware for future cohort-based servers. Our evaluation of Rhythm on future server platforms shows that it achieves 4× the throughput (reqs/sec) of a core i7 at efficiencies (reqs/Joule) comparable to a dual core ARM Cortex A9. A Rhythm implementation that generates transposed responses achieves 8× the i7 throughput while processing 2.5× more requests/Joule compared to the A9.

* Now with Samsung Electronics, d.tarjan@samsung.com

ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541956

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

Keywords high throughput; power efficiency; execution similarity

1. Introduction

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. 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, 11, 16, 17, 39, 41].

For example, 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 [25]. Reducing CPU load inspired the design of HipHop, which translates PHP to native code thereby increasing throughput per server by $> 5\times$ [15].

Current system designs based on commodity multicore processors may not be the most power/energy efficient for all server workloads. There is ongoing debate over which architecture is best suited for specific workloads [17, 31, 32, 39, 43]. Figure 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.

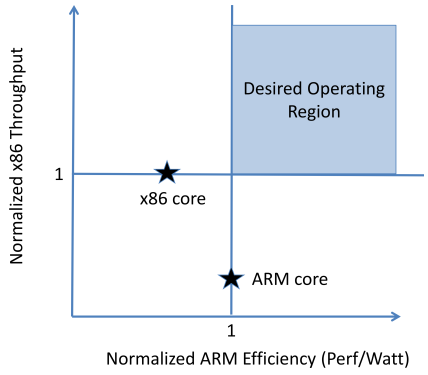


Figure 1. Server Design Space

Recent highly-threaded accelerators (i.e., NVIDIA Kepler [44], Intel Xeon Phi [30]) 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.

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 [45], AMD Fusion [8], or many core systems [35]) 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 paper 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 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.* Our goal is to trade an increase in response time for improvement in server throughput per Watt by exploiting similarity across requests using cohort scheduling [34] to launch data parallel executions.

The contributions of this work include:

- *Rhythm*, a software architecture for high throughput SIMT-based servers. (*Rhythm* derives from the Greek word *rhythmos*, meaning any regular recurring motion or symmetry)
- A prototype implementation of *Rhythm* and the SPECWeb Banking service on NVIDIA GPUs that demonstrates it is possible to run server workloads on a GPU.[1]
- 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.
- Standalone C and C+CUDA implementations of the SPECWeb2009 Banking workload that we plan to release publicly.

The remainder of this paper is organized as follows. Section 2 examines technology trends and workload characteristics that motivate our work. We present our overall server architecture (*Rhythm*) in Section 3 and provide implementation details in Section 4. Sections 5 and 6 describe our methodology and evaluates our prototype's potential performance on future server platforms. Related work is discussed in Section 7, while Section 8 concludes and discusses future work.

2. Motivation

This section motivates our work by first reviewing GPU-style accelerator efficiencies and then discussing several networking and system architecture trends that remove constraints of existing systems. We conclude this section with a study that demonstrates the existence of similarity across requests.

2.1 Accelerator Efficiency

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%-

[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.

50% of overall core power [49]. 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 paper we focus on GPU-style accelerators, our work is broadly applicable to data parallel architectures.

The potential efficiencies available with emerging GPU-style accelerators is compelling. However, several aspects of current systems may limit the ability to exploit these efficiences. These limitations include PCIe bandwidth and latency, network bandwidth and the potential lack of similarity across requests. Below we address these potential limitations.

2.2 Enabling Trends

Current systems may introduce bottlenecks that limit the overall server throughput possible with a highly threaded accelerator. Here we identify three trends that can remove or mitigate these bottlenecks enabling scale-up solutions for high throughput servers.

2.2.1 Increasing 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 [1] and announced systems claim to sustain these bandwidths [29]. Furthermore, research is exploring techniques to achieve 10^{12} bps to 10^{15} bps using novel multicore fiber optics [47].

Once network bandwidth increases, the PCIe bus may become the bottleneck. Fortunately, PCIe 4.0 doubles bandwidth to 16G Transfers/sec [3] 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.

2.2.2 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 [45], AMD Fusion [8]). 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 [35, 36] shows overall benefits to data center design by using SoC architectures to reduce TCO, and mention the possibility of integrated accelerators for servers.

2.2.3 Operating System and Backend Services

A final enabling trend is 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 [50] provides a filesystem abstraction that enables access from the GPU. Similar in spirit is the recent development of vector interfaces [55] 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, 28, 57] or constructing specialized hardware for key-value (e.g., memcached [19]) servers [10, 36].

2.3 Request Similarity

The potential efficiencies of parallel accelerators is compelling, and the above trends expose the opportunity to exploit those efficiences in future high throughput server designs. However, it remains to be determined if server workloads can exploit those efficiencies. As a first step toward answering this question we performed preliminary analysis of the SPECWeb2009 Banking workload [51]. This workload is a PHP web site representative of typical web-based banking.

We use Pin [40] to obtain x86 dynamic basic block traces for 61 individual dynamic web content (PHP) requests from the SPECWeb2009 Banking workload. These requests access 14 distinct top-level PHP files served by the Apache web server.[2] We use the UNIX utility *diff* to merge traces and obtain a measure of similarity between request traces based on the length of the merged trace. Minimal differences between traces indicates the two executions followed the same control paths during their execution.

The opportunity for exploiting GPU-style hardware for server workloads is obtained by examining the length of merged traces (e.g., shorter merged traces implies faster execution time). The resulting merge is an approximation of the actual execution order for all the batched requests executing on data parallel hardware. Note that our analysis is performed offline, and thus assumes a very high request arrival rate.

We merge traces for independent requests that access the same initial PHP file, resulting in 14 distinct merged traces. Execution speedup is approximated as the sum of traces (in number of basic blocks) divided by the merged trace size (i.e., cohort execution on idealized SIMD hardware). In our experiments between 2 and 6 traces per request (PHP file) are merged, with most requests having 5 unique traces. Figure 2 shows per request speedup normalized to ideal (linear) speedup. From these results we observe nearly linear speedup (i.e., nearly identical executions) for each request type.

[2] There are a total of 16 requests in this workload, but our experiments only exercised 14 of them.

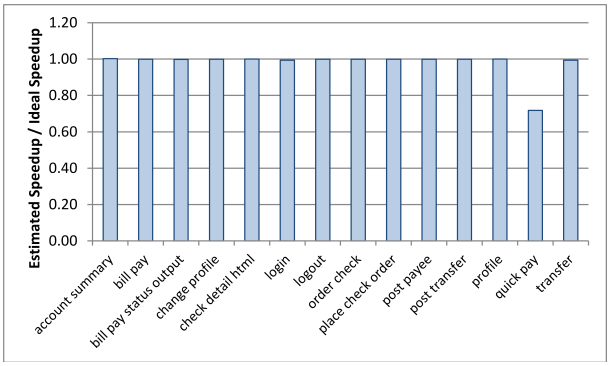


Figure 2. Potential Speedup of SPECWeb2009 Banking Workload on Data Parallel Hardware Relative to Ideal Speedup

We performed a similar study using Facebook's HipHop PHP to C++ system [15]. This experiment removes any potential affects of online PHP interpreting in Apache/Zend. We observed qualitatively similar results (not shown) with significant similarity among requests of the same type. Furthermore, on average there are approximately 33,000 instructions between system calls with 97% of them related to filesystem or network I/O and could utilize the high throughput interfaces described above (i.e., GPUfs and vector interfaces).

The above results quantitatively demonstrate the intuitive result that similar instruction control flow exists in a server workload for requests of the same type. 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.

3. *Rhythm* Server Software Architecture

Rhythm is a software architecture that extends event-based staged servers and cohort scheduling [34, 52, 56] 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 4 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.

3.1 Overview

Figure 3 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

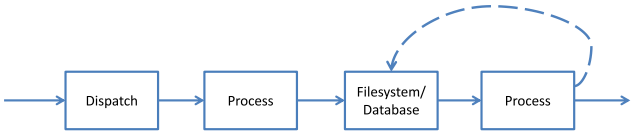


Figure 3. Typical Web Server Request Processing Overview

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 [18] or a staged event-based server [34, 52, 56].

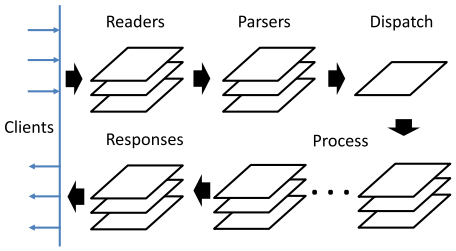


Figure 4. *Rhythm* Pipeline (Each stage can have multiple instances)

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 4: 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 process stage is composed of n backend stages and $n + 1$ process stages. A specific implementation of the *Rhythm* pipeline can map each of these stages to either a general purpose CPU or an SIMT accelerator. We defer discussion of a specific implementation to Section 4.

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 [52]). 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 5) that provides cohort scheduling [34]. The server thread is generally responsible for delaying requests an appropriate amount of time to form

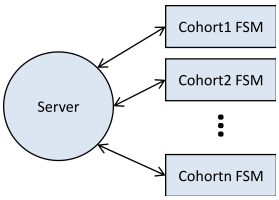


Figure 5. *Rhythm* Event-based Server: Each Finite State Machine (FSM) is associated with a cohort of requests.

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 [41]. *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 *Free*d after the responses are sent to their respective clients.

3.2 The *Rhythm* Pipeline

We now elaborate more on the design and function of the individual stages of the *Rhythm* pipeline. 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 [20], 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 parameters 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 [50]) would enable dispatch execution on the device and decrease transfer overheads, but we leave exploring this option to future work. 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; we leave exploring alternative implementations as future work.

4. 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.

4.1 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. 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, a callback 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 the next stage in the pipeline. Although this approach doesn't appear to limit throughput, it introduces unnecessary power consumption for polling many inflight 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.

4.2 Pipeline Stages

Reader/Parser The reader is double buffered to overlap request processing and 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/Backend 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. Exploring alternatives that do not preallocate resources is part of our future work.

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.

4.3 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.

4.3.1 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.

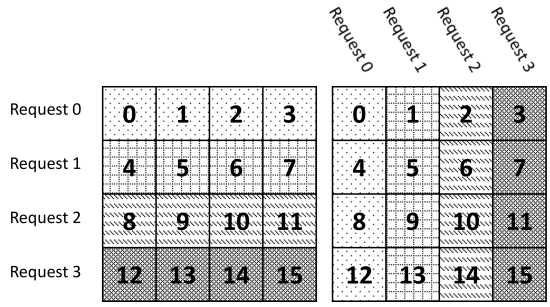


Figure 6. Request Buffer Layout (for illustrative purposes)

4.3.2 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.

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 6. 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 [48]. 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.

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.

4.5 Request Flow in *Rhythm*

Figure 7 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).

4.6 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

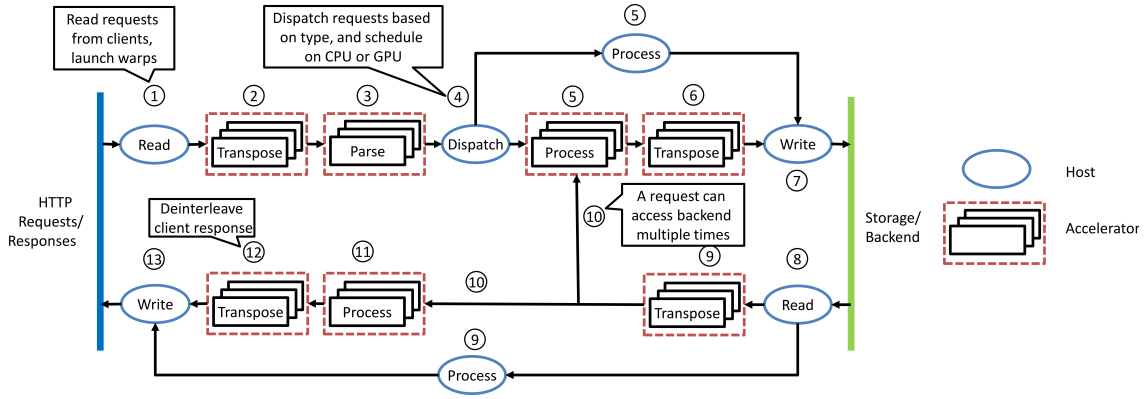


Figure 7. Rhythm Web Server Request Processing Overview

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. We also perform several optimizations to *Rhythm* based on CUDA features, including:

- 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. We evaluate the benefits and pitfalls of *Rhythm* on existing platforms, and explore its potential to serve as a guideline for future server architectures.

5. Methodology

5.1 Platforms

Table 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 [51] 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 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

Platform	GHz	Description
Core i5	3.4	Core i5 3570, 22 nm, 4 cores (4 threads), 8GB DDR3 RAM, 1Gbps NIC
Core i7	3.4	Core i7 3770, 22 nm, 4 cores (8 threads), 16GB DDR3 RAM, 1Gbps NIC
ARM A9	1.2	OMAP 4460, 45 nm, Panda board, 2 cores, 1GB LPDDR2 RAM
Titan	0.8	GTX Titan, 28 nm, 14 Streaming Multiprocessors, 6GB GDDR5 Memory

Table 1. Experimental System Platforms

kernel launches based on backend data, making it difficult to implement, and *check detail images* is completely disk bound, requiring GPUfs integration to allow us to process it on the GPU. We plan to address both these requests in future work. Table 2 summarizes characteristics about the Banking workload. The second column is the dynamic instruction count for our standalone C implementation and is the average 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 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

Request Type	x86 Instructions per Request	Response Size (KB)		Fraction of Requests (%)	Backend Requests
		SPECWeb	Rhythm		
login	132,401	4	8	28.17	2
account summary	392,243	17	32	19.77	1
add payee	335,605	18	32	1.47	0
bill pay	334,105	15	32	18.18	1
bill pay status output	485,176	24	32	2.92	1
change profile	560,505	29	32	1.60	1
check detail html	240,615	11	16	11.06	1
order check	433,352	21	32	1.60	1
place check order	466,283	25	32	1.15	1
post payee	638,598	34	64	1.05	1
post transfer	334,267	16	32	1.60	1
profile	590,816	32	64	1.15	1
transfer	277,235	13	16	2.24	1
logout	792,684	46	64	8.06	0
Average	429,563	15.5	26.4	100	1.2

Table 2. SPECWeb Banking Workload

as well using GPUfs [50], however, we leave that to future work. Static images can also be served at high throughputs via Content Delivery Networks (CDNs) like Akamai [46]. We do not evaluate image throughput for our prototype.

5.2 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. 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[3] 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.

5.3 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.

5.3.1 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 48M requests. Using the request distributions from Table 2, we compute a weighted harmonic mean of request efficiency(throughput/watt) to obtain the efficiency for the entire workload.

[3] We use dynamic to represent the non-idle power under load. This is distinct from dynamic/static power used for circuit analysis.

5.3.2 Emulation

Our test system uses a 1Gbps NIC, and for an average response size of 16KB, cannot support more than $\sim$ 8K requests/sec. 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. 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 per-

formed 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 [2]). 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. Eliminating the network only eliminates the overheads of the *read()* and *write()* system calls and the network stack, and optimizing them is an important, but orthogonal issue to our work. A local or device backend emulates a high throughput key-value store [55], or the use of a database cache [19] on the local machine, which is commonly used to tolerate backend latencies.

6. Evaluation

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, which is beyond the scope of this paper, would account for many differences such as technology node, different accelerators, etc. and include additional workloads. Nonetheless, our results point towards the design of future data parallel accelerators specialized for server workloads.

6.1 Overall Results

Table 3 shows the throughput, latency, wall and dynamic power of the banking workload for our various modeled platforms. More worker threads is always more 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 8 shows the throughput-efficiency for the various platforms. Considering both wall power(Figure 8a) and dynamic power (Figure 8b), 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 $\sim 1.5\times$ more requests per Joule compared to the ARM and delivering $7\times$ more requests per second compared to the Core i7. Titan B provides more than $4\times$ 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 [13], 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*.

6.1.1 Titan A: Emulated Remote Backend

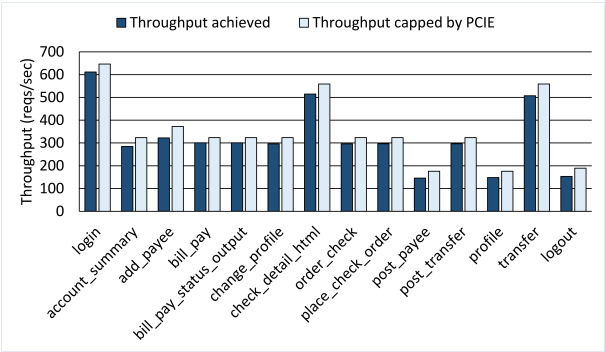
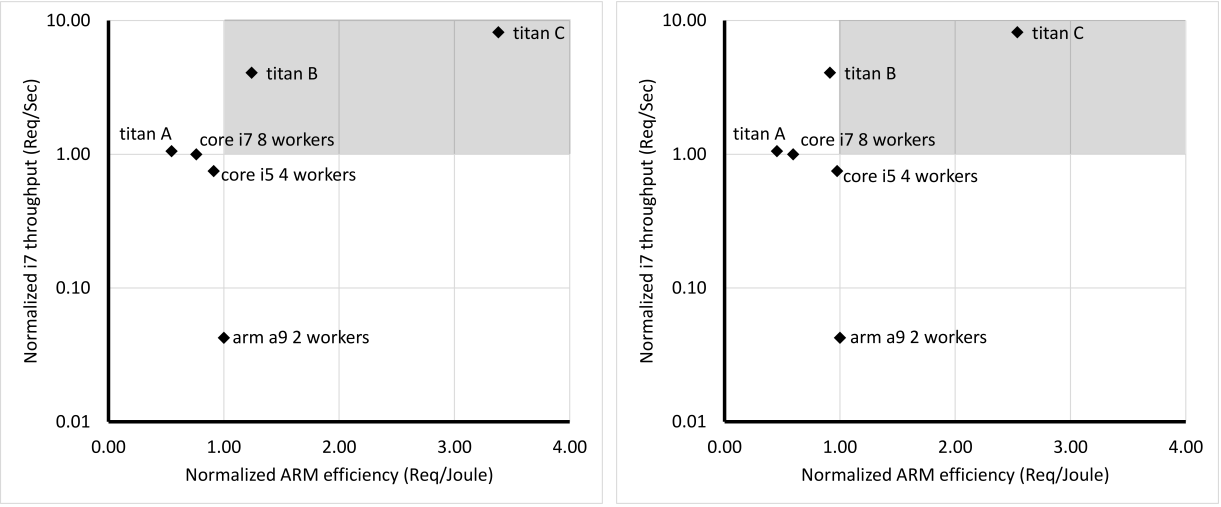


Figure 9. PCIe 3.0 Limitations in Titan A for various request types

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 9 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



(a) Wall Power

(b) Dynamic Power

Figure 8. Throughput-Efficiency for Wall Power (a) and Dynamic Power (b). 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.

Platform	Power (Watts)			Latency (ms)	Throughput (KReqs/s)	Reqs/Joule (Efficiency)	
	Idle	Wall	Dynamic			Wall	Dynamic
Core i5 1 worker	47	67	20	0.016	75	972	3283
Core i5 4 workers	47	98	51	0.016	282	2447	4712
Core i7 4 workers	45	147	102	0.014	331	1901	2735
Core i7 8 workers	45	156	111	0.014	377	2042	2873
ARM a9 1 worker	2	3.4	1.4	0.176	8	1672	4061
ARM a9 2 workers	2	4.5	2.5	0.176	16	2683	4830
Titan A	74	226	152	86	398	1469	2193
Titan B	74	306	232	24	1535	3329	4410
Titan C	74	285	211	10*	3082	9070	12264

Table 3. SPECWeb Banking Experimental Results. *Titan C Latency is for transposed response

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.

6.1.2 Titan B: Integrated NIC and Device Backend

Titan B increases *Rhythm*'s average throughput to more than $4\times$ 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 2 with dynamic efficiency in Figure 10 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 $3.5\times$-$5\times$ higher than the core i7, with dynamic efficiencies 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.

6.1.3 Titan C: Increasing Device Utilization

Titan C tries to push the *Rhythm* pipeline to its limits by increasing GPU utilization and offloading the response trans-

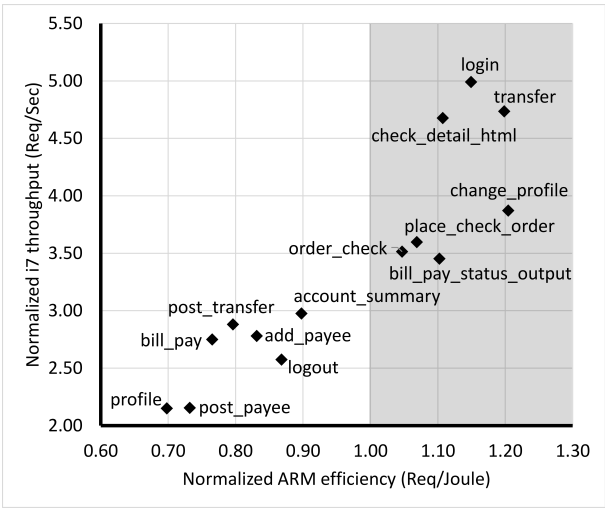


Figure 10. Throughput-Efficiency for different request types on Titan B (dynamic power). *Rhythm* buffer sizes that are close to required sizes perform well (shaded area represents the desired operating range).

pose. With these optimizations, Titan C achieves more than 3M reqs/sec on an average, or more than $8\times$ the throughput of the Core i7. Increasing GPU utilization amortizes the fixed power overheads, achieving a dynamic efficiency of more than $2.5\times$ that of the ARM, and a wall efficiency of more than $3.3\times$. 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.

6.2 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 3). Based on our measurements, and other studies [35], 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. However, a more detailed analysis of complete server design, including specialized data parallel accelerators, is required, and we leave that to future work.

6.3 System Resource Requirements

Network Bandwidth We perform our experiments on *Rhythm* for a request size of 512B, a backend request size of 1KB and a 4KB backend response size. Using the average value of the response size of the SPECWeb Banking workload (Table 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 N/W 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 [37]. 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 [1].

Memory Capacity We are currently limited by the memory on the device. For our experiments, we emulate 16M active sessions on the GPU, 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.

6.4 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. We leave this exploration to future work.

Cohort Size sensitivity We performed experiments on *Rhythm* for cohort sizes ranging from 256 to 8192, and found 4096 to provide the right balance between high throughput and memory limitations. 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 for 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 tranpose, 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.

7. Related Work

This paper 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 [9, 12, 27] and mapping non-traditional workloads onto GPUs, such as MIMD programs [14], database queries (e.g., [6, 24, 57], etc.) and memcached [28], and software routers [26]. Other work explores the potential opportunities created by delaying server requests for either improved memory hierarchy performance [34] or energy management [33, 41], and allocating compute resources dynamically based on load in staged servers [56].

Recent microarchitecture work [54] shows the benefits of data-triggered threads and methods for eliminating redundant computation, while other work (MMT [38] and Thread Fusion [23]) 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 [7] 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 [21, 22, 42, 53]. More generally, several researchers are exploring processor design for data center workloads [10, 17, 31, 32, 35, 36, 39, 43].

8. 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 $4\times$ to $8\times$ 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. We plan to explore ways to increase the efficiency of *Rhythm* by designing data parallel processors specialized for server workloads. Currently, we test *Rhythm* on GPUs, and other accelerators like the Xeon Phi, NEON and Tegra also provide interesting possibilities. Programming for accelerators involves a significant amount of effort, and we are working on creating a language for regaining programmer productivity. We are also exploring other workloads like Search, Email and Chat, and deploying them using *Rhythm*. This paper is just a first step in exploring an exciting and varied design space.

Acknowledgments

We would like to thank the Duke Computer Architecture Group, Mark Silberstein and Emmett Witchel for their support, and the reviewers for their valuable time and feedback. This work was supported in part by the National Science Foundation (CCF-1335443) and NVIDIA.

References

[1] http://www.ieee802.org/3/.

[2] Hybrid memory cube. `http://hybridmemorycube.org`.

[3] Pci express 4.0 faq. http://www.pcisig.com/news_room/faqs/FAQ_PCI_Express_4.0/.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. . URL http://doi.acm.org/10.1145/1629575.1629577.

[5] I. Atta, P. Tözün, X. Tong, A. Ailamaki, and A. Moshovos. Strex: Boosting instruction cache reuse in oltp workloads through stratified transaction execution. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 273–284, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. . URL http://doi.acm.org/10.1145/2485922.2485946.

[6] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 94–103, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-935-0. . URL http://doi.acm.org/10.1145/1735688.1735706.

[7] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-Execution: Multicore Caching for Data-Similar Executions. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 164–173, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. .

[8] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Micro, IEEE*, 32(2):28 –37, march-april 2012. ISSN 0272-1732. .

[9] I. Buck. GPU computing with NVIDIA CUDA. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 6, New York, NY, USA, 2007. ACM. .

[10] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An fpga memcached appliance. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '13, pages 245–254, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1887-7. . URL http://doi.acm.org/10.1145/2435264.2435306.

[11] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 103–116, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. . URL http://doi.acm.org/10.1145/502034.502045.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC, pages 44–54. IEEE, 2009. .

[13] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/2408776.2408794.

[14] H. G. Dietz and B. D. Young. Mimd interpretation on a gpu. In *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 65–79, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13373-8, 978-3-642-13373-2. . URL http://dx.doi.org/10.1007/978-3-642-13374-9_5.

[15] Facebook. Hiphop-php. https://github.com/facebook/hiphop-php/wiki.

[16] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. . URL http://doi.acm.org/10.1145/1250662.1250665.

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

[18] R. Fielding and G. Kaiser. The Apache HTTP Server Project. *IEEE Internet Computing*, 1(4):88–90, 1997. ISSN 1089-7801. .

[19] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1012889.1012894.

[20] T. I. E. T. Force. Hypertext transfer protocol – http/1.1. http://www.ietf.org/rfc/rfc2616.txt.

[21] W. Fung and T. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *17th IEEE International Symposium on High-Performance Computer Architecture*, HPCA-17, 2011.

[22] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. .

[23] J. González, Q. Cai, P. Chaparro, G. Magklis, R. Rakvic, and A. González. Thread Fusion. In *Proceeding of the 13th international symposium on Low power electronics and design*, ISLPED '08, pages 363–368, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-109-5. .

[24] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. . URL http://doi.acm.org/10.1145/1007568.1007594.

[25] M. Hachman. How facebook will power graph search. http://slashdot.org/topic/datacenter/how-facebook-will-power-graph-search/.

[26] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM*

SIGCOMM 2010 conference, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2. . URL http://doi.acm.org/10.1145/1851182.1851207.

[27] J. Hensley. AMD CTM Overview. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM. .

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

[29] http://netronome.com.

[30] Intel Corp. Intel xeon phi coprocessor. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html.

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

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

[33] W. Lang and J. Patel. Towards Eco-friendly Database Management Systems. In *4th Biennial Conference on Innovative Data Systems Research*, Asilomar, California, USA, January 4-7 2009.

[34] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance (Extended Abstract). In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 182–187, New York, NY, USA, 2001. ACM. ISBN 1-58113-425-8. .

[35] S. Li, K. Lim, P. Faraboschi, J. Chang, P. Ranganathan, and N. P. Jouppi. System-level integrated server architectures for scale-out datacenters. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 260–271, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1053-6. . URL http://doi.acm.org/10.1145/2155620.2155651.

[36] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. . URL http://doi.acm.org/10.1145/2485922.2485926.

[37] Z. Liu, Y. Saifullah, M. Greis, and S. Sreemanthula. Http compression techniques. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 4, pages 2495–2500 Vol. 4, 2005. .

[38] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong. Minimal Multi-threading: Finding and Removing Redundant Instructions in Multi-threaded Processors. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 337–348, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. .

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

[40] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. .

[41] D. Meisner and T. F. Wenisch. Dreamweaver: architectural support for deep sleep. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 313–324, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. . URL http://doi.acm.org/10.1145/2150976.2151009.

[42] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. .

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

[44] NVIDIA. Tesla k20 gpu accelerator board specification. http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf, .

[45] NVIDIA. Nvidia tegra mobile processors. http://www.nvidia.com/object/tegra-3-processor.html, .

[46] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, Aug. 2010. ISSN 0163-5980. . URL http://doi.acm.org/10.1145/1842733.1842736.

[47] D. J. Richardson, J. M. Fini, and L. E. Nelson. Space-division multiplexing in optical fibres. *Nat Photon*, 7(5):354–362, 05 2013. URL http://dx.doi.org/10.1038/nphoton.2013.94.

[48] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda. http://docs.nvidia.com/cuda/samples/6_Advanced/transpose/doc/MatrixTranspose.pdf.

[49] J. Sartori, B. Ahrens, and R. Kumar. Power balanced pipelines. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, 2012. .

[50] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: integrating file systems with gpus. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13. ACM, 2013.

[51] Standard Performance Evaluation Corporation. SPECweb2009. http://www.spec.org/web2009/.

[52] I. Sysoev. Nginx introduction and architecture overview. http://nginx.org/en/docs/introduction.html.

[53] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 22:1–22:11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. .

[54] H. Tseng and D. Tullsen. Data-Triggered Threads: Eliminating Redundant Computation. In *Proceedings of 17th International Symposium on High Performance Computer Architecture*, HPCA-17, 2011.

[55] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 8:1–8:13, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. . URL http://doi.acm.org/10.1145/2391229.2391237.

[56] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. .

[57] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 107–118, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4924-8. . URL http://dx.doi.org/10.1109/MICRO.2012.19.