Achieving scalability in a k-NN multi-GPU network service with *Centaur*

Amir Watad*, Alexander Libov†, Ohad Shacham‡, Edward Bortnikov‡, Mark Silberstein*
* Technion † Amazon ‡ Yahoo! Labs

***Abstract*—Centaur is a GPU-centric architecture for building a low-latency approximate k-Nearest-Neighbors network server. We implement a multi-GPU distributed data flow runtime which enables efficient and scalable network request processing on GPUs. The runtime eliminates GPU management overheads from the CPU, making the server throughput and response time largely agnostic to the CPU load, speed or the number of dedicated CPU cores.**

Our experiments systems show that our server achieves near-perfect scaling for 16 GPUs, beating the throughput of a highly-optimized CPU-driven server by 35% while maintaining about 2msec average request latency. Furthermore, it requires only a single CPU core to run, achieving over an order of magnitude higher throughput than the standard CPU-driven server architecture in this setting.

***Keywords*-GPU; Parallel Computing;**

I. Introduction

High-concurrency memory-demanding server applications are ubiquitous in high performance computing systems and data centers [12]. They pose three distinctive requirements to developers: low, strictly bounded response time for client requests, high throughput for higher server efficiency, and large physical memory to keep the data set resident to achieve these performance goals. Fulfilling all these requirements together is a significant challenge.

GPUs are a compelling platform to boost system compute capacity and to meet the challenging requirements of network servers at a fraction of operational and energy costs. Recent works have shown that GPUs may greatly improve performance and power efficiency for server applications [9], [11]. The abundance of inter-request and intra-request parallelism enables efficient use of GPUs as a Multiple-Instruction-Multiple-Data multi-tasking platform [53]. As GPUs are increasingly deployed in data centers and public clouds [42], [22], [10], broadening the range of applications that can benefit from using them is particularly appealing.

The question we address in this paper is how to design a scalable GPU-accelerated network server which can accommodate large data sets while maintaining low latency and high throughput. More specifically, we focus on the design of a low-latency *approximate k-Nearest Neighbors (k-NN)* server [27]. k-NN is a class of machine learning algorithms forming the core of many latency-critical information retrieval systems operating on large multi-dimensional datasets. These include content-based image retrieval and similarity search, e.g., Google images, and recommender systems for ads such as Yahoo! Gemini. In these systems, the processing time of the k-NN remains the main bottleneck, in particular for larger data sets in higher dimension.

Prior works showed that accelerating k-NN computations on GPUs may bring substantial speedups [19], [31], [20], [32]. These works mostly considered efficient parallelization and implementation of a stand-alone version of k-NN algorithms on GPUs. This paper, however, is different in that it deals with the challenges of *integrating GPU-accelerated algorithms into a realistic low-latency network service performing interactive k-NN searches*.

The common structure of the approximate k-NN search algorithms is as follows. The dataset is pre-processed into *clusters* offline. At runtime, each query is processed in three stages: (1) `filter`: find the subset of w clusters which are likely to hold the matching items; (2) `search`: search the candidate clusters exhaustively; (3) `reduce`: return top k matches. This structure is representative of many k-NN algorithms [27], [18], [17].

The combination of the following three aspects of this algorithm makes building such a service on GPUs particularly challenging. First, the algorithm requires fast access to the *entire* data set because the clusters in the `search` stage are chosen depending on the outcome of the `filter` stage. Second, the dataset must be resident in GPU memory to benefit from GPU acceleration. Therefore, to scale to large datasets one must use multiple GPUs by splitting the clusters into *shards* and distributing them across the GPUs. With today's GPUs it is possible to scale up to 512GB per server using 16 GPUs. Last, both `filter` and `search` are computationally demanding, therefore it is essential to run them on the GPU to achieve low latency for each request.

The main design challenge is to efficiently execute the `filter-search-reduce` pattern over multiple GPUs for a stream of queries. Each `filter` stage starts in one GPU and produces the candidate clusters. It is then followed by the `search` stage invoked on one or more other GPUs where the candidate clusters are stored. Running such computations requires close coordination of data transfers, kernel invocations and multi-GPU synchronization. Moreover, handling multiple independent queries arriving at the server requires careful pipelining to achieve full system utilization.

More fundamentally, as the number of GPUs increases to scale to larger data sets, the multi-GPU scaling becomes bounded by *the CPU throughput* due to GPU management overheads. Figure 1 shows the effect of changing the number of CPU cores dedicated to GPU management on the

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00027

IEEE computer society

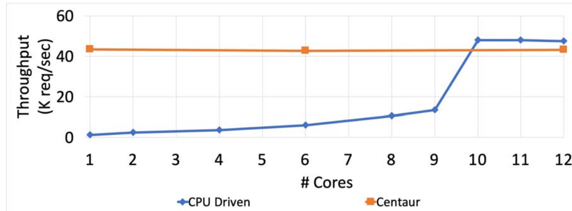


Figure 1: The throughput of 6-GPU k-NN server for CPU-driven and Centaur-based GPU-centric design, as a function of the number of dedicated CPU cores. Centaur performance is agnostic to CPU availability. Higher is better.

throughput of a k-NN server that uses 6 GPUs (3xNVIDIA K80 GPUS). Without enough CPUs (10 in this case) to drive the computations the throughput is low, leaving GPUs underutilized. As we show analytically in Section III, such poor multi-GPU scaling of low-latency k-NN computations is *inherent* to the traditional CPU-centric server design.

In this paper we propose a novel *GPU-centric* k-NN server architecture we call **Centaur**, which scales well with the number of GPUs and provides millisecond-range query latency *without relying on the main host CPU*. The performance of the server is largely *agnostic to the number of CPU cores* (Figure 1). Moreover, we prove theoretically (Section IV-C) that the GPU-centric system design enables linear weak scaling for any number of GPUs.

A key principle of Centaur is that the *system refrains from using the CPU for inter-GPU data transfers, multi-GPU syncrhonization or GPU kernel invocation*. Instead, all the GPUs run *persistent* kernels [23] which continuously execute an event loop, waiting for pre-defined external evens to trigger actual computations. These kernels are inter-connected to form a data flow graph using a new *gpipe* programming abstraction. As common in a data flow model, the computations are self-scheduled, such that each kernel invokes its computations after receiving all the expected inputs. To allow the system to interact with clients over the network, some kernels implement an in-GPU network server using GPUnet [46] GPU-side networking layer which provides the standard socket interface.

Gpipe is a communication and synchronization primitive akin to UNIX pipes that enables GPU threadblocks to interact directly with each other, regardless whether they are running in the same or different GPUs. It implements a lockless producer-consumer queue which requires no atomic operations, therefore it correctly works for connecting GPU kernels across PCIe bus.

The host CPU sets up the system by carefully allocating GPU cores to kernels to maximize GPU utilization. Centaur runs without CPU involvement after the setup.

Such a design enables exploiting multiple levels of parallelism in server workloads (1) intra-request, by running the stage logic in multiple GPU threads, (2) inter-request, by allowing concurrent processing of multiple requests in different `filter-search-reduce` pipelines and different stages of the same pipeline, and (3) communication - computation overlap via gpipes and GPU-side network API.

We implement a k-NN server using one of the available approximate k-NN algorithms as the building block [28]. We choose this algorithm for the convenience of implementation, however we believe that our results will hold for other algorithms that share the same `filter-search-reduce` computing structure, such as [18], [17], [21].

We prototype the Centaur k-NN server for NVIDIA GPUs, and run an image similarity search service on the top. We use a 1M subset of the ANN_SIFT1B [28] image dataset, split approximately evenly into 8K shards across all the server GPUs. Centaur achieves perfect throughput scaling with up to 9 GPUs and 91% efficiency with 16 GPUs (eight NVIDIA K80 boards) running in Amazon EC2. In contrast, the highly-optimized traditional host-centric design fails to scale linearly beyond 4 GPUs, and levels off entirely for 9 GPUs. Thus, with 16 GPUs, Centaur outperforms the baseline by 35%, serving about 85K queries/sec at *2 msec* end-to-end average latency measured from a remote client.

We also demonstrate that Centaur's performance is insensitive to CPU load, speed, and number of available CPU cores. It maintains the same throughput even with a *single CPU core* and runs up to $40\times$ faster than the host-centric baseline running on one core. This property of the GPU-centric architecture allows significant cost reduction in multi-GPU servers enabling allocation of a single CPU core instead of using one-CPU-core-per-GPU as commonly suggested in state-of-the-art systems [1].

Prior works on GPU OS abstractions [41], [30], [46], [14], [44], [43] and GPU servers [25], [11], [9] have already demonstrated the benefits of turning GPUs into first-class system processors, relaxing the GPU tight dependence on the CPU. Centaur adopts the same concepts applying them to multi-GPU servers.

We believe that with the hardware trends toward fast inter-GPU communications via NVLINK [16], growing availability of large-scale multi-GPU nodes [1] and emerging multi-GPU architectures which rely on wimpy low-power processors alone [4], the benefits of GPU-centric design will become even more pronounced in the future.

This paper makes the following contributions:

- A novel GPU-centric multi-GPU server design for scalable k-NN server based on distributed data flow model,
- A theoretical scalability analysis of the CPU-centric (traditional) design highlighting its inherent limitations,
- A Gpipe abstraction for inter-kernel communication and synchronization,
- A comprehensive performance and scalability evaluation of the low-latency GPU-centric k-NN server on 16 GPUs.

[1]See, e.g., http://timdettmers.com/2018/12/16/deep-learning-hardware-guide.

II. Background

We use NVIDIA CUDA terminology because we implement Centaur on NVIDIA GPUs, but most other GPUs that support the cross-platform OpenCL standard [49] share the same concepts.

Persistent GPU kernels. Traditionally, GPUs are used as co-processor, where the main program that runs on the CPU and invokes GPU kernels to execute individual functions. Another approach is to use *persistent* kernels [23], which run in an event loop on the GPU. These kernels process *task execution requests* without the need to invoke GPU kernels. The requests are fed and retrieved via a communication channel implemented in GPU global memory.

GPUDirect. GPUs and other peripheral devices may communicate directly via PCIe bus. In particular, NVIDIA GPUDirect [37] technology enables NICs to read/write from/to GPU memory without the CPU involvement. GPUDirect reduces latency and CPU overhead by eliminating redundant copies otherwise necessary to transfer data between the GPU and the NIC.

GPUnet. GPUnet [30] is a GPU-side networking layer that provides high-level networking APIs for GPU kernels. It enables the use of socket abstractions within GPU kernels, simplifying the development of GPU network server applications. GPUnet allows direct communication between the GPU and the NIC.

III. Scalability of CPU-centric server architecture

Our goal is to analyze the system scaling to more GPUs, while taking into account the GPU management overheads and the computing structure of the k-NN algorithm.

We first analyze the throughput assuming that network requests are fed to the GPUs one by one. Here we conclude that multi-GPU scaling in k-NN servers is bounded by the CPU capacity to manage those GPUs.

We then compute the response latency and the server throughput assuming the common practice of batching multiple requests to amortize the overheads. We show that larger batch size significantly affects respon. latency, whereas small batches result in load imbalance with more GPUs.

Conventional multi-GPU k-NN server design. We assume that `filter`, `search` and `reduce` are implemented each as an individual GPU kernel. We further assume that the dataset is split into shards and distributed among the GPUs. The `search` kernels are invoked on the GPUs which store their respective clusters.

For every request the execution flow is as follows:

1) Invoke the `filter` kernel,
2) Invoke W `search` kernels, one for each GPU which stores the respective cluster found by `filter`. W is the parameter of the algorithm [28],
3) Invoke the `reduce` kernel,

For maximum throughput, the server follows the state-of-the-art Staged Event-Driven Architecture (SEDA) [52] whereby few CPU threads are multiplexed among multiple concurrent incoming requests. Therefore, all the GPU invocations are asynchronous, and are monitored by periodically *querying for completion* while concurrently handling other incoming requests.

We assume that the bulk of the dataset, i.e., the clusters in the `search` stage, are distributed randomly among the GPUs. This is a valid assumption: there is usually no way to predict which set of clusters gets accessed in a specific query, precluding locality optimizations – the very reason why approximate k-NN algorithms are necessary.

For simplicity we ignore the cost of data transfers between GPUs, thus producing an optimistic estimate.

A. Multi-GPU Server without batching

The server throughput is given by:

$$T_{max} = \min\{T_C, T_G(N_G)\}, \tag{1}$$

where T_C is the CPU throughput necessary to manage GPUs, e.g., kernel invocation, and $T_G(N_G)$ is the total GPU throughput achievable for this application for N_G GPUs. Intuitively, if the CPU cannot keep up feeding the GPUs with new kernels, the system throughput will be constrained by the CPU.

For a perfectly scalable system the throughput with N_G GPUs is:

$$T_G(N_G) = N_G T_G(1). \tag{2}$$

The CPU single-core throughput is given by:

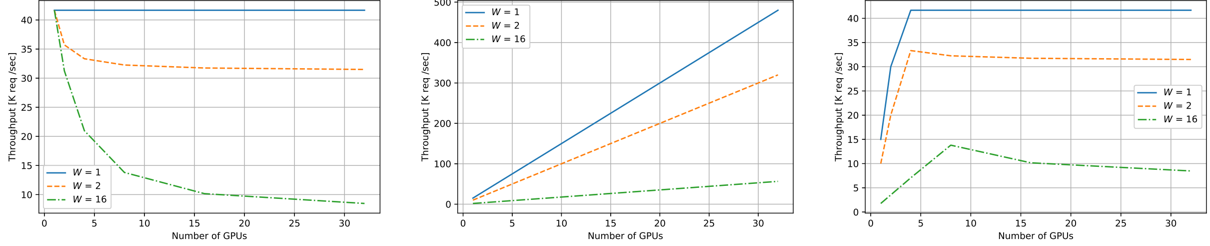
$$T_C = \frac{1}{(O_q + O_k)K(N_G)} \tag{3}$$

where O_k and O_q are the overheads of kernel invocation and completion status query respectively, and $K(N_G)$ is the number of kernel invocations for this workload with N_G GPUs. For simplicity we optimistically assume that each status query always retrieves a completed task, but in practice multiple queries per task might be necessary.

For a given input request, we necessarily invoke one `filter` kernel and one `reduce` kernel. The number of `search` kernels depends on the input: it is 1 if all the W clusters are located on the same GPU, and W if they are in multiple GPUs. Assuming random cluster distribution, the average number of kernel invocations is equivalent to the expectation of a number of non-empty bins out of N_G bins if we randomly assign each of the W balls to a bin. Then,

$$K(N_G) = 2 + N_G\left(1 - \left(1 - \frac{1}{N_G}\right)^W\right) \tag{4}$$

Note that this expression converges to 2+W in a limit when $N_G \to \infty$.



(a) CPU management scaling. (b) GPU performance scaling. (c) System scaling is capped by the CPU

Figure 2: Scalability of a CPU-driven server. W is the number of `search` kernels, i.e., the number of concurrently invoked GPUs.

Combining Eq 4 and Eq 3 yields the expression for the expected CPU management throughput. This, in turn, allows us to compute T_{max} – the maximum server throughput as a function of N_G and W.

The graphs in Figure 2 are obtained by using Eq 3, Eq 2 and Eq 1 respectively, while assigning values measured in a real system for the kernel overheads and k-NN throughput on NVIDIA K80 GPU. Specifically, $O_k = 5\mu sec$, $O_q = 3\mu sec$ and $T_G(1) = 15Kreq/sec$.

As expected, the throughput saturates when the CPU is no longer able to feed GPUs. Moreover, it even *drops* for larger values of W and more GPUs, because the chances that the `search` will run on fewer than W GPUs is decreasing: most `search` kernels are executed on different GPUs, increasing the overall GPU invocation overhead.

Takeaway: multi-GPU scaling of the conventional CPU-centric design for the k-NN algorithm is inherently limited by the CPU management throughput, and requires more CPUs to support more GPUs.

B. Multi-GPU server with batching

One approach to reduce the GPU management overhead is to process incoming requests in batches. As a result, the GPU runs larger kernels and the invocation overheads are amortized. The downside is that batching significantly increases the response latency, which makes it unsuitable for low-latency servers, as we show below.

Assume that the batch size is B requests, and the system receives the requests at the rate equal to its maximum throughput $T_G(N_G)$. We denote by $t_W(B)$ the mean waiting time of a request for batch aggregation, and by $t_C(B)$ the average computation time of a batch. Then, the mean total service latency is given by:

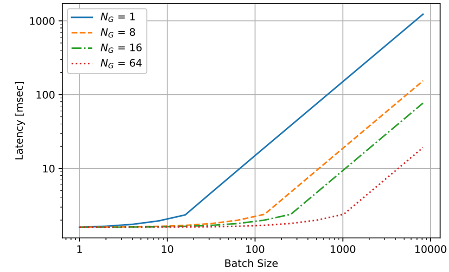
$$t_S(B) = t_W(B) + t_C(B). \quad (5)$$

The average waiting time per request is:

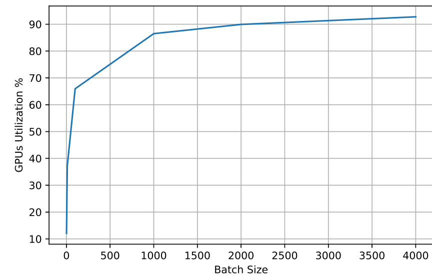
$$t_W(B) = \frac{1}{B}\sum_{n=0}^{B-1}\frac{n}{T_G} \quad (6)$$

Each request has to wait for the whole batch to finish before it can advance to the next stage. Thus,

$$t_C(B) = \lceil\frac{B}{B_{concurrent}}\rceil t_s, \quad (7)$$



(a) Mean per-request service latency vs. batch size



(b) Utilization of 16 GPUs vs. batch size

Figure 3: Server performance analysis with batching for $W = 2$

where $B_{concurrent}$ requests are processed by the GPUs concurrently, and t_s is the time to serve a sub-batch.

We omit the rest of the derivation for the lack of space, but the intuition is that batches delay execution significantly, and larger batches on fewer GPUs even more so as their constituent requests cannot be processed in parallel.

Figure 3a shows the latency as a function of the batch size for different number of GPUs. As expected, large batch size causes the request latency to grow. For example, the batch of 2048 requests as used in prior works [26] invoked with $W = 2$ and 16 GPUs leads to 25msec request latency. Recall that it is an optimistic estimate that ignores data transfer overheads. In fact, this result is not far from the actual value we observed: 32msec on 16 GPUs with batches of 1000 requests (see §VI-H).

Unfortunately, this latency is too high when using the system for real interactive services. As shown in prior works [29], [33], the end-to-end latency of more than 50 milliseconds negatively affect user engagement and service

revenue. Since about 90% of the time is spent in frontend servers [47], the k-NN computations are left with the latency budget of few milliseconds.

Small batches cause load imbalance. Reducing the batch size might reduce latency, but it hurts throughput. Specifically, small batches suffer more from load imbalance among GPUs running the `search` stage leading to the tail effect [15]. Intuitively, requests in a batch must be invoked on multiple GPUs and cannot proceed until they all complete. The smaller the batch, the more pronounced will be the effect of the momentary load imbalance between the GPUs.

This intuition matches the theoretical analysis below. We calculate the utilization of all the GPUs as a function of the batch size: a batch B produces BW queries for the `search` stage. These queries are distributed randomly between N_G GPUs, with $\frac{BW}{N_G}$ queries per GPU on average.

Given a batch B, b_i is the number of queries destined to the ith GPU. b_i is a multinomial random variable (with $n = BW$, $k = N_G$, $p_i = \frac{1}{N_G}$).

Since all the GPUs wait for the largest batch to finish, the utilization of GPU$_i$ is:

$$U_i = \frac{b_i}{\max_i(b_i)} \tag{8}$$

Hence, the average utilization of all GPUs for this batch:

$$U = \mathbf{E}\left(\frac{\frac{1}{N_G}\sum_{i=0}^{N_G-1} b_i}{\max_i(b_i)}\right) = \frac{BW}{N_G}\mathbf{E}\left(\frac{1}{\max_i(b_i)}\right) \tag{9}$$

We numerically evaluate this function while varying the batch size and show it in Figure 3b for 16 GPUs. Note that reducing the batch size below 1000 results in poor system utilization and consequently lower throughput.

Takeaway: large batches increase the service latency, making it a poor choice for interactive network services. Small batches improve latency over large batches, but lead to low GPU utilization due to load imbalance. Today's solution is to *dedicate more CPU cores to GPU management to scale to more GPUs*, which is costly and inefficient.

Centaur aims to offer a more efficient alternative by focusing on the root cause of the problem: CPU management overhead. It suggests a design which minimizes the CPU involvement, allowing requests processing one-by-one without batching, while achieving low latency, high throughput and multi-GPU scaling.

IV. Design

At a high level, Centaur provides a runtime for multi-GPU execution of data flow graphs each representing one k-NN query, without using the CPU for task invocation and data transfers.

The runtime comprises three main components:

Worker threadblock and affinity-aware scheduler. Each task is executed by a generic GPU worker threadblock implemented as a persistent kernel. The number of threads is determined by the application task this worker is assigned to execute.

Workers are interconnected via *gpipes* which work akin to UNIX pipes as we explain later. The worker may receive inputs from and send outputs to multiple other workers. Depending on the specific task, the worker may wait to receive more than one input to invoke its own compute function. Similarly, the worker may send to a subset of other workers connected to its outputs. For example, a `filter` node is connected to several `search` nodes.

Affinity-aware scheduler determines the destination of the next workers to which the outputs are sent. In particular, the scheduler sends the output of a `filter` worker to the `search` workers on the GPUs which store their respective clusters. If such locality constraints are irrelevant, (i.e., there are multiple `search` nodes running on the same GPU), the scheduler chooses the worker via some load-balancing policy (e.g., random or power-of-two choices).

Inter-worker communication. *Gpipe* is the main communication mechanism between workers. A gpipe is a single-producer single-consumer queue which connects two workers and allows data transfer, synchronization and in-place data manipulation. Workers may reside in the same GPU, or across different GPUs.

The gpipe mechanism is specifically designed to enable atomics-free implementation, thus it works correctly over PCIe bus that lacks atomic operations. To achieve high performance, gpipes leverage peer-to-peer DMA between the GPUs where available, otherwise falling back to the compatibility mode which transfers data via CPU memory. We discuss the implementation details in Section V-A.

`reduce` **worker and space reservation.** This is a special worker with additional synchronization requirements. On the one hand, `reduce` is blocked until inputs from all the workers arrive. On the other hand, different workers connected to it may be handling *different* requests, and may produce results out of order. As a result, some gpipes may run out of space while the `reduce` worker is waiting for the inputs that belong to the same request, leading to a deadlock. This is, in fact, a known problem in data flow systems.

Centaur introduces a *reservation mechanism* to prevent `reduce` deadlocks. This mechanism reserves space in the specialized `reduce` input gpipes for each incoming request the moment the request enters the `filter` worker. Only when the space is reserved the request begins its processing. One useful byproduct of this design is that the `search` workers are provided with the exact location to place their outputs, thus saving space and polling overheads in the `reduce` stage, as we explain in Section V-B.

A. Mapping execution on GPUs

Single GPU. Each worker is assigned to run one of the tasks: `filter`, `search` or `reduce` and configured with

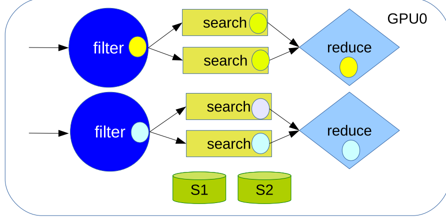


Figure 4: k-NN execution on a single GPU. The tags (circles) of the same color represent tasks belonging to the same request.

the number of threads appropriate for the task. In addition, the configuration specifies the ratio between the number of workers of each type. The best ratio would balance the amount of computations between the stages, allowing no worker stalls and better utilization. Centaur invokes the workers on the GPU according to the requested ratio. All the workers are invoked to run concurrently, eliminating GPU kernel invocations at runtime.

Each worker running a specific task is connected to its neighbors in the data flow graph. The GPU effectively runs multiple such graphs, as many as there are `filter` nodes. However, to achieve better load balancing and higher utilization, we allow worker sharing between the graphs. For example, a few `filter` workers can be connected to a few `search` workers in an all-to-all fashion. We call the set of interconnected workers *a worker group*. For perfect load balancing, all `filter` workers can be connected to all `search` workers, but this would incur high memory and scheduling overhead.

Example. Figure 4 shows the system with two worker groups, each implementing a full data flow graph. S1 and S2 represent the data shards on that GPU.

Multiple GPUs. First, a single-GPU setup is replicated across all available GPUs. Then, we connect the workers across the GPUs as follows. All `filter` workers in the same worker group on one GPU are connected to all `search` workers in a single worker group in all the other GPUs. Similarly we connect `search` and `reduce` workers of the same worker group across all the GPUs.

The data is split evenly among the GPUs. Each GPU holds an index table of all the clusters and their physical locations. This table is used by the affinity-aware scheduler.

Example. Figure 5 shows the connectivity between the workers when using two GPUs each with its own shard. We show only connections from GPU0 to GPU1, and omit the symmetric connections from GPU1 to GPU0 for clarity. Note that the `reduce` workers can now get tasks from the `search` workers located in both GPUs.

B. Connecting GPUs to the outside network

Each `filter-reduce` pair is associated with its own GPUnet [30] network socket to receive a request from and send a response to the client. This design follows the design of GPU servers presented in GPUnet. Specifically, it assumes

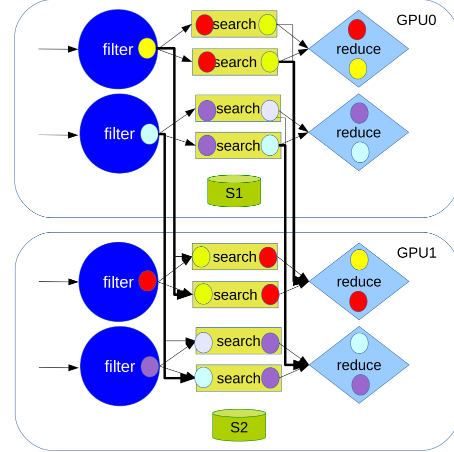


Figure 5: Execution on multiple GPUs with shards split among them.

that there is an external load balancer that routes the clients to different server sockets, and that all the clients maintain persistent (always open) connections to the server to mitigate connection establishment overheads.

C. Scalability Analysis

The primary factor that limits Centaur's scalability is the allocation of gpipes for connections between `filter` stages and `search` stages in all the GPUs. The number of gpipes connected to `search` stage increases linearly with the number of GPUs. Thus, a `search` stage gets occupied by scanning for a new job instead of doing useful work.

We show that as long as *the load increases proportionally with the number of GPUs*, aka weak scaling, there are no runtime overheads due to the increased number of gpipes.

Each `search` stage has $N_G N_f$ input gpipes, where N_G is the number of GPUs, and N_f is the number of `filter` workers per GPU. There are N_s `search` workers in total. We assume that each request to `filter` results in execution of W `search` workers, that the client load λ_{client} creates $W\lambda_{client}$ load on the `search` workers, which is evenly distributed among the N_G GPUs. Hence the load per gpipe λ_q is given by:

$$\lambda_q = \frac{W\lambda_{client}}{N_G^2 N_s N_f}$$

We assume that each `search` worker serves requests at rate of μ_s. We also assume that the service rate is split evenly between gpipes, and so the service rate per gpipe is $\mu_q = \frac{\mu_s}{N_G N_f N_s}$.

We now ask: what is the expected number of times n_{trials} the `search` worker checks its input gpipes until a task is found? This quantity determines the overhead of polling and connects it to the number of GPUs.

To answer these question we assume the approximation of M/M/1 queues. For simplicity we assume that a `search` worker randomly selects a gpipe and keeps scanning until a task is found.

In this case,

$$n_{trials} = \frac{\mu_q}{\lambda_q} = \frac{\mu_s N_G}{\lambda_{client}}$$

Therefore, if the system load λ_{client} grows linearly with N_G, the number of trials until a job is found is independent of the number of GPUs.

V. Implementation

We implement Centaur using CUDA for NVIDIA GPUs. Here we describe the main implementation details. Notice however that although the implementation uses CUDA, the problem we address is a fundamental problem (as we show in III).

A. Gpipes

There are several types of gpipes, each providing the communication channel for a different setup.

Local gpipes. We implement a gpipe as a lock-less single-producer single-consumer ring buffer, similar to those used in prior works [30], [24]. The buffer resides in the GPU global memory, and its size is set at setup time to accommodate load fluctuations, and also depends on the size of the objects transferred between the stages.

This design trades space for performance, as it requires a private gpipe per consumer-producer pair while eliminating the contention. It is a conscious design choice, however, which allows low-overhead fine-grain inter-task communications. On the other hand, as the number of gpipes grow, system scaling might get affected due to the increased polling overhead for consumers. We analyze these overheads below.

Cross-GPU gpipes. When the producer and the consumer reside on different GPUs, we place the ring buffer in the consumer's GPU, and use GPUDirect to directly access the gpipe in the consumer's GPU memory. Placing the buffer closer to consumer has two benefits: it allows low-overhead polling of the head/tail pointers by the consumer, which is a frequent operation when the consumer is waiting for new tasks; and employs remote-write-local-read principle [30] which optimizes the system performance for synchronization over PCIe.

Fallback: cross-GPU gpipes without GPUDirect GPU support. Unfortunately, GPUDirect is not always available. Specifically, it is not supported across QPI for different NUMA nodes, and does not allow a GPU to connect to more than 8 other GPUs [6]. This limitation is likely to be removed in the future to meet the growing demands for multi-GPU machines [7], [1], [2].

We replicate ring buffer in the CPU and in the consumer's GPU memory. The producer writes to the host, while a host runs a helper thread that copies the tasks from the CPU memory into the consumer's GPU buffer. Direct GPU memory access from the CPU is implemented using NVIDIA gdrcopy [36] kernel module.

Memory management. gpipe serves not only for inter-task communications but also as a memory manager. gpipe stores all its data by value. To save redundant memory copies, the producer access gpipe to allocate a gpipe slot, to which it then writes directly without using private memory. When the data is ready, the producer commits, thereby making the slot available to the consumer. The consumer, similarly, operates on the slot, and commits it back when done.

This mechanism simplifies the intra-task memory management, and eliminates the overheads associated with it.

B. Reduce stage

Tracking sub-tasks for the same request. The `reduce` stage aggregates multiple results pertaining the processing of the same request by previous stages in the data flow graph. To maintain this semantics while allowing concurrent handling of multiple requests by different graph stages, a naive solution would annotate each request with its ID when it enters the system. Instead, we introduce a simple reducer buffer mechanism which we describe next.

Reducer buffer and reservations. Reducer buffer is used instead of gpipe to connect the reducer. It is an array broken into slots, one per request, each intended to store all the partial results from the previous stages that pertain the same request. The memory in the reducer buffer is *reserved* for each request it is received. While simple, this reservation mechanism works well for k-NN. The slot cannot be used by any other request. When the tasks preceding the `reduce` worker terminate, their outputs are placed directly into the correct slot in the reducer buffer.

This mechanism has several benefits. First, it prevents the deadlock due to insufficient space in `reduce` gpipes. In addition, it saves the overhead of searching for the partial results belonging to the same request. Last, it does not require maintaining unique request ID.

C. GPU invocation

Centaur uses task-to-threadblock mapping, while allowing different number of threads for workers of different types. To invoke them on the GPU, we consolidate the workers of the same type in the same kernel. Doing so is essential to reduce the total number of concurrently running kernels, which is limited to 32 (the size of the Hyper-Q) [6], [38].

This design creates an opportunity for additional performance optimizations in terms of the task placement on GPU streaming multiprocessors (SMs). Specifically, earlier works demonstrate substantial improvements in GPU performance when threadblocks running *different* tasks are placed on the same SM [40]. The reason is that different tasks are likely to use different hardware resources, thereby easing resource contention.

We therefore strive to schedule that workers such that different tasks end up on the same SM if possible. Specifically, we experimentally find that the NVIDIA GPU scheduler

places threadblocks of a kernel in a round-robin fashion over the available SMs. Thus, if a kernel with n `search` threadblocks is invoked after the kernel with n `filter` threadblocks, the `search` and `filter` will be placed on the same SM if n is the number of SMs in the GPU. This simple heuristic achieves the best results, also because it balances the load between different SMs.

For example, in our setup we choose 1024,480 and 32 threads for `filter`,`search` and `reduce` workers respectively. This configuration ensures that all the stages will fit on the same SM when invoked as three kernels with 15 threadblocks each, to fit on the 15SMs in the GPU.

VI. Evaluation

A. Evaluation Objectives

We aim to answer the following questions:

1) How well does Centaur-based server scale compared to the CPU-driven design? (§VI-F)
2) How sensitive Centaur-based server is to the availability of CPU cores and CPU speed (§VI-G)
3) What is the performance cost of using Centaur design in smaller scale systems (§VI-H)

B. Competing Designs

We implement three servers, each following a different design approach: CPU-centric micro-batching server, CPU-centric server without batching, and Centaur.

Micro-batching server design. A batching server is a poor match for interactive network services due to its high latency, but we provide it as a reference point for the maximum achievable throughput.

Our implementation uses two CPU threads.The *receiving thread* receives requests from the network, and waits until a predefined number of requests arrive in a batch, or for a predefined timeout, whichever comes first. It passes the batch to the *processing thread*, and continues with the next batch.

The *processing thread* sends the batch to a randomly selected GPU to run the `filter` stage, waits for the results, partitions them into sub-batches according to the location of the shards required for each request in the batch, and sends each sub-batch to the correct GPU to perform the `search` stage. After completion of all the `search` tasks for all the sub-batches, the *processing thread* invokes the `reduce` stage on the CPU (running it on the GPU is slower due to overheads).

CPU-driven server design. This server is a multi-threaded, highly optimized lock-free server. We allocate 4 *IO threads*: two *receiving threads* and two *sending threads*. In addition we allocate a *control thread* for each GPU. We found that fewer I/O threads fail to saturate the server and shift the bottlenecks to the I/O path.

All the threads multiplex the processing of multiple requests to achieve high GPU utilization. We use an event-driven server design [52] in which the server maintains a state machine for each request. Each one of the GPU threads is assigned a set of active requests it should handle. The thread scans through the state machines of all its requests until it finds a request with a pending action, and performs this action. Example actions are: receiving a request from a network, sending computation to the GPU, checking the completion of a computation request, or sending a response to the client.

C. Setup

We perform the experiments on two different systems described in Table I. First, we use a server in our lab to run a complete end-to-end evaluation by measuring the request latency and throughput as observed by a client running on a separate machine connected via Infiniband. The Centaur's server implementation uses GPUnet GPU-side networking library to handle requests on GPU. The PCIe PLX switch in our server provides stable and symmetric performance across all the GPUs and the network adapter.

As our machine is limited to 6 GPUs (3 x K80 boards), we perform additional evaluations on an Amazon EC2 P2.16xlarge instance that scales to 16 GPUs with 64 vCPUs. Unfortunately, EC2 does not offer Infiniband NICs, and therefore we cannot use GPUnet. To achieve consistent and reliable results, we generate the load locally on the server, and emulate the network traffic by transferring client requests and server responses in/from running GPU kernels as if they were received/sent via network.

Another limitation of the large-scale multi-GPU setup is that NVIDIA GPUs do not support more than eight peer-to-peer GPU connections each [6]. This constraint, in turn, puts a hard limit on scaling experiments with Centaur when using peer-to-peer gpipes. Therefore, for the experiments with more than 9 GPUs, we resort to using gpipe compatibility mode which streams the data via CPU memory using CPU helper threads (see §V-A for more details). Figure 6a shows that with 8 GPUs this CPU-assisted compatibility mode results in slightly lower throughput than when using peer-to-peer gpipes. Therefore, the multi-GPU evaluation using the compatibility mode provides a *pessimistic* estimate of the server performance on future hardware which hopefully will not have these peer-to-peer scaling restrictions.

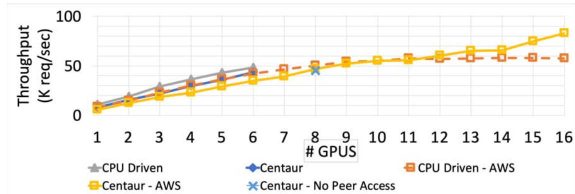
All the GPUs are configured to the highest supported clock rate. The CPUs are also set to the highest frequency unless stated otherwise. EC2 vCPUs do not support frequency scaling and run in the default mode.

D. Workload

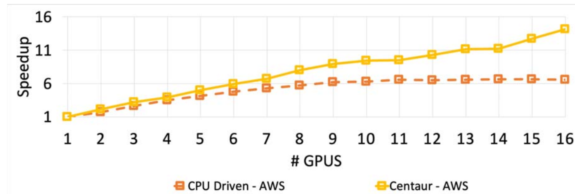
We evaluate the k-NN server by using it for running a low-latency image similarity search service. In our experiments

Location	Software	CPU	GPU	PCIe	Network
Local	CUDA 8.0, Ubuntu 14.04, kernel 3.13.0 . NVIDIA Driver v375.39, GPUnet	6 cores (12 hyperthreads) Intel Xeon E5-2620 v3 (Haswell)	3x NVIDIA K80 (6 GPUS)	PLX PEX 8747	Mellanox Connect-IB 56Gbps
Amazon EC2 (p2.16xlarge instance)	CUDA 7.5, Amazon Linux AMI release 2016.03, kernel 4.4.16-27.56, NVIDIA Driver v352.99	64 virtual CPUs, custom Intel Xeon E5-2686 v4 (Broadwell)	8x NVIDIA K80 (16 GPUs)	unknown	none

Table I: Experimental setup.



(a) Server throughput scaling. Higher is better.



(b) Server speedups over one GPU. Higher is better.

Figure 6: Scalability of different server designs.

we configure the algorithm to use two clusters to search for the match ($W = 2$ in Section III), and $k = 1$ (top 1 result).

We use a 1 million images SIFT data set [28], and split it equally across all the available GPUs used in the experiment. Each image in the image data set is represented as a 128-D SIFT [34] vector. The data set is pre-processed offline to group similar images into separate clusters. The clustering uses the k-Means algorithm [35], [48]

E. Measurement methodology

For the experiments with the local server we measure per-request latency and average server throughput as observed by a remote network client. The client generates the load at a requested throughput level. Unless stated otherwise, the throughput reported for different designs is the maximum achievable throughput. The throughput-latency curves (Figure 9) confirm that the system achieves maximum throughput while retaining the desired low per-request latency.

F. Performance scaling

We perform the experiment on both the local machine with 6 GPUs and a remote EC2 machine with 16 GPUs. We present the absolute throughput results in Figure 6a and the speedup over a single GPU for each design in Figure 6b.

We first validate that the Amazon EC2 experiments produce meaningful results by comparing them with the local execution. We observe that the Amazon EC2 instance is consistently slower than the local server, both in CPU-driven and Centaur-based servers. Due to poor visibility into the actual hardware resources in EC2, we cannot explain this performance gap. However, since it equally affects both server designs, we believe that its effect on the reported scaling is negligible.

Centaur's throughput surpasses that of the CPU-driven design starting from 11 GPUs, with the performance advantage growing up to 35% with 16 GPUs. In fact, the CPU-driven design throughput reaches its maximum with 9 GPUs, with its resource efficiency of only 56% with 16 GPUs.

According to Figure 6b, the CPU-driven server fails to scale linearly beyond 4 GPUs, and levels off entirely for 9 GPUs. This result is consistent across a variety of assignments of CPU management threads to CPU cores.

In contrast, *Centaur achieves perfect scaling for up to 9 GPUs, and with the resources efficiency of 91% with 16 GPUs.* We speculate that the main reason for reduced scalability is the lack of peer-to-peer inter-GPU communications that is not available beyond 9 GPUs. Therefore, we believe that Centaur may achieve even better scalability if this constraint is lifted in future GPU architectures.

Notice that the classic CPU-driven design outperforms Centaur if the number of GPUs is low. This is due to the following inefficiencies in Centaur's design:

- All the threadblocks of each stage have the same size. This leads to underoccupied GPUs due to failure to allocate the maximum possible number of threads.
- The different stages of the computation (`filter`, `search`, `reduce`) are not perfectly balanced, which results in wasting resources waiting to the slowest stage.
- Each threadblock in Centaur design indefinitely performs a single computation stage. This prevents the CUDA runtime from allocating resources on demand, which results in wasting GPU resources.

We are planning to address the above disadvantages in future work. However, in spite of these inefficiencies, Centaur outperforms the CPU-driven design when the number of GPUs is high.

Performance scaling vs. GPU compute load. How much does the performance scaling depend on the computational load per kernel? The answer to this question helps assess the benefits of Centaur servers for setups where interactions with GPUs are less frequent, possibly easing the CPU management bottlenecks.

We configure the k-NN algorithm to increase the number of clusters (W) in the filter search, thus proportionally increasing the load on GPUs.

We run the GPU scaling experiments in EC2 for each value of W. The results in Figure 7 confirm that the

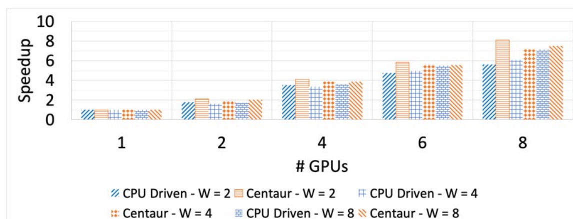


Figure 7: Server scalability for different levels of GPU compute loads while varying the number of `search` tasks. Higher is better.

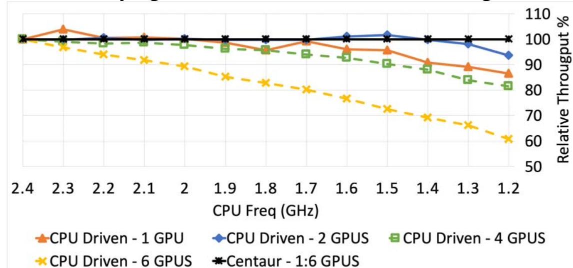


Figure 8: Server performance for different *CPU* frequencies. Higher is better.

scalability of the CPU-driven server improves gets closer to that of Centaur for higher GPU load (larger W). However, *Centaur scales better even for higher GPU loads.*

G. Server performance sensitivity to CPU capacity

Using fewer CPU cores. We measure the maximum throughput that different server designs can achieve as a function of the number of available *CPU* hardware threads. We use the local server with 6 GPUs. We note that the experiment is performed with a hyperthreaded CPU with 12 logical CPU cores. When hyperthreading is off, however, Centaur consistently outperforms the CPU-driven server by over 7× when using all 6 CPU physical cores. Thus, by using hyperthreading we provide a more favorable execution environment for the CPU-driven design.

To perform the measurements we gradually turn off CPU logical cores via `procfs`. The results in Figure 1 highlight the main performance benefits of the Centaur's CPU-less design. The throughput degradation of the CPU-driven server compared to Centaur ranges from 40× for one logical core to over 4× for eight. It finally regains its performance back when using all the 10 cores, which matches the number of CPU threads used with the 6-GPU CPU-driven implementation. In contrast, *Centaur maintains stable performance regardless of the number of dedicated CPU cores*.

To quantify the influence of the number of dedicated CPU cores on request latency, we run a throughput-latency experiment while varying the number of cores as above. Each point in the graphs in Figure 9 is obtained as follows: the client applies a certain load on the server, sends 100K requests, measures the latency of each request, and the server throughput. In figure 9 we see that the latency degradation of the CPU-driven design due to the lack of CPU cores is

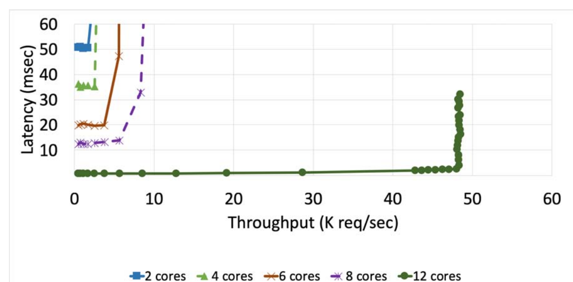


Figure 9: Throughput-Latency curves of CPU-driven design under different number of dedicated CPU cores. Centaur throughput and latency *do not* depend on CPU availability (not shown).

even worse than the drop in its throughput. For example the latency spikes 20× with six CPU threads. The latency of Centaur server remains the same regardless of the number of CPU cores (not shown).

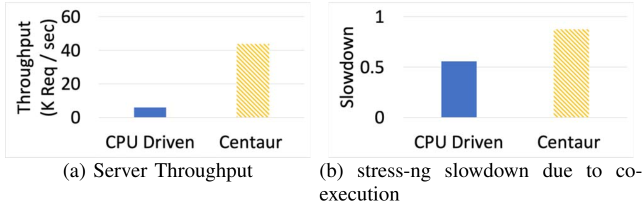
Server throughput vs. CPU frequency. We evaluate the effect of CPU frequency on the server performance. The goal is to provide a more fine-grain assessment on of the performance as the function of CPU speed. We run this experiment on the local server with 6 GPUs, because EC2 does not enable frequency scaling for vCPUs. We take extra care to obtain reliable CPU-driven server results with the hyper-threaded CPU. Specifically, we pin all four I/O threads to two physical cores and keep them running at the highest frequency. We then reduce the frequency of all the other logic cores (effectively 4 physical cores), and measure the system throughput.

We perform the same experiment for different number of GPUs and present the results in Figure 8. The performance of the CPU-driven server decreases due to CPU frequency scaling as we add more GPUs to the system. Indeed, the server loses up to 40% of its throughput with 6 GPUs when the CPU is slowed down by half. This is because additional GPUs increase the frequency of inter-GPU synchronizations, which amplifies the overall effect.

Similarly to the previous experiment, *Centaur server performance is insensitive to the changes in the CPU frequency*.

Server throughput vs. CPU Load. We evaluate the effect of additional CPU load on the throughput of the server. We use the `stress-ng` [3] benchmarking tool to spawn 12 compute-intensive CPU threads, stressing all the CPU cores in our setup. In parallel, we invoke the GPU server with 6 GPUs and measure its maximum throughput. We also measure the number of compute operations per second executed by the `stress-ng` to estimate the slowdown of CPU computations due to the execution of the GPU server.

We present the results in Figure 10. Under stress, the CPU-driven server experiences a 8.2× slowdown in its throughput, while the Centaur server is not affected at all. `stress-ng` itself is dramatically slowed down when co-running with the CPU-driven server, while only slightly affected when co-running with Centaur.



(a) Server Throughput (b) stress-ng slowdown due to co-execution

Figure 10: Co-execution of the server with a CPU demanding workload. Centaur not affected. Higher is better.

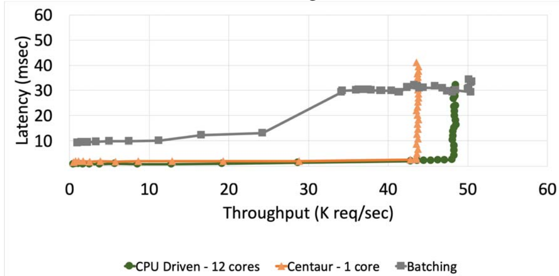


Figure 11: Throughput-Latency curves of CPU Driven, micro-batching and Centaur-based designs.

H. Performance costs of GPU-centric design

Latency and throughput for different server designs. We evaluate the three server designs described in VI-B, with the goal to show their latency-throughput tradeoff.

The adaptive micro-batching is configured to aggregate up to 1000 requests for 10ms, whichever comes first. Alternative configurations have resulted in lower maximum throughput and are not shown.

Figure 11 shows that both the CPU-driven and Centaur-based designs achieve similar average latency of around 2ms/request in steady state, whereas the latency of the batching approach is larger by up to 15×, and grows dramatically the load increases.

The CPU-driven design attains 10% higher maximum throughput than Centaur. This overhead is the cost Centaur pays for better scaling and independence from the host CPU's characteristics (Centaur scaled better though and will become favorable after a higher number of GPUs).

VII. Related work

To the best of our knowledge, Centaur is the first system to show the benefits of GPU-centric low-latency multi-GPU server architecture. We build upon prior art in several areas.

GPU server architectures. Several earlier works evaluate the benefits of running server workloads on GPUs: Kim et al. [30] report performance advantages using GPU for running a face verification server; Agrawal et al. show [9], [8] boost in power efficiency and operational costs for web servers and text similarity search server; Unlike Centaur, these works do not deal with data affinity and multi-GPU scaling.

Task-parallel processing on GPUs. Chatterjee et al. [13] implement on-GPU work stealing mechanism for irregular workloads. Tzeng et al. [50] introduce an on-GPU runtime for task-parallel execution on GPUs for dependent tasks, advocating for the use of persistent kernels, on-device load balancing and task dependency handling. We employ similar ideas in our work, extending the system to a multi-GPU environment.

GPU-centric designs. Recent works [30], [44], [45], [14] advocate to transform the GPU into a first-class system processor by offering I/O abstractions to GPU kernels, allowing CPU-less execution of GPU-native programs. Centaur builds on the same concept, but applies it in the context of multi-GPU servers for improved scalability.

Inter-GPU communication. GPU-side networking libraries [30], [14], [24] allow GPUs to communicate via Infiniband without CPU involvement, but do not provide efficient inter-GPU communication in a single machine. MVAPICH2 [51] provides GPU-aware MPI primitives. NCCL [39] implements efficient collective communication in multi-GPU machines. However, all of these solutions are triggered from the host.

OpenCL has the concept of CLpipes [5] for inter kernel communication, However, unlike gpipe which connects running kernels, it requires explicit synchronization of both ends of the CLpipe by stopping the kernels to achieve consistent updates.

VIII. Conclusions

We presented Centaur, a CPU-less multi-GPU server design for latency-sensitive, memory-demanding k-NN server. We show that Centaur scales perfectly to 9 GPUs and achieves 91% efficiency on 16 GPUs, with 35% higher throughput than a highly-optimized CPU-driven design, while providing 2msec average latency per request. Centaur's performance is agnostic to the CPU load, frequency or the number of dedicated cores, which makes it over an order of magnitude faster than the CPU-driven design under the same load conditions.

Centaur demonstrates the benefits of fully autonomous GPU operation in a highly demanding application using current GPU generations, but it will likely become even more advantageous with new generations of fast GPU-GPU interconnects such as NVLINK. We believe, therefore, that this work provides a valuable point in the design space of multi-GPU systems, and hope to motivate further research in this direction.

References

[1] "Amazon EC2," 2006, https://aws.amazon.com/ec2/.

[2] "Google Cloud Platform," 2011, https://cloud.goog.com/.

[3] "stress-ng," 2014, https://openbenchmarking.org/test/pts/stress-ng.

[4] "Mellanox BlueField," http://www.iptronics.com/page/products_dyn?product_family=256&mtag=soc_overview, 2015.

[5] "OpenCL 2.0," https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf, 2015.

[6] "CUDA Programming Guide," 2017, http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[7] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[8] R. Agrawal, "K-nearest neighbor for uncertain data," *International Journal of Computer Applications*, vol. 105, no. 11, 2014.

[9] S. R. Agrawal *et al.*, "Rhythm: Harnessing data parallel hardware for server workloads," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 19–34.

[10] Amazon Web Services, Inc., "Amazon EC2 P2 Instances," n.d., https://aws.amazon.com/ec2/instance-types/p2/.

[11] M. E. Belviranli, L. N. Bhuyan, and R. Gupta, "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 57, 2013.

[12] R. Buyya, "High performance cluster computing," *New Jersey: F'rentice*, 1999.

[13] S. Chatterjee *et al.*, "Dynamic task parallelism with a GPU work-stealing runtime system," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2011, pp. 203–217.

[14] F. Daoud, A. Watad, and M. Silberstein, "GPUrdma: GPU-side library for high performance networking from GPU kernels," in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2016, p. 6.

[15] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[16] D. Foley, "NVLink, Pascal and stacked memory: Feeding the appetite for big data," *Nvidia. com*, 2014.

[17] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic time," *ACM Trans. Math. Software*, vol. 3, no. SLAC-PUB-1549-REV. 2, pp. 209–226, 1976.

[18] K. Fukunage and P. M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE transactions on computers*, no. 7, pp. 750–753, 1975.

[19] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, 2008, pp. 1–6.

[20] V. Garcia *et al.*, "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," in *2010 IEEE International Conference on Image Processing*. IEEE, 2010, pp. 3757–3760.

[21] A. Gionis *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, no. 6, 1999, pp. 518–529.

[22] Google, "GPUs on Compute Engine," n.d., https://cloud.google.com/compute/docs/gpus/.

[23] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *2012 Innovative Parallel Computing (InPar)*. IEEE, 2012, pp. 1–14.

[24] T. Gysi, J. Bär, and T. Hoefler, "dCUDA: hardware supported overlap of computation and communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 52.

[25] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "Memcachedgpu: Scaling-up scale-out key-value stores," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 43–57.

[26] K. Jang *et al.*, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. Berkeley, CA, USA: USENIX Association, 2011, pp. 1–14. Available: http://dl.acm.org/citation.cfm?id=1972457.1972459

[27] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[28] H. Jégou *et al.*, "Searching in one billion vectors: re-rank with source coding," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 861–864.

[29] F. Khan, "The cost of latency," *Online at: https://www.digitalrealty.com/blog/the-cost-of-latency/*, 2015.

[30] S. Kim *et al.*, "GPUnet: Networking abstractions for GPU programs." in *OSDI*, vol. 14, 2014, pp. 6–8.

[31] Q. Kuang and L. Zhao, "A practical GPU based kNN algorithm," in *Proceedings. The 2009 International Symposium on Computer Science and Computational Technology (ISCSCI 2009)*. Citeseer, 2009, p. 151.

[32] S. Liang *et al.*, "CUKNN: A parallel implementation of k-nearest neighbor on CUDA-enabled GPU," in *2009 IEEE Youth Conference on Information, Computing and Telecommunication*. IEEE, 2009, pp. 415–418.

[33] G. Linden, "Marissa Mayer at web 2.0," *Online at: http://glinden. blogspot. com/2006/11/marissa-mayer-atweb-20. html*, 2006.

[34] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[35] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.

[36] NVIDIA, “gdrcopy,” n.d., https://github.com/NVIDIA/gdrcopy.

[37] NVIDIA, “GPUDirect,” n.d., https://developer.nvidia.com/gpudirect.

[38] NVIDIA, “Whitepaper: NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110/210,” n.d., https://images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[39] NVIDIA Corporation, “NCCL: Optimized primitives for collective multi-GPU communication,” 2016.

[40] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, “Improving GPGPU concurrency with elastic kernels,” in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 407–418.

[41] C. J. Rossbach *et al.*, “PTask: operating system abstractions to manage GPUs as compute devices,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 233–248.

[42] C. Sanders, “Azure N-Series: General availability on December 1,” 2016, https://azure.microsoft.com/en-us/blog/azure-n-series-general-availability-on-december-1/.

[43] S. Shahar, S. Bergman, and M. Silberstein, “ActivePointers: a case for software address translation on GPUs,” in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 596–608.

[44] M. Silberstein *et al.*, “GPUfs: integrating a file system with GPUs,” in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 485–498.

[45] M. Silberstein *et al.*, “GPUfs: Integrating a file system with GPUs,” *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 1, p. 1, 2014.

[46] M. Silberstein *et al.*, “GPUnet: Networking abstractions for GPU programs,” *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 3, p. 9, 2016.

[47] S. Souders, “The performance golden rule,” *Online at: https://www.stevesouders.com/blog/2012/02/10/the-performance-golden-rule/*, 2012.

[48] H. Steinhaus, “Sur la division des corp materiels en parties,” *Bull. Acad. Polon. Sci*, vol. 1, no. 804, p. 801, 1956.

[49] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A parallel programming standard for heterogeneous computing systems,” *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[50] S. Tzeng, B. Lloyd, and J. D. Owens, “A GPU task-parallel model with dependency resolution,” *Computer*, vol. 45, no. 8, pp. 34–41, 2012.

[51] H. Wang *et al.*, “MVAPICH2-GPU: optimized GPU to GPU communication for infiniband clusters,” *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 257, 2011.

[52] M. Welsh, D. Culler, and E. Brewer, “SEDA: an architecture for highly concurrent server applications,” in *ACM SIGOPS European Workshop*, vol. 35, no. 5, 2001, pp. 230–243.

[53] T. T. Yeh *et al.*, “Pagoda: Fine-grained GPU resource virtualization for narrow tasks,” in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 221–234.