Hetero-Edge: Orchestration of Real-time Vision Applications on Heterogeneous Edge Clouds

***Abstract*—Running computer vision algorithms on images or videos collected by mobile devices represent a new class of latency-sensitive applications that expect to benefit from edge cloud computing. These applications often demand real-time responses (e.g., <100ms), which can not be satisfied by traditional cloud computing. However, the edge cloud architecture is inherently distributed and heterogeneous, requiring new approaches to resource allocation and orchestration. This paper presents the design and evaluation of a *latency-aware* edge computing platform, aiming to minimize the end-to-end latency for edge applications.**

The proposed platform is built on Apache Storm, and consists of multiple edge servers with heterogeneous computation (including both GPUs and CPUs) and networking resources. Central to our platform is an orchestration framework that breaks down an edge application into Storm tasks as defined by a directed acyclic graph (DAG) and then maps these tasks onto heterogeneous edge servers for efficient execution. An experimental proof-of-concept testbed is used to demonstrate that the proposed platform can indeed achieve low end-to-end latency: considering a real-time 3D scene reconstruction application, it is shown that the testbed can support up to 30 concurrent streams with an average per-frame latency of 32ms, and can achieve 40% latency reduction relative to the baseline Storm scheduling approach.

I. Introduction

In the last decade, hosting major computing jobs on central clouds has proven effective since central clouds generally have abundant computing/storage resources [1]. Recently, as mobile devices and Internet of Things (IoT) sensors keep increasing, an unprecedented amount of data have been generated, and a new class of applications is quickly looming on the surface. These applications involve performing intensive computations on sensor data (typically image/video) in real time, aiming to realize much faster interactions with the surrounding physical world and thus providing truly immersive user experiences.

With this trend, central clouds may no longer be the appropriate platform for supporting these applications, in view of performance limitations caused by network bandwidth and latency constraints. For example, a mobile AI assistant application needs responses within tens of milliseconds. An autonomous driving system, as another example, may generate gigabytes of data every second by its stereo camera or LIDAR, and needs responses within a few milliseconds. Yet, for a client instance in New Jersey which connects to Amazon EC2 cloud servers located in West Virginia, Oregon and California, the round-trip latency *alone* is 17, 104 and 112ms, with achievable bandwidths of 50, 18 and 16Mbps, respectively. In order to support these emerging edge applications, edge cloud computing has been proposed as a viable solution [2], [3], [4], which moves the computing towards the network edge to reduce the response latency while also avoiding edge-to-core network bandwidth constraints.

Several aspects of edge computing have been studied in the recent years. For example, the study in [5] proposes systems that enable rapid virtual machine (VM) handoff or live migration across edge clouds. Edge computing also raises many interests from the analytic perspective as it introduces a new communication and computing paradigm [6]. In order to minimize service delay in edge computing, works in [7], [8], [9] introduced optimization frameworks to minimize transmission delay and/or processing delay for mobile users. Applications proposed in [4], [10] offloaded intensive computing tasks to edge clouds to achieve low latency image processing.

Despite the earlier and ongoing work on various aspects of edge computing, the problem of how to efficiently deploy these new edge applications within an edge cloud has not been systematically studied. Simply duplicating the successful cloud computing design will not work for the edge applications. This is mainly due to the highly heterogeneous nature of edge clouds. Unlike central clouds, edge clouds are often comprised of heterogeneous computation nodes with widely diverse network bandwidths. For example, the studies in [11], [12], assume the computation nodes and their interconnects are relatively homogeneous in central clouds, while the edge servers considered in [5], [13] exhibit widely varying capabilities. Thus, an important new challenge associated with edge clouds is that of efficiently orchestrating these heterogeneous resources in order to meet application latency constraints.

To address this problem, we set out to build and test such an edge computing orchestration platform. Our design is driven by the requirement of deploying and accelerating this new class of edge applications – e.g., processing large volumes of data such as video data generated by mobile/IoT sensors (including 3D cameras) in real time. We first build an edge cloud testbed that consists of four different CPU settings, four different GPU settings and five different link bandwidth settings. On these nodes, we run Apache Storm [14] as the baseline distributed edge computing framework. Apache Storm provides real-time support, but has an implicit assumption that the underlying computing/networking resources are homogeneous. Also, it does not provide proactive support for GPUs. In this work, we address these shortcomings. Note that our platform design is not specific to Apache Storm. In fact, it can easily interface with other distributed computing frameworks such as Apache Flink.

The design of Hetero-Edge mainly focuses on distributed resource orchestration for edge computing. Specifically, we

intend to answer the following important questions. Firstly, if an edge cloud consists of both GPUs and CPUs, when do we serve requests on GPUs and when do we use CPUs? How do we partition our jobs so that we can most efficiently utilize the available resources? Secondly, after partitioning the job to several pipelined and parallel tasks, how can we map them to appropriate computing nodes (including both GPUs and CPUs) to minimize their overall latency? Thirdly, how can we effectively prevent a parallel task from completing significantly slower than its peers and becoming a straggler [15]? Since edge clouds are highly diverse, the likelihood of having stragglers is much higher than in a homogeneous setting. By carefully studying these questions, we devise the resource orchestration schemes in Hetero-Edge, featuring: (1) matching a task's resource demand with the underlying node's resource availability, (2) matching a task's workload level with the underlying node's resource availability, and (3) suitably splitting work on processors with vastly different processing power (GPUs vs CPU).

We have implemented an example edge application on our Hetero-Edge testbed, i.e., real-time 3D scene reconstruction from two stereo video streams [16]. We use this example application to drive our evaluation effort. We emulate a realistic setting where user streams dynamically join and leave the system and track the detailed system performance for two hours. We show that with seven edge servers, we are able to support all the streams that arrive within the two hours with an average per-frame latency of 32 milliseconds. We also show that our schemes can effectively prevent straggler tasks and can shorten a frame's latency by 40% compared to the state-of-the-art Storm schedulers when we have heterogeneous resources. We summarize our contributions as follows:

- We have designed and implemented a distributed edge computing platform Hetero-Edge that extends the capabilities of a stream processing framework, Apache Storm, for use in heterogeneous distributed edge environments with a focus on latency reduction. We will make our code open source and share it through GitHub.
- We have devised a dynamic task topology generation scheme, a *latency-aware* task scheduler and a proportional workload partitioning scheme, which, when combined, can proactively minimize the overall latency in heterogeneous distributed edge environments.
- We have implemented 3D scene reconstruction as a driving application example and have shown how to optimize this category of applications on our edge platform to achieve low latency. Note that we only use this application as an example to drive the discussion and evaluation. Other real-time edge vision applications will be readily supported in the same way without changing our system in any way.
- We have learned valuable lessons in deploying real-time edge applications on heterogeneous edge servers. Such lessons will help us realize the wide adoption of edge computing.

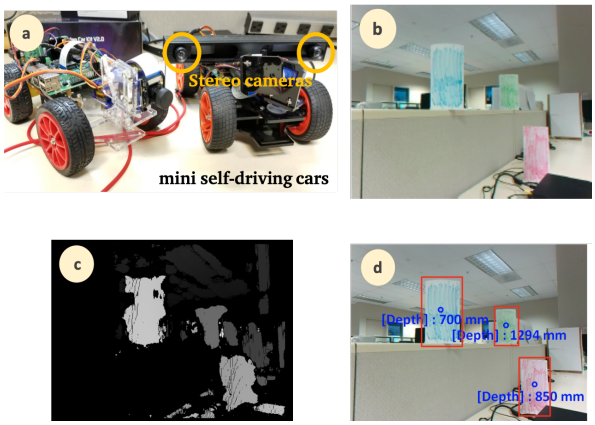


Fig. 1: An example 3D reconstruction application. (a) is the mini self-driving cars with the stereo cameras, (b) is the raw input from the left stereo camera, (c) is the resultant disparity map by which we can infer the depth of each pixel, and (d) is the reprojection result which shows the depth of objects in the real world.

II. System Model

In this section, we discuss the emerging real-time edge vision applications, summarize the system assumptions for edge clouds, and present the architecture of Apache Storm.

A. Characteristics of Real-Time Edge Vision Applications

In this study, we focus on supporting a new class of applications, which we refer to as real-time edge vision applications. These applications usually take image/video data that are captured by mobile or IoT devices as input, perform complex processing on each frame and have stringent latency requirements. For examples, consider real-time 3D scene reconstruction [16], virtual reality [9], augmented reality [17], vision-based autonomous driving [18], etc. Though diverse, these applications share quite a few common characteristics, such as low latency requirement and high computation/networking demand. Importantly, they are usually parallel and pipelined by nature.

In the rest of this paper, we will use 3D scene reconstruction [16] as the example use case to drive the discussion and evaluation. We believe that the ability to perform low latency 3D scene reconstruction not only can help build mobile augmented reality or virtual reality to enhance immersive user experience, but also can enable an array of applications with tight feedback loops. For instance, 3D scene reconstruction for autonomous vehicles is used to detect the relative positions of the obstacles and trigger collision-avoidance reactions [19]. It typically consists of the following steps: (1) offline camera calibration, (2) stereo image rectification, (3) disparity calculation, and (4) 3D re-projection. The essence is to infer the disparity of each pixel from multiple 2D images and then use this extra dimension data to reconstruct the object jointly. Disparity measures the difference in retinal position between two points that correspond to the same point on the real object. By definition, a more remote point tends to have a smaller disparity value than a nearer one. This step mainly decides

the quality of the reconstruction effect and usually involves heavy computation overhead.

Figure 1 illustrates a 3D reconstruction example that we have implemented in our laboratory.

B. System Assumptions for Edge Clouds

The definition of edge clouds varies from study to study, ranging from a smart traffic light that has some computing capability [2] to a small-scale data center [3]. In this study, we assume an edge cloud consists of multiple edge servers within radio access networks, e.g., eNodeBs, that are available for hosting computing tasks [20]. Different from traditional central clouds that are generally equipped with homogeneous and well-provisioned resources, edge clouds are opportunistic and heterogeneous by nature. An edge cloud is usually composed of nodes with varying computing capabilities (CPUs with different cores, GPUs, etc.), storage capacities (hard drives, memories, etc.), and network capacities.

We further assume there is no resource contention among different application processes on the same node. This assumption can be achieved by deploying edge applications in light containers such as Kubernetes [21] or NVIDIA docker [22] that supports GPU-level isolation.

In this work, we have implemented a Hetero-Edge testbed consisting of 7 edge servers. An edge server has one of the following CPU configurations: (1)Xeon E5-2630, 2.40GHz, 32 cores, (2)Xeon E5-2698, 2.30GHz, 64 cores, (3)Xeon W5590, 3.33GHz, 16 cores and (4)i7-3770, 3.40GHz, 64 cores. Their GPU configurations are the following: (1) no GPU, (2) Tesla K40 GPU, (3)Tesla K80 GPU and (4)Tesla C2050 GPU. Each computation node can have the following link bandwidth configurations: .5Gbps, 1Gbps, 2Gbps, 5Gbps, and 10Gbps.

C. Background on Apache Storm

In Hetero-Edge, we choose to adopt Apache Storm [23], a popular distributed real-time data stream processing framework, to support the distributed processing. Apache Storm has been deployed in various scenarios such as algorithmic trading, real-time video processing, distributed remote procedure call, etc. We choose Apache Storm because it offers the following advantages: (1) designed for pursuing ultra-low latency, (2) easily scale to dynamically available resources, (3) no need to store any intermediate results (the main bottleneck of those distributed computing frameworks with the MapReduce design, e.g., Hadoop [24])

Here, we briefly overview its architecture and go over the core components that are relevant to our study. Apache Storm consists of a single *master* node and a pool of *slave* nodes. The master node is in charge of distributing tasks to the slave nodes while a slave node manages *worker* processes. Each worker process further manages *executor* threads, each of which executes a task in a given task graph.

In the logical layer, Apache Storm is composed of three core components: *Spout*, *Bolt* and *Topology*. A spout is usually the source of a data stream, a bolt is an intermediate processing function, and the topology (represented by a Directed Acyclic Graph (DAG)) steers the data flow according to the job logic. We present the overview of Apache Storm in Figure 2.

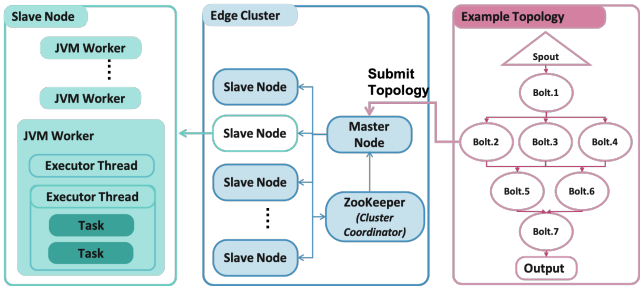


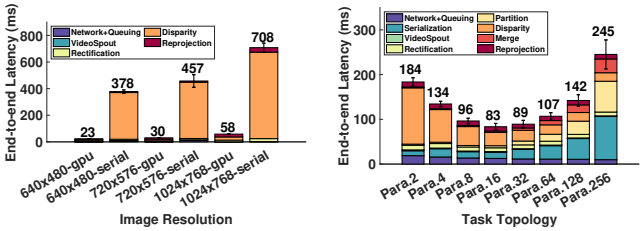
Fig. 2: Apache Storm Architecture and Example Topology. The rightmost figure shows an example topology that consists of a user-specified DAG. A user needs to submit this topology to the master node in a Storm cluster. Then, the master node distributes the tasks of the topology to the pool of its slave nodes who take the job of executing those tasks. The detailed system design of a slave node is shown in the leftmost figure.

When deploying an application on edge clouds with Apache Storm, we need to consider the following main issues:

1) *Task topology construction*. In this step, we construct one or more suitable task topologies for the application, considering both data parallelism and task parallelism. We take into consideration the resource diversity when generating the task topology – we choose to have different task topologies for the same application under different resources. See Section III-B.
2) *Task scheduling*. When a topology is constructed, we assign each task bolt in the topology to a computation node based on certain scheduling principles. The default Storm task scheduler does not consider resource diversity and simply assigns nodes in a round-robin fashion. Such a schedule leads to long latency in heterogeneous edge cloud. In this study, we devise a *latency-aware* task scheduler that can considerably outperform the Storm default scheduler and the state of the art *resource-aware* scheduler. See Section III-C.
3) *Stream grouping*. When a frame arrives at the system, this step considers how the output stream of a bolt (e.g., bolt 1 in Figure 2) is partitioned among the next step data parallel bolts (e.g., bolts 2, 3 and 4 in Figure 2). If the resource variation of these bolts is not considered, one of the data parallel bolts can become much slower than others, thus slowing down the entire processing. In this study, we devise a proportional partitioning scheme that can alleviate this problem. See Section III-D.

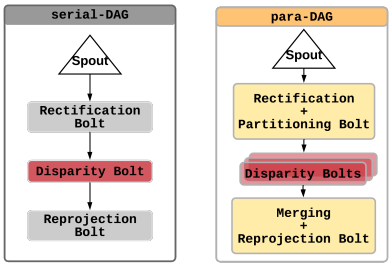
III. Detailed Hetero-Edge Design

In this section, we present the detailed design of Hetero-Edge. Hetero-Edge has many design details, and we specifically focus on those that can shorten the end-to-end application latency. Below, we use the 3D scene reconstruction application to drive our discussion. However, our design is not limited to this particular use case but is applicable to all the real-time edge vision applications that we describe in Section II-A.



(a) (b)

Fig. 3: (a) Latency breakdown of the 3D reconstruction application under different configurations; (b) Latency of different degree of data parallelism in the para-DAG with the resolution of 640x480. The 16-way para-DAG gives the lowest latency.



(a) (b)

Fig. 4: We consider two task topologies: (1) serial-DAG whose bottleneck bolt (i.e., the disparity bolt) is usually scheduled on GPUs, and (2) para-DAG whose bolts are scheduled on CPUs.

A. Preparation: Bottleneck Analysis

Before presenting our design, we first break down the application into several functions – for the 3D reconstruction application, we have rectification, disparity calculation, and re-projection. We profile the processing latency of these functions with different image resolutions on Xeon E5-2360, and present the measured latency values in Figure 3(a). We find that the disparity calculation function is the predominant bottleneck, which becomes even more pronounced as the image resolution goes up.

We next execute the disparity calculation function on Tesla K40 GPU and find its latency drops significantly. For example, 94% latency with the resolution of 640x480 drops.

In our subsequent steps, we will use the above latency information to make scheduling decisions. Usually, it is a good practice to perform such a bottleneck analysis before trying to deploy an application. Fortunately, there are various tools we can use for this step. For example, we can use NVIDIA OPENACC [25] to identify the execution bottleneck as well as function dependency of an application.

B. Task Topology Construction

Given the above function breakdown and the identified system bottleneck, we consider the following two practical topologies when with different available computing/networking resources and will evaluate these choices in Section IV-B.

Serial Topology (serial-DAG): In a serial-DAG, we can either implement all functions to a single bolt for the benefit of introducing little inter-communication overheads or assign an individual function to different bolts for the merit of generating a pipelined flow to decrease task queuing time for an available processor, as shown in Figure 4(a). Importantly, a serial-DAG is often more practical when we execute the bottleneck function on GPUs than on CPUs. In this case, the non-bottleneck functions can be scheduled either on the hosting CPU or even on a remote CPU.

Parallel Topology (para-DAG): Since the disparity step predominates the entire CPU processing latency, we next consider a topology that enables data parallelism to accelerate the computation, which we call a para-DAG (which is usually demanded when GPU is unavailable). With an n-way para-DAG, we partition the data set into n partitions and feed each partition into a data-parallel bolt that runs the same disparity calculation function. We illustrate this topology in Figure 4(b).

Different partitioning strategies, e.g., range partition, hash partition or composite partition, can be adopted according to different algorithm designs of bottleneck functions. In this specific application, considering the disparity calculation function processes each row of the image independently, we partition the image in a row-major fashion. Say the original image has $M \times N$ pixels, and we partition the image into n ways. Then each partition has $\frac{M \times N}{n}$ pixels. Also, in order to guarantee each partition has enough pixels to generate the disparity map, we need to include the boundary rows in both partitions. As a result, additional pixels need to be included in each partition. In this case, the total number of pixels a partition thus becomes $(\frac{M \times N}{n} + M \times \frac{d}{2})$ where d is the searching block size defined in the block matching algorithm that calculates disparity. The resulting para-DAG topology is shown in Figure 4(b). Due to data partitioning, we need to introduce two more processing steps to the topology: partitioning and merging.

Considering each partition will combine more boundary rows that introduce additional computation/networking overheads (which is a usual case among any image partition algorithms), we further examine the topology to decide the suitable partitioning degree n. We choose the 640x480 image resolution at $1fps$, and measure the overall latency as well as the latency for each component for different n values. The measured results are shown in Figure 3(b). We find that the 16-way para-DAG gives the lowest latency, 83ms in our case. Note we need to explore particular best value n for different applications.

In reality, if we need to support many more applications in the edge clouds, we can use tools such as those in [25] for automatic DAG partition and parallelization. Finally, we remark that data partitioning does not only apply to the para-DAG, but it should also be considered in generating the GPU code for the serial-DAG.

C. Task Scheduling

After generating the two topologies for the application, we next enter the task scheduling phase and try to schedule both topologies on the available edge server nodes. That is, we need to assign each bolt in a topology onto a suitable edge server.

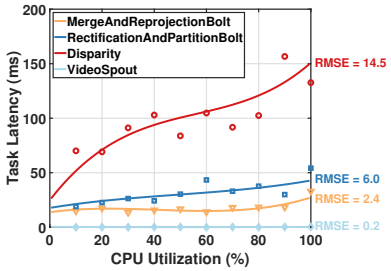


Fig. 5: Estimated processing latency vs CPU utilization. Using the measured latency values under different CPU utilization, we build a 3-order polynomial curve to estimate the processing latency under any utilization.

Apache Storm provides two task schedulers: the *round-robin* scheduler and the *resource-aware* scheduler. The *round-robin* scheduler is the default Storm scheduler. It allocates bolts to computing nodes in a *round-robin* fashion, oblivious of the bolt resource demand and the available node resource. It may allocate, for example, a computation intensive task on a node that is short of CPU cycles, leading to long execution latency. To address the problem with the *round-robin* scheduler, the *resource-aware* scheduler [26] selects a node with the most available CPU resource and fills it up before assigning any task to any other node. Here, users estimate the available CPU resource on each node as well as the requested CPU resource for each bolt at the compile time. Without a reliable estimation mechanism, it can either lead to resource waste (by overestimating the requested CPU) or result in resource contention (by overestimating the available CPU). Furthermore, neither *round-robin* nor *resource-aware* scheduler considers GPU scheduling.

Our proposed task scheduler has the following three main components: (1) a mechanism to estimate the performance and resource requirement of a task/bolt (we use these two terms interchangeably), (2) a tool to track the available resources within an edge cloud, and (3) a *latency-aware* task scheduling algorithm. Below we describe these components one by one.

1) Estimating a Bolt's Performance and Resource Demand: Before devising our task scheduling scheme, we first need to develop a mechanism to estimate a bolt's performance (i.e., processing latency) and resource demand (i.e., memory usage, network usage, GPU/CPU usage). Among these four items, a bolt's memory usage and network usage remain constant no matter where the bolt is executed, which we can capture through ThreadMXBean.getThreadAllocatedBytes(), and count the byte array length of output stream in Storm.

The other two items – a bolt's processing latency and processor usage (we consider both GPUs and CPUs here) – are not only determined by which server the bolt is executed on but also the load on the server. As such, in the profiling phase, we run each bolt on every edge server (including both CPUs and GPUs) at different processor utilization. We increase the processor utilization level from 0% to 100% with an increased interval of 10%. For each bolt-processor-utilization combination, we measure the latency (by recording the elapsed time through JAVA Timer API) and processor time (which is the amount of GPU/CPU time dedicated to this bolt process and can be captured through ThreadMXBean.getThreadCpuTime(), while GPU utilization can be read from nvidia-smi).

We can then feed the measured values into n-order polynomial curves (n=3 in this work because it provides the lowest predictive mean squared error) to derive an estimation model for each bolt. This model takes a particular processor's resource level and utilization as input, and gives an estimation of the bolt's processing latency and consumed processor utilization. Figure 5 plots the latency estimation models for the four bolts of the 3D reconstruction application on the Xeon E5-2630 processor.

Finally, note that the estimation model in VideoStorm [11] does not consider the processor load, but we believe processor load is an important parameter to consider when estimating a bolt's latency and resource demand.

2) Real-time Edge Resource Monitoring: We periodically collect the available resources for each edge server. For this purpose, the following actions are performed: (1) collecting the port bandwidth of a node using the *iPerf/scp* utility; (2) collecting CPU frequency and utilization of a node using the *lscpu* utility; and (3) collecting the current memory usage of a JVM worker from the Storm's daemon. (4) collecting GPU utilization of a node using the *nvidia-smi* utility.

3) Proposed Heuristic: Latency-Aware Task Scheduling (LaTS): As mentioned earlier, we construct two task topologies for each application: a serial-DAG and a para-DAG. In the task scheduling phase, we need to consider both topologies.

A serial-DAG only makes sense if we schedule the bottleneck bolt on a GPU; otherwise, the latency will be too long. The other non-bottleneck bolts will be scheduled on CPUs because the functions associated with these bolts usually receive much lower speedup by switching to GPU.

Below, we first describe our CPU scheduling strategy – how we schedule a list of bolts to CPUs – and we then briefly describe our GPU scheduling strategy which essentially uses the same technique as CPU scheduling. The CPU scheduling part is needed when we schedule non-bottleneck bolts for a serial-DAG as well as when we schedule all the bolts for a para-DAG. Given a pool of bolts to be scheduled on the CPUs (we refer to them as CPU bolts below), we rank them in the descending order of the required CPU time – we estimate their required CPU time on the same processor. Then we schedule the bolt from the bolt list one by one.

For a given bolt and an edge node, we perform the following two types of latency estimation. First, we measure the node's CPU utilization and use the latency estimation model as shown in Figure 5 to estimate the processing latency. Next, we measure the node's available bandwidth and use the bolt's output streaming size to estimate the network transmission latency. The total latency for this bolt-node combination is then the sum of these two types of latency values. In this way, we can estimate the total latency of this bolt on every edge node in the system.

We assign the bolt to the node that gives the minimal latency. After the assignment, we also need to decrease the

available resources on that node by removing the amount of resources consumed by this bolt (processor utilization, memory and bandwidth). We repeat the above process until we finish scheduling all the CPU bolts.

We next describe how we schedule the bottleneck bolt in a serial-DAG to the appropriate GPU. The idea is very similar to the above CPU scheduling. We choose the GPU that gives the shortest overall latency (we estimate both processing latency and networking latency here) to host the bottleneck bolt.

We refer to the above task scheduling algorithm as *LaTS*. Ideally, when a new user stream connects to the edge cloud, we need to run the algorithm on both topologies and choose the assignment that has the lowest latency. Note that when the edge cloud becomes larger or when many uses are connected to the edge cloud, it may be too costly to exactly follow this procedure. In this case, we need to develop faster heuristics to perform task scheduling. We describe and evaluate such heuristics in Sections IV-B and IV-C (check Figure 11).

D. Stream Grouping

Apache storm provides the flexibility of specifying how to steer a bolt's output stream to the connecting bolt(s). This decision becomes particularly important when the connecting bolts run in the data parallel mode – they run the same task function but on their own partition of the data set. For example, in our para-DAG shown in Figure 4(b), we can specify which disparity bolts we choose to use and how to partition the stream among the chosen disparity bolts. This decision can vary from frame to frame.

A good stream grouping algorithm can take into consideration the resource variation among the bolts and then partition the data according to ensure these data-parallel bolts finish at about the same time. If one such bolt is scheduled on a node with fewer resources and incurs a much longer latency than its peers, then it becomes a straggler[15] and slows down the entire processing. In general, their succeeding bolt has to wait for all the data-parallel bolts to finish before it can start processing. We note that our LaTS task scheduling algorithm is already effective in avoiding stragglers because it tries to place each bolt on the node that yields the shortest latency. However, when the edge servers have vastly different resource levels, stragglers cannot be avoided by the task scheduling phase alone. Clever stream grouping techniques thus become critically important in avoiding stragglers.

Proportional Partition Stream Grouping (Pro-Par): The default stream grouping algorithm in Apache Storm partitions the data set equally among the bolts and cannot avoid stragglers if the bolts have varying computing capabilities. In Hetero-Edge, we propose to adopt a proportionally partitioning method, *Pro-Par* in short. In Pro-Par, we periodically estimate the computing capacity of those nodes that host the data parallel bolts and then partition the stream in such a fashion that a node's partition is proportional to its computing capacity.

By default, we equally partition the stream among all the data parallel bolts and measure their processing latency periodically. When the gap between the fastest bolt and the slowest bolt exceeds a certain threshold, we trigger Pro-Par. Specifically, let us suppose each disparity bolt's latency is $\{t_1, ..., t_m\}$ where m is the degree of data parallelism.

We adopt the min-max normalization method to remap the m different latency values to the range (0,1). We estimate the computing capability of each node c_i as $\frac{1}{m \times t_i}$, and the overall computing capability C provided by m nodes is $\sum_{i=1}^{m} c_i$. Then we partition the stream proportionally to each node's computing capacity and transmit the resulting partition to each bolt.

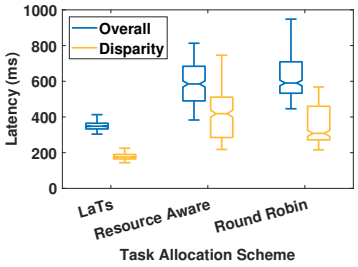


Fig. 6: Comparing the latency of *LaTS*, *round-robin* and *resource-aware* task scheduler at a low frame rate of $1fps$ with high resource heterogeneity.

IV. Measurements and Experiments

We have implemented the proposed techniques (described in Section III) on the Hetero-Edge testbed. In this section, we present our evaluation effort in detail. Our evaluation has the following three components: (1) evaluating the proposed schemes using an edge cloud that does not have GPUs, (2) evaluating the proposed schemes using an edge cloud with GPUs, (3) putting everything together and taking a close look at the Hetero-Edge run-time dynamics.

A. Evaluation of Hetero-Edge with only CPU and Network Heterogeneity

LaTS Better Handling Resource Heterogeneity: We compare three task schedulers – i.e., *round-robin*, *resource-aware*, and *LaTS* – in terms of the ability to handle CPU and network diversity. The image resolution is 1440x1080. We consider a very low frame rate, $1fps$, so that we can focus on the latency alone. Here, we consider seven edge servers (Xeon E5-2630, 2.4GHz) with different CPU and network resources. Five nodes have 10Gpbs network links and no other processes. Two nodes have 1Gbps links and their CPUs are already partially occupied (available CPU utilization 10%-40%). We find that LaTS performs the best: 40.0% shorter than round-robin, and 41% shorter than *resource-aware* as shown in Figure 6. In addition, we use the yellow bars to show the latency distributions of the 16 disparity bolts that are in data parallel mode. We find that LaTS leads to lower average latency for the 16 disparity bolts and a much lower gap between the fastest and slowest bolts, which is the key to minimizing the overall latency. Note that we have also tried other edge cloud configurations and have observed similar trends. As a result, we believe that LaTS can better address CPU and network diversity among edge servers and lead to a shorter end-to-end latency. It also does a better job preventing stragglers.

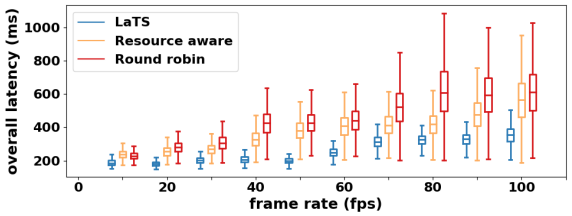


Fig. 7: Comparing the latency of *LaTS*, *round-robin* and *resource-aware* at high system load by increasing the single stream's frame rate.

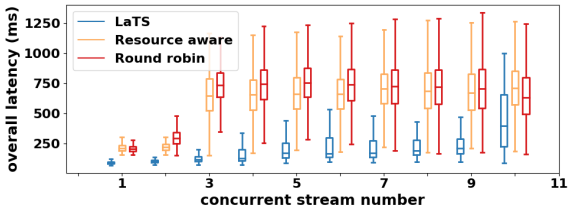


Fig. 8: Comparing the latency of *LaTS*, *round-robin* and *resource-aware* at high system load by increasing the number of concurrent streams.

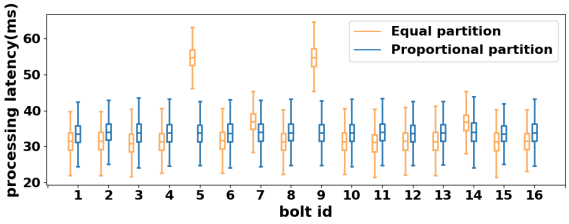


Fig. 9: Comparing the latency distribution of data parallel bolts with Pro-Par and equal partition stream grouping.

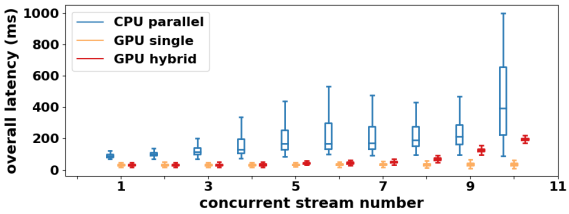


Fig. 10: Comparing the latency of CPU Parallel, GPU Single and GPU Hybrid when we have utilized both GPUs and CPUs.

LaTS Better Handling High System Load: Next, we look at how these three schemes handle a higher load.

In the first set of experiments, we have 4 Xeon E5-2630 (2.4GHz) edge nodes, two with 10Gbps links and 100% CPU available, one with 1Gbps links and 100% CPU available, and one with 10Gbps links and 20% CPU available. We have a single stream and increase the stream's frame rate, from 10 to 150fps, with an increase of 10fps. The image resolution is 640x480. The results are shown in Figure 7. We find that LaTS continuously outperforms the other two schedulers by more than 25%.

In the second set of experiments, we fully load Hetero-Edge with the setting as the resource heterogeneity experiment in Figure 6. We fix each stream's frame rate as 30fps and increase the number of concurrent streams, from 1 to 10, with an increase of 1 stream. The results are shown in Figure 8. We find that LaTS can support up to 9 concurrent streams without the latency significantly going up. On average, its latency is 66% lower compared to *round-robin* and 61% to *resource-aware*.

Pro-Par Better Handling Stragglers: Next we evaluate the effectiveness of Pro-Par in mitigating the stragglers. In this set of experiments, we target a case wherein the straggler bolts continue to slow down because their nodes have insufficient resources (which is already the best effort by LaTS). The equal partition stream grouping leads to two straggler bolts (5, 9 in Figure 9) that require 41% more time compared to other bolts. With our proportional partition stream grouping, it balances the workload based on the computing capability of each bolt and therefore effectively avoids stragglers. Its slowest bolt is 36% faster than the equal partition steam grouping scheme.

B. Evaluation of Hetero-Edge with GPU, CPU and Network Heterogeneity

In this subsection, we consider edge clouds that have heterogeneous CPUs, networks and GPUs. Since Storm does not consider GPU by default, we only focus on our own schemes in this subsection. We compare the latency results of the following three scheduling strategies: (1) *CPU Parallel*: para-DAG bolts running on CPUs, (2) *GPU Single*: serial-DAG bolts running on the same node (with the bottleneck bolt running on the GPU while non-bottleneck bolts running on the host CPU) and (3) *GPU Hybrid*: the bottleneck bolt of a serial-DAG running on one node's GPU while non-bottleneck bolts run on other nodes' CPUs. Please note that these three schemes are special cases of our LaTS scheme. By understanding which of these three strategies is faster and when to use which, we can greatly speed up LaTS as we do not need to search every possible combination.

We increase the number of streams from 1 to 10, with the frame rate for each stream to be 30fps. Figure 10 shows the comparison results. As expected, CPU Parallel gives much longer latency than the two GPU solutions. GPU Hybrid performs better than GPU Single when the concurrent stream number is more than 8 due to the over-utilization of the host CPUs. When the stream number is less than 8, two GPU schemes perform similarly as a result of the low communication overhead.

C. Supporting real-time edge vision applications through Hetero-Edge

After evaluating each technique in different settings, we finally put together everything and evaluate whether our Hetero-Edge platform can effectively support the intended real-time edge vision applications (3D construction in our example). Suppose we provide 3D reconstruction services to nearby mobile users with our edge cloud (that involves all the nodes in our testbed, including both GPUs and CPUs). Each interested user connects to our edge cloud and starts a stream; after a certain number of frames, the user ends the stream. In our experiments, we use the following synthetic workload: each user stream has a frame rate of 30fps and a video resolution of 640x480; each user initiates a session of 1 minutes and

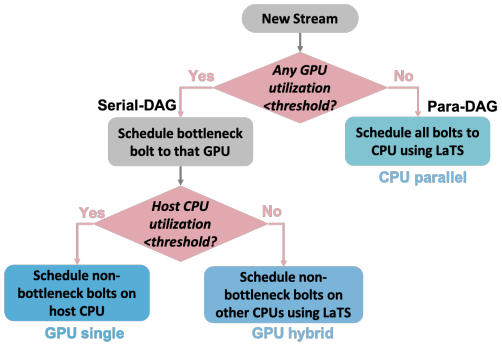


Fig. 11: When a new stream arrives at the system, we follow this flow to find out which scheme we are going to use to schedule this stream: GPU Single, GPU Hybrid or CPU Parallel. This flow is faster than exactly going through the LaTS scheduler.

leaves the system; the user session arrival process follows a Poisson distribution with an average arrival rate of 20.

Suggested by the results reported in Figure 10, our edge server adopts the policy described in Figure 11. Following this policy, when the GPUs and their host CPUs are rather empty, we choose GPU Single to schedule streams. Slowly, the CPUs on those nodes that have GPUs will become busy, and we can switch to GPU Hybrid to schedule the arriving streams. Finally, when all the GPUs get busy, we resort to CPU Parallel to serve the subsequent streams. We run our service for 2 hours, and report important run time parameters in Figure 12. From top to bottom, we have (1) the number of connected streams, (2) the number of GPU Single, GPU Hybrid, and CPU Parallel streams, (3) GPU utilization, (4) CPU utilization of those nodes that have GPUs, (5) CPU utilization of those nodes that do not have GPUs, and (6) the histogram of the end-to-end per frame latency.

We would like to highlight the following observations from the results. Firstly, the average per-frame latency is 32ms, which we believe is satisfying for many real-time edge vision applications. Secondly, our platform can effectively schedule streams across highly heterogeneous computation nodes – on average, we have the most GPU Single streams and the least CPU parallel streams. Thirdly, we find that the GPUs have lower utilization than their host CPUs because these CPUs spend more time processing the non-bottleneck bolts than GPU processing the bottleneck bolts. As a result, when host GPUs become fully utilized, GPU Hybrid becomes useful.

Lessons Learned: By offering the edge service for 2 hours, we have learned a few lessons regarding application deployment on edge servers. The most valuable lesson we have learned is that it is important to include GPUs in an edge cloud. It can help significantly reduce the per-frame latency. However, we need to pay extra attention to GPU scheduling as well as coordinating GPUs and CPUs to finish one job efficiently. Finally, optimizing CPU scheduling is also very important, such as carefully matching the task demand with resource availability and matching the workload level with resource availability.

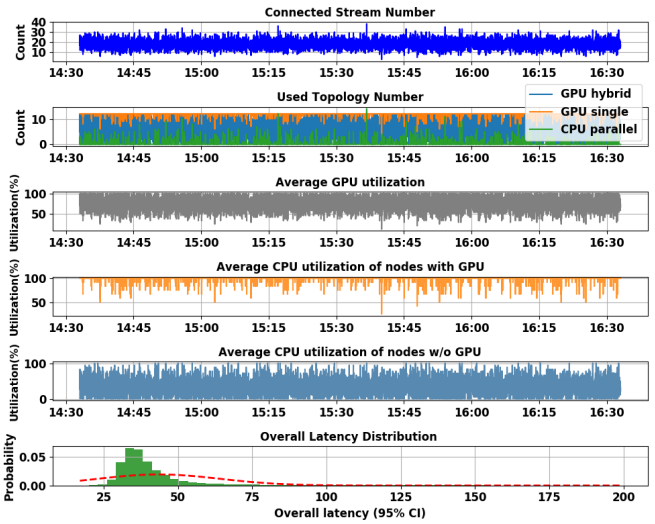


Fig. 12: Important run time parameters for our 3D reconstruction edge cloud in a 2-hour duration.

V. Related Work

In this section, we briefly discuss related work in execution acceleration by edge cloud computing, popular distributed and parallel computing platforms, and relevant task allocation algorithms.

A. Execution Acceleration via Edge Cloud Computing

Many works in the rising Edge Computing field have been proposed to tackle challenges in systems, models, and applications as it introduces a new computing and communication paradigm. Yang et al. [27] propose to dynamically partition data stream between mobile and cloud server to minimize processing latency by a centralized genetic algorithm. Chaufournier et al. [28] use multi-path TCP to accelerate edge cloud service migration to reduce the network latency when mobile users move away. Pang et al. [29] consider a latency-motivated cooperative task computing framework for selection of edge clouds to provision edge services. Bahreini et al. [30] design an online heuristic algorithm that efficiently places application tasks in edge clouds to minimize execution time. Zhang et al. [9] propose an edge-based VR gaming architecture where edge clouds perform heavy frame rendering tasks to reduce end-to-end latency significantly . FemtoCloud [31], P^3-Mobile [32] explore idle mobile devices to configure a compute cluster and provisions cloud services at the edge. Users can leverage this mobile cluster to perform parallel programming to accelerate computation speed. These works focus on optimizing the communication pattern in the systems to achieve lower latency, but they fail to provide a practical execution platform and a resource orchestration mechanism which are aware of resource heterogeneity at the edge.

B. Task Allocation Algorithms

Efficiently assigning tasks of an application to proper processors is critical to achieve high performance in a heterogeneous computing environment [33]. Task allocation, as an NP-complete problem, has been extensively studied and many

heuristics solutions have been proposed according to diverse optimization goals [34]. In terms of Storm platform specific task allocation schemes, several major schedulers have been proposed. T-storm [35] and the work in [36] proposed a traffic-aware task allocation that tries to minimize inter-node and inter-process traffic. R-storm [26] introduced a resource-aware task allocation that intends to increase overall throughput by maximizing resource utilization. Although the above schedulers showed performance improvement over the default Round-Robin mechanism, they failed short in achieving lower latency in the context of edge computing which required proactive available resource estimation and task profiling. The state of art of stragglers mitigation is to introduce speculative execution that waits to observe the progress of the tasks of a job and launches duplicates of those tasks that are slower [15]. This approach, however, is usually applied in cloud computing where computing resource is much more abundant to utilize.

VI. Concluding Remarks and Future Directions

In this paper, we develop a *latency-aware* edge resource orchestration platform based on Apache Storm. The platform aims to support real-time responses to edge applications that are computation intensive. The main contribution of our platform stems from a set of *latency-aware* task scheduling schemes. By deploying the proposed platform on a group of edge servers with heterogeneous CPU, GPU and networking resources, we show that we are able to support real-time edge vision applications, rending the per frame latency around 32ms. Our study shows that edge cloud computing is indeed a promising platform to support emerging edge applications. Moving forward, we will continue to investigate how to further drive down the latency, e.g., distributing the bottleneck bolt across multiple GPUs. We will also investigate how to efficiently integrate our edge cloud computing platform with traditional central clouds to support applications that need to utilize both modes. Another future work topic is that of understanding the impact of access network bandwidth on edge resource assignment and scheduling.

References

[1] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *EuroSys*, pp. 301–314, ACM, 2011.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *MCC*, pp. 13–16, ACM, 2012.

[3] Y.-Y. Shih, W.-H. Chung, A.-C. Pang, T.-C. Chiu, and H.-Y. Wei, "Enabling low-latency applications in fog-radio access networks," *IEEE Network*, vol. 31, no. 1, pp. 52–58, 2017.

[4] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *MobiSys*, pp. 68–81, ACM, 2014.

[5] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, "You can teach elephants to dance: agile vm handoff for edge computing," in *SEC*, p. 12, ACM, 2017.

[6] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[7] T. G. Rodrigues, K. Suto, H. Nishiyama, and N. Kato, "Hybrid method for minimizing service delay in edge cloud computing through vm migration and transmission power control," *TOC*, vol. 66, no. 5, pp. 810–819, 2017.

[8] A. Ceselli, M. Premoli, and S. Secci, "Mobile edge cloud network design optimization," *TON*, vol. 25, no. 3, pp. 1818–1831, 2017.

[9] W. Zhang, J. Chen, Y. Zhang, and D. Raychaudhuri, "Towards efficient edge cloud augmentation for virtual reality mmogs," in *SEC*, p. 8, ACM, 2017.

[10] J. Wang, B. Amos, A. Das, P. Pillai, N. Sadeh, and M. Satyanarayanan, "A scalable and privacy-aware iot service for live video analytics," in *MMSys*, pp. 38–49, ACM, 2017.

[11] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance.," in *NSDI*, vol. 9, p. 1, 2017.

[12] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *SEC*, p. 15, ACM, 2017.

[13] W. Cerroni and F. Callegati, "Live migration of virtual network functions in cloud-based edge networks," in *ICC*, pp. 2963–2968, IEEE, 2014.

[14] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm twitter," in *SIGMOD*, pp. 147–156, ACM, 2014.

[15] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones.," in *NSDI*, vol. 13, pp. 185–198, 2013.

[16] Q. Shan, B. Curless, Y. Furukawa, C. Hernandez, and S. M. Seitz, "Occluding contours for multi-view stereo," in *CVPR*, pp. 4002–4009, 2014.

[17] O. Hilliges, D. Kim, S. Izadi, M. Weiss, and A. Wilson, "Holodesk: direct 3d interactions with a situated see-through display," in *SIGCHI*, pp. 2421–2430, ACM, 2012.

[18] J. Kim and C. Park, "End-to-end ego lane estimation based on sequential transfer learning for self-driving cars," in *CVPR, 2017*, pp. 1194–1202, IEEE, 2017.

[19] N. Bernini, M. Bertozzi, L. Castangia, M. Patander, and M. Sabbatelli, "Real-time obstacle detection using stereo vision for autonomous ground vehicles: A survey," in *ITSC*, pp. 873–878, IEEE, 2014.

[20] H. Liu, F. Eldarrat, H. Alqahtani, A. Reznik, X. de Foy, and Y. Zhang, "Mobile edge cloud system: Architectures, challenges, and approaches," *ISJ*, no. 99, pp. 1–14, 2017.

[21] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[22] "NVIDIA Docker." https://github.com/NVIDIA/nvidia-docker.

[23] "Apache Storm." http://storm.apache.org/.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, pp. 1–10, IEEE, 2010.

[25] "NVIDIA OPENACC." https://developer.nvidia.com/openacc.

[26] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Middleware*, pp. 149–161, ACM, 2015.

[27] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *SIGMETRICS PER*, vol. 40, no. 4, pp. 23–32, 2013.

[28] L. Chaufournier, P. Sharma, F. Le, E. Nahum, P. Shenoy, and D. Towsley, "Fast transparent virtual machine migration in distributed edge clouds," in *SEC*, p. 10, ACM, 2017.

[29] A.-C. Pang, W.-H. Chung, T.-C. Chiu, and J. Zhang, "Latency-driven cooperative task computing in multi-user fog-radio access networks," in *ICDCS*, pp. 615–624, IEEE, 2017.

[30] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *SEC*, p. 5, ACM, 2017.

[31] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," in *CLOUD*, pp. 9–16, IEEE, 2015.

[32] J. Silva, D. Silva, E. R. Marques, L. Lopes, and F. Silva, "P3-mobile: Parallel computing for mobile edge-clouds," in *CrossCloud*, p. 5, ACM, 2017.

[33] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *TPDS*, vol. 13, no. 3, pp. 260–274, 2002.

[34] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *CCR*, vol. 45, pp. 393–406, ACM, 2015.

[35] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *ICDCS*, pp. 535–544, IEEE, 2014.

[36] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *DEBS*, pp. 207–218, ACM, 2013.