HeteroEdge: Taming The Heterogeneity of Edge Computing System in Social Sensing

Daniel (Yue) Zhang, Md Tahmid Rashid
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana, USA
yzhang40@nd.edu,mrashid5@nd.edu

Xukun Li
Department of Statistics
Kansas State University
Manhattan, KS, USA
xukun@ksu.edu

Nathan Vance, Dong Wang
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana, USA
nvance1@nd.edu,dwang5@nd.edu

ABSTRACT

Social sensing has emerged as a new sensing application paradigm where measurements about the physical world are collected from humans or devices on their behalf. The advent of edge computing pushes the frontier of computation, service, and data along the cloud-to-IoT continuum. The merge of these two technical trends (referred to as Social Sensing based Edge Computing or SSEC) generates a set of new research challenges. One critical issue in SSEC is the *heterogeneity* of the edge where the edge devices owned by human sensors often have diversified computational power, runtime environments, network interfaces, and hardware equipment. Such heterogeneity poses significant challenges in the resource management of SSEC systems. Examples include masking the pronounced heterogeneity across diverse platforms, allocating interdependent tasks with complex requirements on devices with different resources, and adapting to the dynamic and diversified context of the edge devices. In this paper, we develop a new resource management framework, HeteroEdge, to address the heterogeneity of SSEC by 1) providing a uniform interface to abstract the device details (hardware, operating system, CPU); and 2) effectively allocating the social sensing tasks to the heterogeneous edge devices. We implemented HeteroEdge on a real-world edge computing testbed that consists of heterogeneous edge devices (Jetson TX2, TK1, Raspberry Pi3 and personal computer). Evaluations based on two real-world social sensing applications show that the HeteroEdge achieved up to 42% decrease in end-to-end delay for the application and 22% more energy savings compared to the state-of-the-art baselines.

1 INTRODUCTION

Social sensing has recently emerged as a new application paradigm where humans and the devices they owned collect sensor measurements from the physical world [43]. Examples of social sensing applications include urban traffic monitoring using mobile apps [22], obtaining real-time situation awareness in the aftermath of a disaster using self-reported observations from citizens [17], and abnormal event detection using portable video devices [23]. One critical limitation of existing social sensing solutions is that the computation and data analytics tasks are primarily performed on the backend servers (e.g., dedicated server or cloud platform), which often causes excessive bandwidth consumption and unnecessary delay [29, 48]. Edge computing has become a new computing paradigm that pushes the frontier of computation, data, and services to the edge of the network where social sensing happens [34].

In this paper, we focus on a Social Sensing based Edge Computing (SSEC) paradigm where the sensing and devices owned by individuals are used as the computational resource at the edge to accomplish the tasks from social sensing applications. The SSEC paradigm has several unique advantages. First, SSEC provides a more economical and scalable edge computing solution by leveraging the SSEC devices that are owned by the end users. Second, the SSEC devices form a mobile sensor network to accomplish the sensing tasks that are challenging for infrastructure/static sensors. Third, SSEC is suitable for delay-sensitive applications by pushing the computation and service to the edge where the users reside [37]. Finally, SSEC reduces the risk of overloading the back-end servers and avoids the single point of failure in the system [45].

Despite its immense benefits, SSEC also introduces a set of research challenges such as the non-cooperativeness of end users [46], privacy and security concerns [42] and incentive design [21]. In this paper, we focus on a critical challenge in SSEC that has yet to be well addressed: *heterogeneity*. In particular, the edge devices in SSEC often have diversified computational power, runtime environments, and hardware equipment, making it hard to orchestrate these devices to collaboratively accomplish the tasks in a social sensing application. Previous efforts were made to accommodate heterogeneous devices in a computing cluster. Examples include HTCondor [19] and FemtoCloud [9]. However these solutions cannot address the heterogeneity problem in SSEC due to several unique technical challenges elaborated below.

Pronounced Heterogeneity: the heterogeneity in SSEC is more pronounced than regular distributed/cloud based systems because i) it is not possible for the application to cherry-pick the devices in a fully controlled manner given the fact the devices are owned by individuals [6, 33]; ii) the degree of heterogeneity of SSEC devices is much more significant than the distributed or cloud computing systems that assume homogeneous tasks [9] or homogeneous architecture [19]. With pronounced heterogeneity, existing resource

IoTDI'19, April 2019, Montreal, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

management techniques must be re-designed to cater for heterogeneous SSEC. In particular, traditional backend-based social sensing applications are often designed for specific hardware or operating system environments and may not be directly applicable in a heterogeneous system. For instance, the application written for X86 is not executable on ARM processors due to architectural and hardware difference. Similarly, the applications developed for the Windows OS cannot be executed on the Android OS without code modification and re-configuration. The privilege of "write once run anywhere" is often divested from SSEC developers [5]. Therefore, the pronounced heterogeneity requires the resource management in SSEC to support diverse social sensing tasks to maximize resource utilization while relieving the burden of the developers.

Complex Task-Resource Mapping: the second challenge refers to the complexity of efficiently allocating interdependent social sensing tasks with diversified resource requirements to heterogeneous devices with complex delay-energy tradeoff in SSEC. First, the social sensing tasks are often complicated and require heterogeneous hardware resources (e.g., some tasks require sensors, some tasks require GPU) [36]. Second, the edge devices often have diversified configurations of hardware components (e.g., GPU, single core CPU, multi-core CPU, sensors). Third, it is difficult to orchestrate the heterogeneous edge devices in accomplishing collaborative social sensing tasks with non-trivial task dependencies. Finally, various task allocation strategies may yield complex tradeoffs between energy cost and delay overhead (e.g., assigning a task to a GPU may incur less delay but higher energy cost as compared to assigning the task to a CPU.) Given the above unique complexities, we found existing resource management schemes that leverage heterogeneous devices (e.g., HTCondor [19], CoGTA [47], and FemtoCloud [9]) cannot solve the complex task mapping problem in SSEC.

Dynamic Context: the third challenge refers to the fact that the edge devices may have different and dynamic contextual environments. The contextual environment refers to the detailed status of edge devices (e.g., the location of the device, the CPU/memory utilization, and the battery status), which often change over time as the system runs. It is important to keep track of the contextual information in SSEC to optimize many run-time decisions (e.g., task allocation, incentive adjustment) [4]. Existing edge computing systems often assume that a central controller in the system has a full knowledge about the context information of all edge devices [7, 34, 35], which is not practical in SSEC for two reasons. First, the end users have ultimate control of their edge devices and they will decide what type of context should be visible to the application. For example, a user who chooses to share the GPU resource in a SSEC application might change her mind if the battery of the device becomes low. Second, it also introduces a significant synchronization overhead by tracking the exact CPU usage and other context environments in real-time (e.g., the edge devices have to constantly update its status to the server) [46]. Therefore, there is a lack of a approach that allows end users to self-configure and provide the useful dynamic context to the application.

To address the above challenges, we propose a new resource management framework called *HeteroEdge* to tame the heterogeneity of SSEC. In particular, we develop a novel supply chain based task mapping model that allows heterogeneous edge devices to collaboratively finish complicated and interdependent social sensing tasks with an optimized delay-energy tradeoff. HeteroEdge addresses the pronounced heterogeneity of SSEC devices and dynamic context challenges by developing hardware abstraction and runtime modules. We implemented a system prototype of HeteroEdge on a real-world SSEC testbed that consists of RaspberryPi3, Nvidia Jetson TX2, Jetson TK1 boards, and personal computers. The HeteroEdge was evaluated using two real-world social sensing applications: *Disaster Damage Assessment* and *Collaborative Traffic Monitoring*. We compared HeteroEdge with the state-of-the-art resource management schemes used in edge computing systems. The results show that our scheme achieves a significant performance gain in terms of delay and energy consumption: our scheme achieved up to 42% decrease in the end-to-end delay for the application and 22% more energy savings for edge devices compared to the baselines.

2 RELATED WORK

2.1 Social Sensing and Edge Computing

Social sensing has received a significant amount of attention due to the proliferation of low-cost mobile sensors and the ubiquitous Internet connectivity [43]. A large set of social sensing applications are sensitive to delay, i.e., have real-time requirements. Examples of such applications include intelligent transportation systems [22], environmental sensing [24], and disaster and emergency response [17]. Traditional social sensing applications push all the computation tasks to the remote servers/cloud, which can be quite ineffective, particularly for delay-sensitive applications, when the network bandwidth is limited and the communication latency is high [48]. Edge computing systems complement traditional centralized social sensing solutions by offloading computation tasks to the edge devices to significantly reduce communication costs and application latency [14]. A comprehensive survey of edge computing is given by Shi *et al.* [37].

Social Sensing base Edge Computing (SSEC) is enabled by a few key technical trends: i) the IoT devices owned by individuals are becoming increasingly powerful and some of them even have similar computing power as the dedicated servers in traditional edge computing systems [46, 47]. Therefore, it becomes a growing trend to push the computation directly to the edge devices rather than dedicated remote servers or cloudlets [9]; ii) The popularity of mobile payment provides a more convenient way for common individuals to receive incentives by contributing the spare resources on their IoT devices for accomplishing the social sensing tasks [47]. In this paper, we focus on the heterogeneity challenge in SSEC systems.

2.2 Resource Management in Edge Computing

Resource management is a fundamental problem in edge computing systems and many solutions have been developed to address this problem [7, 34, 40]. For example, Zhu *et al.* proposed a Mixed Integer Linear Programming based approach to meet the deadlines and minimize the end-to-end latency in hard real-time systems [50]. Su *et al.* developed a mixed criticality task allocation model to maximize the number of low-criticality tasks being executed without influencing the timeliness of high-criticality tasks [40]. Most of the above schemes adopt a centralized approach that employs a central decision maker to allocate tasks in the system. Such an

approach fails in the edge computing systems where the devices might refrain from providing necessary information to accomplish the centralized task allocation [46]. Decentralized task allocation schemes combined with incentive mechanisms have been developed to address the above limitation. For example, Ahmad *et al.* proposed a game theoretic approach for scheduling tasks on multi-core processors to jointly optimize performance and energy [2]. Zhang *et al.* developed a game-theory based approach to assign computation tasks to *non-cooperative* edge devices with dynamic incentives by considering the conflicting objectives between edge devices and applications [46]. Liu *et al.* proposed a decentralized data offloading scheme using the multi-item auction and congestion game approach to allow edge devices to decide the optimal strategy for offloading tasks to the cloud [20]. However, these schemes assume the computing nodes have homogeneous task execution interface and largely ignore the heterogeneity of edge devices.

2.3 Distributed System with Heterogeneous Computing Nodes

Taming heterogeneity of heterogeneous computing nodes has been identified as a critical undertaking in distributed systems. Various solutions have been developed in the past that target at either resource heterogeneity or network heterogeneity. For example, the HTCondor system can harness the idle computational cycles from distributed workstations to accomplish computation tasks [18]. Habak *et al.* proposed FemtoCloud, which is a dynamic and self-configuring system architecture that enables privately owned mobile devices to be configured into a coordinated computing cluster[9]. More recently, Zhang *et al.* proposed CoGTA, an edge computing system that allows non-cooperative and heterogeneous edge devices to trade tasks and claim rewards [47]. However, the above schemes suffer from two major limitations: i) they made strong assumptions that the tasks only require homogeneous computational resources (i.e., CPU and memory) which is not true in SSEC where complicated social sensing tasks may require a diversified set of resources such as sensors, CPU and GPU; ii) they assume the edge devices are always compatible with the computation tasks which is again not necessarily true in SSEC where the edge devices may have different runtime environments such as OS and software dependencies. In contrast, our HeteroEdge framework explicitly models the heterogeneous edge devices and manages the diversified resources in the system using a novel economics-based model.

2.4 IoT Middleware

The proposed work is related to the some existing literature on IoT middleware. The IoT middleware targets at enabling connectivity for heterogeneous IoT devices, making communication possible among devices that would not otherwise be capable. Typical IoT middleware solutions include TerraSwarm [16], Xively [38], and Global Sensor Networks [1]. There exist an important knowledge gap in the above IoT middleware solutions by largely ignoring the potential of performing non-trivial computation tasks on increasingly powerful and ubiquitous edge devices owned by individuals. HeteroEdge focuses on providing reliable *computation power* over heterogeneous edge devices in IoT systems to accomplish computationally intensive tasks (e.g., deep learning based inference, image processing) that are traditionally done in the backend/cloud. Such "computation-centric" focus is different from the focus of providing *interconnectivity* and *interoperability* from the traditional *sensing-centric* IoT middleware solutions. Our work is also significantly different from recent computing-centric IoT middleware solutions. Examples of such solutions include FemtoCloud [9], Deviceless [8], CoGTA [47], and AWS Greengrass IoT [15]. However, the current computing-centric solutions have several key limitations: 1) they largely ignore the fact many computationally-capable IoT devices are now owned by individuals and there exist a significant runtime heterogeneity in those devices (different OS, software libraries); 2) they do not fully consider the heterogeneity of the computation tasks (e.g., tasks that require CPU, GPU or sensing modules) together with non-trivial task dependencies; 3) they do not fully explore the diversified configurations of hardware components (e.g., GPU, single core CPU, multi-core CPU, sensors) of the edge devices). In fact, the solutions mentioned above primarily focus on tasks that require CPU. In contrast, our solution makes a unique contribution by jointly addressing the above challenges.

3 PROBLEM FORMULATION

In this section, we formally define the resource management problem in SSEC with heterogeneous edge devices. Figure 1 shows a high-level overview of SSEC system. The SSEC system incorporates a set of edge devices, $ED = \{E_1, E_2 ... E_X\}$, and a local edge server ES. These edge devices can perform sensing tasks (e.g., collecting image data using camera sensor), computation tasks (e.g., running image classification algorithms) or perform data transmission over the wireless network interface. The edge devices are assumed to be heterogeneous by having different system architecture (e.g., X86, ARM), operating systems (e.g., Windows, Linux), hardware (e.g., with/without GPU, sensors), and networking interfaces (e.g., WiFi, Bluetooth). The local edge server serves as a general networking hub with various network interfaces and all the edge devices can communicate with the local edge server.

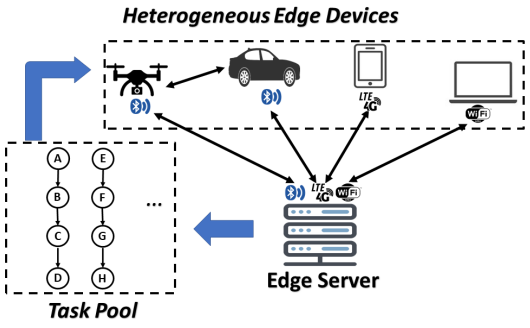


Figure 1: SSEC with Heterogeneous Devices

We first discuss the task model in our framework. A social sensing application is assumed to have a set of Z jobs, $Job = \{J_1, J_2, ... J_Z\}$, which are initialized by the server at the beginning of each sensing cycle (i.e., sampling period). Each job converts the raw sensor input data to the final analysis results. We adopt a frame-based task model [47] commonly used in the real-time

system community where jobs are periodically initialized and have the same period and deadline. We use Δ to denote the common deadline of all the jobs in an application. Δ captures the user desired QoS in terms of when the jobs should be finished.

To accomplish the data processing function, each job consists of M pipeline tasks that have task dependencies. Each task is associated with a 4-tuple: $\tau_i = \{VI_i, VO_i, WCET_{i,x}, R_i\}$ where VI_i is the data volume to be processed by task τ_i and VO_i is the size of the output. $WCET_{i,x}$ is the estimated worst-case execution time (WCET) if τ_i is assigned to edge device $E_x, 1 \leq x \leq X$. We explain the details of WCET estimation in the next subsection. R_i is the reward of completing τ_i to motivate the edge devices to participate the HeteroEdge framework. The dependencies among the tasks in an application are modeled by a *task dependency graph* defined below.

DEFINITION 1. **Task Dependency Graph ($\mathbf{G}_{task}$)**: a directed graph $\mathbf{G}_{task} = (\mathbf{V}_{task}, \mathbf{L}_{task})$ where vertex $V_i \in \mathbf{V}_{task}$ represents task τ_i; link $(\tau_i \rightarrow \tau_j) \in \mathbf{L}_{task}$ signifies that the input of task τ_j depends on the output of task τ_i (Figure 1).

For a given application, we assume that a total of N tasks (from all Z jobs) are to be processed in each sensing cycle, i.e., $\{\tau_1, \tau_2, ..., \tau_N\}$.

To illustrate the task model of SSEC, we show an example application called Disaster Damage Assessment (DDA) of SSEC where a set of edge devices are tasked to provide reports of the severity of damages during a natural disaster (Figure 2). In this application, a job is defined as the inference of the damage severity of a specific location of interest (i.e., location A or B in the figure). Each job can then be broken down into a task pipeline, including i) collecting the raw image data (via camera sensors) of the scene, ii) pre-processing the images and iii) inferring the severity of the damage from the images. Due to the heterogeneous nature of SSEC, a single device may not be capable of processing all tasks in a social sensing job. In the above example, a smartphone device that picks Job 2 collects the raw image but cannot efficiently process the image for the final results due to insufficient GPU power. Therefore, it offloads the image to a nearby device (a laptop) for further processing. Under such a scenario, the laptop and the smartphone complement each other and collectively finish a social sensing task that cannot be accomplished by either of them alone (the laptop has no image sensors and the smartphone does not have enough computing power).

We model the communication channels in the edge as a *Communication Graph*:

DEFINITION 2. **Communication Graph $\mathbf{G}_{com}$:** an undirected graph $\mathbf{G}_{com} = (\mathbf{V}_{com}, \mathbf{L}_{com})$. $\mathbf{V}_{com}$ is the set of all edge devices and the edge server ES. $\mathbf{L}_{com}$ defines the communication channels where $(E_x, E_y) \in \mathbf{L}_{com}$ denotes E_x and E_y can directly communicate with each other. We also have $(E_x, ES) \in \mathbf{L}_{com}, \forall 1 \leq x \leq X$

Given a set of tasks from social sensing application, and a set of heterogeneous edge devices from the end users, the design goal of HeteroEdge is to orchestrate the edge devices in the SSEC system to perform social sensing and computation tasks in an optimized way that minimizes the End-to-End (E2E) delay of the application and maximizes energy savings of the edge devices. We formally define E2E delay below:

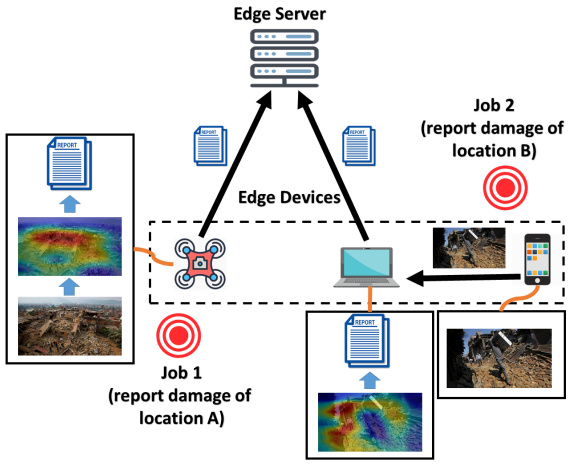


Figure 2: A Social Sensing Edge Computing Application Example: Disaster Damage Assessment

DEFINITION 3. **End-to-end delay of a job ($\mathcal{D}_z$)**: the total amount of time taken for a unit of sensor measurement data (e.g., a video frame) to be processed by all tasks in J_z. It includes the total computation time of J_z and the total communication overhead.

The above objective can be formulated as a multi-objective constrained optimization problem:

$$\begin{aligned} &\text{minimize:} && \sum_{x=1}^{X} e_x \text{ (energy minimization objective)} \\ &\text{minimize:} && \mathcal{D}_z, \forall 1 \leq z \leq Z \text{ (application's QoS objective)} \\ &\text{s.t.:} && \mathbf{G}_{task}, \mathbf{G}_{com} \text{ are satisfied} \\ & && \text{(task and communication constraints)} \end{aligned} \quad (1)$$

where e_x is the energy consumption of edge device E_x in a sensing cycle.

Finally, we summarize a few additional assumptions we made in our model: i) we assume edge devices are not malicious (e.g., give fake outputs) or lazy (i.e., intentionally postpone task executions) [46]; ii) we assume edge devices do not quit or join the system within a sensing cycle; iii) we assume end users are willing to provide their computation resource and energy of their devices by receiving incentives [21]. We discuss how to deal with situations where such assumptions are not satisfied in Section 6.

4 THE HETEROEDGE FRAMEWORK

This section presents the system design and technical details of the HeteroEdge framework. An overview of HeteroEdge is given in Figure 3. It consists of three main modules: i) a runtime abstraction module, ii) a hardware abstraction module, and iii) a task mapping module. The runtime abstraction module and hardware abstraction module abstract away the heterogeneous details of edge devices and provide a uniform resource pool for the social sensing applications. The task mapping module allocates interdependent social sensing tasks to heterogeneous hardware resources in a way that optimizes the delay-energy tradeoff for the application. We discuss the details of these components below.

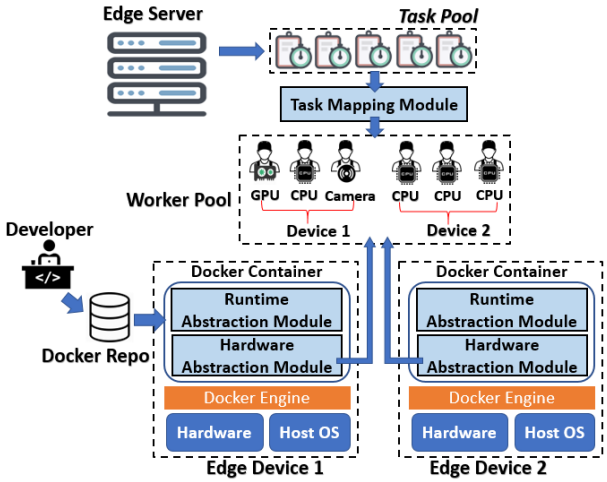


Figure 3: HeteroEdge Architecture Overview

4.1 Runtime Abstraction Module

A critical issue in heterogeneous SSEC is that the devices have different hardware and runtime environments that may not support the social sensing tasks to be processed. For example, a device may have incompatible operating system or lack the necessary dependencies to execute a social sensing algorithm (e.g., a deep learning algorithm cannot run on a device without necessary library such as Tensorflow or CUDA). To address this issue, we leverage the Docker containerization technique[26] which is a computer program that performs operating-system-level virtualization. It abstracts away the hardware details of the devices and provides an virtual environment that offers a lightweight, portable and high performance sandbox to host various applications [**?**]. In particular, the social sensing application developers can "wrap" all necessary dependencies and the OS itself into a Docker container for each social sensing application and then upload the container image to a Docker repository. Any edge devices that have Docker engine installed can pull the image from the repository and run the social sensing application. Since the Docker container is self-contained, neither the application developer nor the device owners need to worry about its own hardware, OS or runtime environment. The task execution module in HeteroEdge therefore allows the edge devices in SSEC to provide the same interface to the social sensing application developers and offers them the "write once and run anyway" feature despite of the heterogeneity of SSEC devices.

4.2 Hardware Abstraction Module

HeteroEdge further performs hardware-level abstraction of the computing resources available on the edge device by developing a hardware abstraction module. The hardware abstraction module abstracts away the details of heterogeneous hardware specifications from the edge devices. We are inspired by the idea from the Work Queue framework [49] where the hardware capability of a device can be represented as a set of workers. In particular, we consider three types of workers that are essential in finishing social sensing tasks in SSEC - CPU, GPU, and Sensor workers. Each worker is associated with a capability descriptor in terms of the estimated WCET of processing social sensing tasks and a visibility flag. We formally define the workers as follows.

DEFINITION 4. **CPU Worker:** A CPU worker represents an idle computation thread (we assume one thread per core for simplicity) of an edge device. The number of workers reflects the capability of a device to handle multiple social sensing tasks simultaneously.

DEFINITION 5. **GPU Worker:** A GPU worker represents an idle GPU of an edge device.

DEFINITION 6. **Sensor Worker:** A sensor worker represents an available sensor on an edge device. The sensor worker has various types e.g., GPS/ Video/ Camera/ Accelerometer.

The workers of an edge device jointly define the context of the device at any given time. We assume the edge devices have constantly changing sets of workers as the system runs or when the users change their system configurations. An example worker pool of a device E_1 at a sensing cycle is {1, CPU, visible, *Alg1*: 500 ms, *Alg2*: 1500 ms}, {1, GPU, invisible, Alg1: 500 ms, Alg2: 100 ms}, {1, Sensor-Camera, visible, *Sens*: 10 ms}, {1, Sensor-GPS, visible, NA}, where *Sens*, *Alg1* and *Alg2* are the task pipelines of a social sensing job. The *visible* and *invisible* are the flags set by users to denote their willingness to disclose the worker to the application.

The benefits of hardware abstraction module are three-fold: i) the set of heterogeneous edge devices form a unified homogeneous worker pool for the social sensing application by mapping the devices to workers; ii) the end users can register and control the workers they would like to provide for a particular social sensing application in a way that preserves their privacy; iii) the edge device can easily keep track of their own dynamic status and provide necessary context information for the run-time decision and optimization in SSEC.

4.3 A Supply Chain-based Task Mapping Module

The above runtime and hardware abstraction modules are designed to provide a "homogeneous" resource pool and execution interface to the social sensing application. However, performing task mapping in SSEC is still challenging because 1) tasks are heterogeneous and have complex execution requirements (e.g., sensing tasks can only be done on devices with compatible sensors and computational tasks may require specific computational resources such as a GPU); 2) the computing resources in our model are also heterogeneous (e.g., some devices have sensors while others do not; some devices are equipped with GPU while others are not); 3) various task allocation strategies may yield complex tradeoffs between energy cost and delay overhead.

To this end, we develop our own supply chain based task mapping model to address the above challenges. In order to adapt the supply chain model to solve the task mapping problem, we develop several novel technical components. In particular, we proposed a novel *supply chain graph mapping* technique and a *node decomposition* component to jointly model the heterogeneous tasks, computing resources, and the trade-off between energy and delay using a directed supply chain graph. The combination of the two techniques reduced the complex problem of finding the optimal task

mapping strategy that optimizes the delay-energy tradeoff to finding the shortest path in the supply chain graph. We also designed a new game-theoretic selfish routing algorithm to find the optimal task mapping strategy with bounded performance guarantee. We elaborate these components in details below.

4.3.1 Supply Chain Graph Mapping. Our solution is motivated by the observation of an interesting mapping between our problem and the *supply chain* model in economics. The supply chain problem involves the transformation of natural resources, raw materials, and components into a finished product that is delivered to the end customer. To become the end product, the raw material has to be transported and processed at different factories/facilities with different capabilities (e.g., sourcing, manufacturing, packaging, assembly). In HeteroEdge, we consider the raw sensing measurements as "raw material" and the sensing devices the "suppliers" of the raw material. The raw material has to be processed through a set of factories (i.e., edge devices) to become the final product (i.e., the end results). We refer to the series of factories/devices that the raw material travels through till reaching the consumer as a *supply chain path*. The factories have to work collaboratively by sending the processed material to one another for further processing. The edge server can be considered as the "consumer" of the final product. In particular, the chain of raw sensing data → computation nodes → edge server is an exact mapping of raw material → factories → consumer in the supply chain model. We illustrate such a mapping in Figure 4.

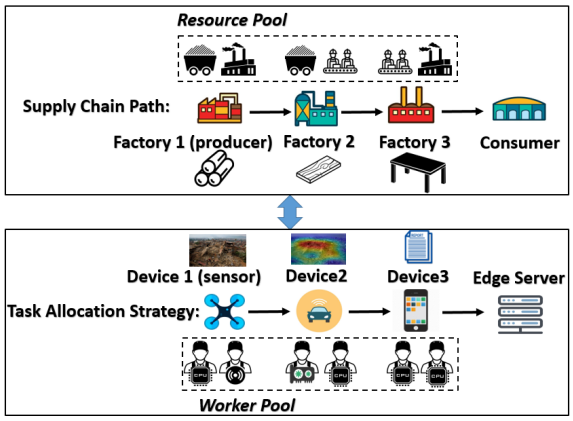


Figure 4: Task Mapping - Supply Chain

Formally, we can map the task mapping problem into a *supply chain graph* $\mathbf{G}_{sc} = (\mathbf{V}_{sc}, \mathbf{L}_{sc})$. The supply chain graph consists of a set of "device nodes" that represent the heterogeneous edge devices. Each device node is associated with the computation delay and energy cost for processing the tasks. Besides the device nodes, we also add some "source nodes" and a "destination node". The source nodes represent the locations that "supply" the raw sensing data decided by the social sensing application. The destination node represents the edge server who receives the end results (the consumer of the supply chain). We also define a set of links to represent the communication channels between edge devices. A link $l \in \mathbf{L}_{sc}$ is associated with a transmission delay and energy cost.

An example of a supply chain graph is illustrated in Figure 5. It involves a social sensing job of three tasks (one sensing task, two computation tasks) and three edge devices. The device capability table shows the tasks the edge devices can execute. To model the task dependency, we divide the supply chain into multiple stages. At each stage, we list all the devices that can execute the corresponding task. For example, stage 1 represents the "sensing task" to collect raw data from two locations. All devices are listed because device A is able to collect data from location 1 and devices B and C are capable of collecting data from location 2. In the next stage, devices B and C can perform the computation task (A1). In the final stage, device C can perform the final task (A2). Note that the edge server *ES* is added to all stages of computation tasks because the edge devices can always choose to offload the computation tasks to the edge server. We use dashed lines to represent "no cost" link (e.g., communication on the same device) and use solid lines to represent a communication associated with delay and energy cost.

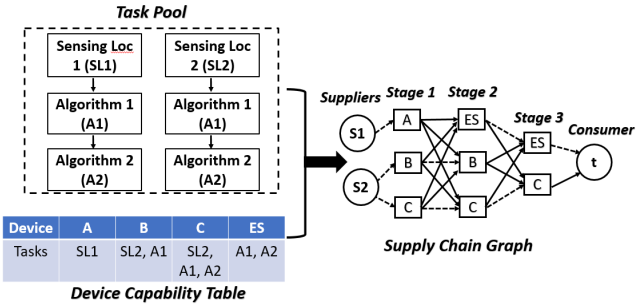


Figure 5: Supply Chain Graph Setup

Given the supply chain graph, our goal is to find the best route (i.e., supply chain path) from each source to the destination that minimizes overall delay and energy consumption. Let P_z denote a supply chain path from a source node s_z to the destination node t. π_z is the total cost of P_z (including the delay and energy cost during data transmission and processing on the device nodes of P_z). The goal is to find:

$$\underset{P_x}{\operatorname{argmin}}\ \pi_z, \forall 1 \le z \le Z \tag{2}$$

To solve the objective function above, we perform i) a node decomposition that unifies the *computation* cost and the *data transmission* cost of the links. This step translates the supply chain problem into a multi-source shortest-path problem; 2) a selfish routing algorithm that allows jobs to selfishly pick their paths to solve the multi-source shortest-path problem.

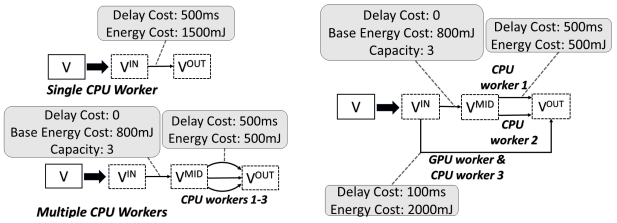
4.3.2 Node Decomposition. For the supply chain graph problem shown in Figure 5, the goal of the task mapping is to find the optimal path (with minimal delay and energy cost) for suppliers s_1 and s_2. To solve this problem, we first transform the supply chain graph into a uniform graph by associating all the cost with the links and expanding the device nodes to model the heterogeneous workers of the devices. The transformation considers the following scenarios.

Device with a single CPU worker: we transform a device node into two virtual nodes: v^{IN} denotes the "entry" of a factory

(edge device) and v^{OUT} denotes the "exit". We create a "virtual link" between v^{IN} and v^{OUT} and the link is associated with the delay and energy cost of performing a task on the CPU worker. The node decomposition of this scenario is illustrated in Figure 6(a).

Device with multiple CPU workers: Multiple CPU workers represent the multi-threading capability of an edge device, which adds more complexities in modeling the energy cost. We use a linear energy model where $power\ consumption = base\ energy + extra\ energy\ consumption \times number\ of\ threads$ [3] where the *base energy* represents the default energy consumption of a CPU independent of the number of cores being used. To model this, we introduce an extra intermediate virtual node v^{MID} in addition to v^{IN} and v^{OUT}. The link from v^{IN} to v^{MID} is created to model the base energy consumption (with no delay). This link also has a *capacity* l^{cap} that is equal to the number of cores. The link capacity denotes the number of supply chain paths that can go through the link simultaneously without causing any extra base energy cost. For example, a three-core device has the capacity of 3 where three tasks can be run on the device at the same time with only 1 unit of base energy consumption plus three extra units per worker energy consumption. We also created virtual links from v^{MID} to v^{OUT} and the number of the virtual links is the same as the number of workers of the edge device v. Multiple virtual links mean the device can handle multiple tasks at the same time. The node decomposition of this scenario is illustrated in Figure 6(a).

Device with a GPU worker: the node decomposition for a device with GPU and 3 CPU cores is illustrated in Figure 6(b). Note that in many scenarios, GPU requires at least one extra CPU core to run programs [12]. Therefore, we dedicate one CPU worker to the device with the GPU worker while the rest of the CPU workers can process other tasks.



(a) Without GPU (b) With GPU

Figure 6: Node Decomposition Scenarios

After the above node decomposition, our problem becomes a multi-source shortest path problem where the goal is to find the best supply chain path from the source to destination that minimizes the cost of the *links* on the path. In particular, a supply chain path P_z consists of a set of links where each link $l \in P_z$ is associated with two types of cost: delay and energy consumption. For simplicity, we use π_l^{delay} to denote the delay cost of a link l, and π_l^{energy} to denote the energy cost of l. Then we have the objective:

$$\text{minimize:} \quad \sum^{l \in P_z} \pi_l^{energy} + \lambda \times \pi_l^{delay}, \forall 1 \le z \le Z \tag{3}$$

where λ is a scalar to tune the importance of energy consumption of edge devices versus the overall delay of the application.

One issue with the above objective is that the minimization of energy cost depends heavily on the energy profile of the edge devices and tends to be unfair to low-power devices. For example, consider a scenario where the edge is composed of low-power mobile device (e.g., 5W) and a high-power laptop (e.g., 300W), the above objective function will try to push as many computation tasks as possible to the mobile devices to save energy on the laptop, creating an undesirable situation for mobile phone users. To address this issue, we normalize the energy consumption as follows:

$$normalized(e_x) = \frac{e_x}{power_{max} \times \Delta}, 1 \le x \le X \tag{4}$$

where e_x is the energy consumption of a device E_x in a sensing cycle with the length of Δ and $power_{x,max}$ denotes the maximum power consumption of the device.

4.3.3 A Selfish Routing Algorithm for Optimal Supply Chain. The objective in Equation (3) is a non-trivial problem. Intuitively, each supplier (job) can selfishly pick a path that minimizes its own cost. However, the path can be congested if both suppliers pick the same route, which would introduce extra delay and energy cost. We develop a new Supply Chain Selfish Routing (SCSR) scheme to solve this problem. The SCSR scheme is based on a game-theoretic framework that allows each job to selfishly pick the route to maximize its own utility while taking into account the other players' strategies. The benefits of the SCSR scheme are threefold: 1) it is simple and effective; 2) it provides the theoretical guarantee on the convergence and execution overhead, which is crucial for delay sensitive applications; 3) it can nicely coordinate a large number of tasks to simultaneously identify the optimal devices for execution. We first define a few terms in SCSR.

Let $\mathcal{P} = P_1, P_2, ...P_Z$ denote the supply chain paths of all jobs and P_z is the task mapping strategy (i.e., supply chain path) for job J_z. We use P_{-z} to denote the strategies picked by all jobs other than J_z. For the job J_z, we define a weight w_z to represent its workload which is assumed to be proportional to the size of the raw sensing data. We also define $d(l), l \in \mathbf{L}_{sc}$ as all the jobs that pick link l in their strategies. From $d(l)$, we define the *weighted congestion rate* $\mathcal{N}(l)$ of l as the sum of weights of all paths in d(l), i.e., $\mathcal{N}(l) = \sum^{z \in d(l)} (w_z - (l^{cap} - 1))$. The *utility* of a strategy P_z can then be calculated as:

$$u_z(P_z) = \sum^{l \in P_z} \sum^{z \in d(l)} (w_z - l^{cap} + 1) \times (\pi_l^{energy} + \lambda \times \pi_l^{delay}) \tag{5}$$

Based on the utility function, we say a job is *satisfied* with its path if it cannot further decrease cost by ϵ by unilaterally changing its path from P_z, i.e., $u_z(P_z) \le u_z(P_z') + \epsilon$. If every job is *satisfied*, we say a ϵ-Nash Equilibrium is reached. When $\epsilon = 0$, the equilibrium is referred to as Pure Nash Equilibrium (PNE). The Nash Equilibrium can be found using a greedy algorithm based on the Best Response Dynamics [25]. We summarize the algorithm in Algorithm 1.

4.3.4 Algorithm Analysis. The SCSR is an iterative algorithm and the quick convergence is critical for delay-sensitive social sensing applications. In this subsection, we derive the upper bound of the iterations till convergence and prove SCSR converges to PNE in polynomial time. We first map SCSR into an *atomic network congestion game* where each link has the same cost. This can be achieved

Algorithm 1 SCSR Algorithm

Input: Supply Chain Graph $\mathbf{G}_{sc}$, ϵ
Output: Supply chain for all jobs, i.e., $P_1, P_2..., P_Z$

1: **function** SCSR($\mathbf{G}_{sc}$)
2: Perform node decomposition, get transformed graph $\mathbf{G'}_{sc}$
3: **Initialize:** $convergence = False$, $\mathcal{P} = NewArray[Z]$, $\mathcal{P}' = NewArray[Z]$
4: **for all** $z \in [1, Z]$ **do**
5: Randomly set initial strategy P_z for J_z
6: $\mathcal{P}[z] = P[z], \mathcal{P}'[z] = P[z]$
7: **end for**
8: **while** $converge == False$ **do**
9: **for all** $z \in [1, Z]$ **do**
10: run $P'_z = ShortestPath(s_z, t, \mathbf{G'}_{sc})$ for J_z
11: **if** $u_z(P'_z) - u_z(p_z) > \epsilon$ **then**
12: $\mathcal{P}'[z] = P'_z$
13: **end if**
14: **end for**
15: **if** $\mathcal{P}' == \mathcal{P}$ **then**
16: Return $\mathcal{P}$
17: **end if**
18: $\mathcal{P} = \mathcal{P}'$
19: **end while**
20: **end function**

by breaking a link $l \in \mathbf{L}_{sc}$ into multiple sub-links where each sub-link $l' \in l$ has a unit cost. For example, assuming the original link cost has a maximum normalized cost of K, and unit cost of 1. Then the cost can be normalized as K+1 integer values, i.e. [0, 1, 2, ...K], the link can be broken into at most K sub-links.

It is known that in the atomic network congestion game, a *potential function* exists according to [30]:

$$\Phi(\mathcal{P}) = \sum_{l' \in \mathbf{L}_{sc}}^{} \mathcal{N}(l') + \sum_{z=1}^{Z} w_z \times \sum_{l' \in P_z}^{} w_z \tag{6}$$

and the potential function has the following property:

$$\Phi(P_z, P_{-z}) - \Phi(P'_z, P_{-z}) = 2 \times w_z \times (u_z(P_z) - u_z(P'_z)) \tag{7}$$

In game theory, the potential function decreases each time a job makes an improvement step, namely switch to another strategy to improve its utilization (i.e., line 10-12 in Algorithm 1). The above property shows that every time a job makes an improvement step of ϵ by changing from P_z to P'_z, the potential function decreases by $2 \times w_z \times \epsilon$. We prove the convergence and upper-bound of SCSR algorithm as follows.

Theorem 4.1. *The SCSR algorithm converges to ϵ-Nash Equilibrium in polynomial time and bounded by $O(\frac{M \times K \times n^{2C}}{\epsilon})$, where C is a constant.*

Proof. Note that in Equation (6), we have

$$\Phi(\mathcal{P}) \leq M \times K \times w_{max}^2 \tag{8}$$

where w_{max} is the maximum of $w_z, 1 \leq z \leq Z$. Suppose that $w_z^{max}/w_z^{min} = O(n^C)$, where w_{min} is the minimum of $w_z, 1 \leq z \leq Z$. We have the potential function $\Phi(\mathcal{P})$ takes at most $O(\frac{M \times K \times n^{2C}}{\epsilon})$ steps to become zero. Hence the SCSR algorithm requires at most $O(\frac{M \times K \times n^{2C}}{\epsilon})$ steps to converge to Nash Equilibrium.

The above proof shows the efficiency of the SCSR algorithm. We provide more detailed analysis of the convergence and scalability of SCSR in Section 5.6.

5 EVALUATION

In this section, we present an extensive evaluation of HeteroEdge on a real-world edge computing test platform. We present the evaluation results through two real-world social sensing case studies: *Disaster Damage Assessment* and *Collaborative Traffic Monitoring*. The results show that HeteroEdge achieves significant performance gains in terms of QoS and energy efficiency compared to the state-of-the-art baselines.

5.1 Evaluation Platform

We implement the HeteroEdge framework on a real-world SSEC platform that consists of a set of 10 edge devices and 1 local edge server. In particular, we use a PC workstation with Intel E5-2600 V4 processor and 16GB of DDR4 memory as the local edge server. The edge consists of 10 heterogeneous devices: 2 Jetson TX2 [28] and 2 Jetson TK1 [27] boards from Nvidia (commonly used in portable computers, UAVs, and autonomous vehicles), and 5 Raspberry Pi3 Model B boards [31], and 1 personal computer. Figure 7 shows the implemented hardware platform for the edge devices. These edge devices represent different system architectures, operating systems and hardware capabilities. We summarize their specifications in Table 1. All devices and the edge server are connected via a local wireless router. The HeteroEdge system was implemented using Python. We leverage TCP socket programming for reliable data communication among edge devices.

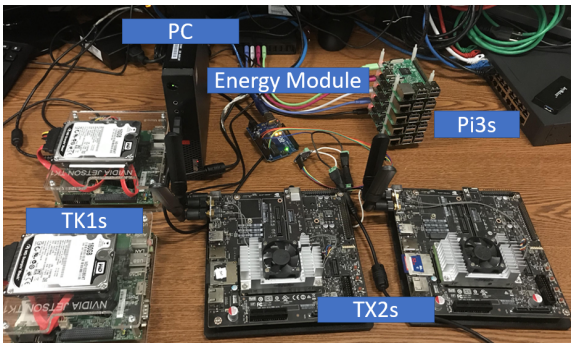


Figure 7: Heterogeneous Edge Computing Platform

5.2 Energy Measurement

Monitoring energy expenditure is a critical performance benchmark in our evaluation. To measure the energy consumption, we used an INA219 Current Sensor IC, as shown in Figure 8, interfaced to an Arduino Uno Microcontroller board via I^2C bus. The mechanism of power calculation in the INA219 involves measuring the voltage drop U_{sense} across a sense resistor connected in series to the main power rail of the device whose energy consumption is to be monitored. The electrical resistance of the sense resistor, R_{sense}, should

Device Type	CPU	GPU	Memory	OS
Pi3	1.2 GHz quad-core ARM Cortex-A53	N/A	1GB LPDDR2	Raspbian
TX2	2.0 GHz ARM quad-core Cortex-A57	256-core NVIDIA Pascal	8GB LPDDR4	Linux (Ubuntu)
TK1	2.32 GHz ARM quad-core Cortex-A15	192-core NVIDIA Kepler	2GB LPDDR3	Linux (Ubuntu)
PC	2.7 GHz i5-7500T quad-core	N/A	8GB LPDDR4	Windows 10

Table 1: Specifications of Edge Devices in SSEC

ideally be minute in comparison to load resistance to prevent any side-effects (i.e., ($R_{sense} << R_{load}$)) [39]. The INA219 amplifies the voltage drop U_{sense}, converts the analog reading to digital using an on-board ADC and computes the power consumption at any given instant P_{load} as follows:

$$P_{load} = \frac{U_{sense}}{R_{sense}} \times U_{load} \tag{9}$$

where U_{load} is the main rail bus voltage potential.

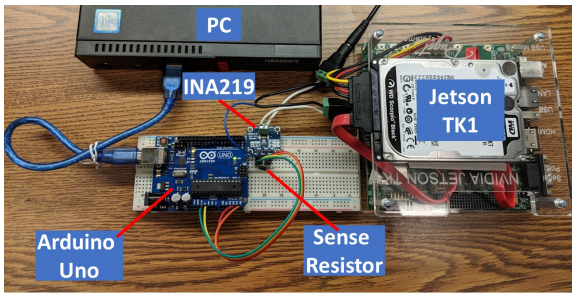


Figure 8: Power Monitoring Module

5.3 Experiment Setup

We choose the following representative baselines from recent literature.

- **Random Assignment (Rand)**: A heuristic computation allocation scheme where the social sensing tasks are randomly assigned to edge devices [44].
- **Greedy Shortest Path (GSP)**: A heuristic resource allocation scheme where each job greedily picks the shortest path of the supply chain graph to minimize the energy and delay cost [11].
- **Centralized Edge Server-based Allocation (CES)**: A centralized resource management scheme where an edge device sends all its computation tasks to the local edge server [34].
- **Bottom-Up Game-theoretic Task Allocation (BGTA)**: A game-theoretic edge computing resource allocation scheme for non-cooperative edge devices. It uses a distributed Fictitious Play algorithm to allow edge devices to selfishly pick tasks and eventually achieve consensus [20].

Note that there exist some systems that are also related to harnessing the heterogeneous computing resources such as HTCondor [19], CoGTA [47], and FemtoCloud [9]. However, the homogeneous task assumption in these systems does not hold in our problem setting as discussed in Section 1. Therefore, we do not include them as baselines.

5.4 Case Study 1: Disaster Damage Assessment

The first case study is *Disaster Damage Assessment (DDA)* where participants are tasked to sense and evaluate whether damages (e.g., potholes and collapsed houses caused by an earthquake) have happened and to what extent to the assigned locations during a natural disaster. The output of this application offers real-time situation awareness and timely alerts to citizens in the affected areas of disasters.

We collected 2,000 images related to the Ecuador Earthquake in 2016 from Instagram and Twitter. We run the application over 100 sensing cycles. The images are organized by the timestamp and are split into 100 subsets and each of which is processed in a sensing cycle. The social sensing jobs in this application consist of 3 pipeline tasks summarized below.

Tasks for DDA: i) edge devices equipped with cameras (e.g., dash cameras, UAVs) are tasked to capture live images of locations of interest; ii) extracting Damage Detection Map (DDM) features using Convolutional Neural Network (CNN) model from raw images; iii) assess damage severity from DDM using the algorithm in [17].

5.4.1 Quality of Service. In the first set of experiments, we focus on how the objective is achieved from the application side. In particular, we evaluate the deadline hit rate (DHR) and end-to-end (E2E) delay of all the compared schemes. The DHR is defined as the ratio of tasks that are completed within the deadline. The results are shown in Figure 9. We use all 10 edge devices and gradually increase the deadline constraints. We observe that HeteroEdge has significantly higher DHRs than all the baselines and is the first one that reaches 100% DHR as the deadline increases. We attribute such performance gain to our SCSR algorithm that finds the optimal "supply chain path" that allows the edge devices to search for the most efficient way to collaboratively finish social sensing jobs.

Figure 10 summarizes the E2E delays of all the schemes as the number of jobs varies. We show both the average delay and the 90% confidence bounds of the results. We observe that our HeteroEdge scheme has the least E2E delay and tightest confidence bounds compared to the baselines. The results further demonstrate the effectiveness of HeteroEdge for meeting real-time QoS requirements of the application. The performance gain of the HeteroEdge is achieved by explicitly modeling the dynamic context of the edge devices (i.e., the dynamic worker pool) and allocating tasks according to the current device status.

5.4.2 Energy Consumption. In the second set of experiments, we focus on the energy consumption of edge devices. As mentioned

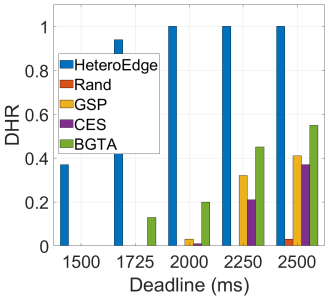


Figure 9: DHR in DDA

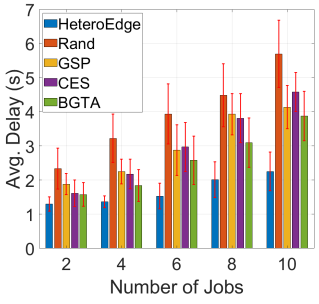


Figure 10: E2E Delay in DDA

in Section 4, the energy consumption is normalized to reflect the proportion of battery that is consumed by a scheme to accomplish all social sensing jobs. The reason for this normalization is to avoid the unfair scenario where minimizing the absolute energy would end up with a strategy that always pushes heavy computation from high-power devices to low-power ones. The results of the average normalized energy consumption on edge devices are shown in Table 2. We use all 10 edge devices and set the number of jobs to 10 and deadline to 3 seconds. We can observe that HeteroEdge consumes significantly less energy as compared to all other baselines except CES. CES consumes the least amount of energy on every edge device because it simply pushes all the computation tasks to the local edge server. In another word, the CES scheme under-utilizes the diverse resources on the edge devices and pushes the extra burden to the server. The results illustrate that the edge devices can achieve the longest battery life under HeteroEdge, which is particularly important for edge devices with limited power supply.

Table 2: Normalized Energy Consumption in DDA

	HeteroEdge	RAND	GSP	CES	BGTA
Jetson TX2	0.913	0.915	0.897	0.572	0.901
Jetson TK1	0.824	0.985	0.904	0.754	0.877
Raspberry Pi3	0.613	0.952	0.803	0.589	0.724
PC	0.798	0.844	0.831	0.766	0.825
All	7.337	9.404	8.448	6.363	8.001

"All" denotes the sum of normalized energy consumption of all 10 devices.

5.5 Case Study 2: Collaborative Traffic Monitoring

The second case study is *Collaborative Traffic Monitoring (CTM)* where participants in a social sensing application use personal mobile devices (e.g., mobile phones, dash cameras) to record and analyze the current traffic conditions. For example, a traffic monitoring application can task a set of drivers to use their dash cameras to take videos of the traffic in front of their vehicles and then infer the congestion rate of the road.

We collected the video data using dash cameras from two vehicles. The data contains a total of 30 video clips and 15 of them are used for training. We divided the application into 100 sensing cycles and each sensing cycle processes video clips of 6 seconds (with each video sampled at 15 frames per cycle). The social sensing job in this application consists of 4 pipeline tasks summarized below.

Tasks for CTM: i) data collection of traffic video signal as image frames; ii) extracting optical flow and Histogram of oriented gradients (HOG) features; iii) object detection using trained SVM to identify vehicle counts; iv) infer the overall traffic condition based on the detected vehicles using the algorithm in [51].

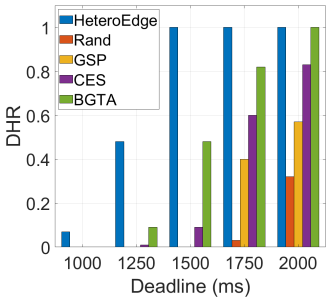


Figure 11: DHR in CTM

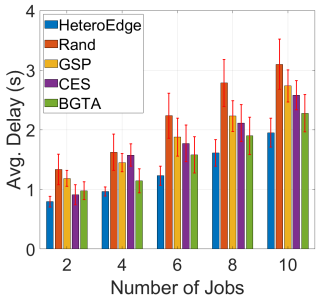


Figure 12: E2E Delay in CTM

5.5.1 Quality of Service. We perform similar experiments as those discussed in the previous case study. In particular, we evaluate all the schemes in terms of DHR and E2E delay. The results are shown in Figure 11 and Figure 12 respectively. We observe similar results of HeteroEdge as the previous case study. This continues to show that the HeteroEdge scheme can provide desirable QoS under different application scenarios.

5.5.2 Energy Consumption. The results of energy consumption at the edge devices are shown in Table 3. We observe that our scheme continues to provide significantly more energy savings to the edge devices than other baselines. This again demonstrates that HeteroEdge is more energy efficient ("user-friendly") by offering participating edge devices a longer battery life.

Table 3: Normalized Energy Consumption in CTM

	HeteroEdge	RAND	GSP	CES	BGTA
Jetson TX2	0.817	0.891	0.884	0.572	0.875
Jetson TK1	0.841	0.903	0.863	0.722	0.886
Raspberry Pi3	0.696	0.913	0.833	0.580	0.822
PC	0.819	0.837	0.820	0.775	0.825
All	7.615	8.990	8.479	6.263	8.457

"All" denotes the sum of normalized energy consumption of all 10 devices.

5.6 Convergence and Scalability

Finally, we study the convergence and computation overhead of the resource management scheme (i.e., SCSR) in HeteroEdge. We set K =1 and the unit cost as 0.1 to normalize the link costs. Figures 13 and 14 show the average number of iterations of SCSR till convergence when we change the number of devices. We observe that the number of iterations significantly decreases as the ϵ value increases. Here ϵ controls how likely a player would change its strategy. The lower value is, the more likely a player is going to change its strategy in

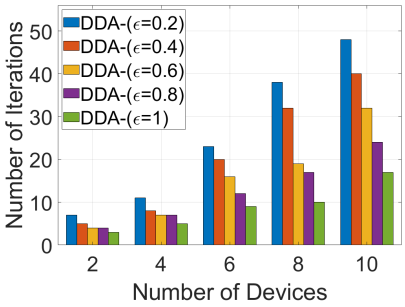


Figure 13: Convergence (DDA) of SCSR

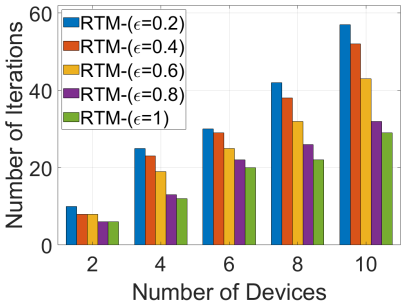


Figure 14: Convergence (CTM) of SCSR

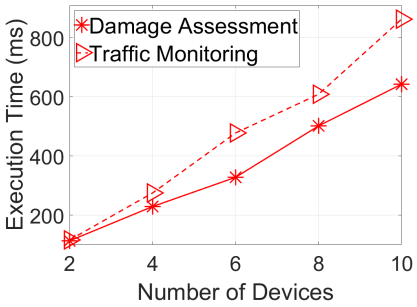


Figure 15: Execution Overhead of SCSR

the game, which often requires more iterations for the algorithm to reach the convergence. The curves also show a linear trend as the number of devices increases. These results also verify the convergence analysis of SCSR scheme in Section 4.3.4.

Figure 15 shows the execution time of SCSR. The execution time includes the running time of the SCSR algorithm as well as the communication delay between the edge servers and the edge devices. We observe the execution time of SCSR grows almost linearly as the number of edge devices increases. The above results again demonstrate the suitability of using HeteroEdge for delay-sensitive social sensing applications. We note that the execution time of the SCSR scheme might still become a non-trivial overhead when the number of edge devices in the system becomes very large. A possible solution to such a scalability problem is to increase the number of local edge servers and run HeteroEdge in the cluster of edge devices coordinated by the same local edge server. This solution is practical in real-world applications thanks to the increasingly popular hierarchical structure of edge computing systems [13].

6 CONCLUSION AND FUTURE WORK

This paper presents the HeteroEdge framework to addresses fundamental challenges in taming heterogeneity issue in social sensing based edge computing (SSEC) systems. We have implemented our proposed framework on a real-world edge computing testbed including Nvidia Jetson TK1, TX2, Raspberry Pi3 boards, and personal computer. The evaluation results from two real-world social sensing applications demonstrate that HeteroEdge achieves significant performance gains compared to state-of-the-art baselines.

Our work has some limitations that deserve further investigation. First, HeteroEdge entails security concerns. In particular, we assume that edge devices are cooperative (e.g., they will not quit abruptly or intentionally delay the execution). However, this assumption may not hold in scenarios where malicious devices do exist: they may intentionally delay the task execution, making it miss its deadline. This issue can be mitigated by adding an extra function in HeteroEdge to keep track of the normal behaviors of edge devices and actively block the identified lazy devices.

Second, HeteroEdge is a soft real-time task allocation scheme that minimizes delay of the system instead of providing the hard deadline guarantee. This is due to several factors. First, the worst-case estimation of the task execution time is not precise due to the complicated computing and communication environment in SSEC systems. Second, the convergence time of the Nash Equilibrium solution is also dynamic and hard to predict precisely. In the future, we plan to explore more sophisticated execution time prediction schemes (e.g., static program analysis [10] and narrow spectrum benchmarking [32]) in HeteroEdge.

Finally, our current experiment platform consists of a limited number of edge devices and the scalability aspect of HeteroEdge deserves further investigation. The HeteroEdge has the nice property of guaranteed Nash Equilibrium and is shown to have quick convergence in real-world social sensing applications. In the future work, we plan to perform additional simulation studies to investigate the scalability of HeteroEdge using the simulator in [41] that is specifically designed for heterogeneous edge devices.

ACKNOWLEDGMENT

This research is supported in part by the National Science Foundation under Grant No. CNS-1831669, CBET-1637251, CNS-1566465 and IIS-1447795, Army Research Office under Grant W911NF-17-1-0409, Google 2017 Faculty Research Award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. We also thank our shepherd Dr. Mirco Musolesi for providing valuable input to our paper.

REFERENCES

[1] Karl Aberer, Manfred Hauswirth, and Ali Salehi. 2006. *Global sensor networks.* Technical Report.

[2] Ishfaq Ahmad, Sanjay Ranka, and Samee Ullah Khan. 2008. Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.* IEEE, 1–6.

[3] Diana Bautista, Julio Sahuquillo, Houcine Hassan, Salvador Petit, and José Duato. 2008. A simple power-aware scheduling for multicore systems when running real-time applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.* IEEE, 1–7.

[4] Jie Cao, Lanyu Xu, Raef Abdallah, and Weisong Shi. 2017. EdgeOS_H: A Home Operating System for Internet of Everything. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on.* IEEE, 1756–1764.

[5] Matt Curtin. 1998. Write once, run anywhere: Why it matters. *Technical Article. http://java. sun. com/features/1998/01/wo* (1998).

[6] Johan Eker, Jörn W Janneck, Edward A Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. 2003. Taming heterogeneity-the Ptolemy approach. *Proc. IEEE* 91, 1 (2003), 127–144.

[7] Wei Gao. 2014. Opportunistic peer-to-peer mobile cloud computing at the tactical edge. In *Military Communications Conference (MILCOM), 2014 IEEE*. IEEE, 1614–1620.

[8] Alex Glikson, Stefan Nastic, and Schahram Dustdar. 2017. Deviceless edge computing: extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*. ACM, 28.

[9] Karim Habak, Mostafa Ammar, Khaled A Harras, and Ellen Zegura. 2015. Femto clouds: Leveraging mobile devices to provide cloud service at the edge. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*. IEEE, 9–16.

[10] Reinhold Heckmann and Christian Ferdinand. 2004. Worst-case execution time prediction by static program analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004, pages 26–30. IEEE Computer Society.*

[11] Stefan Irnich and Guy Desaulniers. 2005. Shortest path problems with resource constraints. In *Column generation*. Springer, 33–65.

[12] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2010. Modeling GPU-CPU workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 31–42.

[13] Abbas Kiani and Nirwan Ansari. 2017. Toward hierarchical mobile edge computing: An auction-based profit maximization approach. *IEEE Internet of Things Journal* 4, 6 (2017), 2082–2091.

[14] Karthik Kumar and Yung-Hsiang Lu. 2010. Cloud computing for mobile users: Can offloading computation save energy? *Computer* 43, 4 (2010), 51–56.

[15] Agus Kurniawan. 2018. Learning AWS IoT: Effectively Manage Connected Devices on the AWS Cloud Using Services Such as AWS Greengrass, AWS Button, Predictive Analytics and Machine Learning.

[16] Edward A Lee, John D Kubiatowicz, Jan M Rabaey, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, John Wawrzynek, David Blaauw, Prabal Dutta, Kevin Fu, Carlos Guestrin, et al. 2012. The terraswarm research center (TSRC)(a white paper). *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-207* (2012).

[17] Xukun Li, Huaiyu Zhang, Doina Caragea, and Muhammad Imran. 2018. Localizing and Quantifying Damage in Social Media Images. *arXiv preprint arXiv:1806.07378* (2018).

[18] Michael J Litzkow. 1987. Remote Unix: Turning idle workstations into cycle servers. In *Proceedings of the Summer USENIX Conference*. 381–384.

[19] Michael J Litzkow, Miron Livny, and Matt W Mutka. 1988. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*. IEEE, 104–111.

[20] Dongqing Liu, Lyes Khoukhi, and Abdelhakim Hafid. 2017. Decentralized data offloading for mobile cloud computing based on game theory. In *Fog and Mobile Edge Computing (FMEC), 2017 Second International Conference on*. IEEE, 20–24.

[21] Yang Liu, Changqiao Xu, Yufeng Zhan, Zhixin Liu, Jianfeng Guan, and Hongke Zhang. 2017. Incentive mechanism for computation offloading using edge computing: a Stackelberg game approach. *Computer Networks* 129 (2017), 399–409.

[22] Zhidan Liu, Zhenjiang Li, Kaishun Wu, and Mo Li. 2018. Urban Traffic Prediction from Mobility Data Using Deep Learning. *IEEE Network* 32, 4 (2018), 40–46.

[23] Cewu Lu, Jianping Shi, and Jiaya Jia. 2013. Abnormal event detection at 150 fps in matlab. In *Proceedings of the IEEE International Conference on Computer Vision*. 2720–2727.

[24] Xufei Mao, Xin Miao, Yuan He, Xiang-Yang Li, and Yunhao Liu. 2012. CitySee: Urban CO 2 monitoring with sensors. In *INFOCOM, 2012 Proceedings IEEE*. IEEE, 1611–1619.

[25] Akihiko Matsui. 1992. Best response dynamics and socially stable strategies. *Journal of Economic Theory* 57, 2 (1992), 343–362.

[26] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[27] Nvidia. [n. d.]. Jetson Tegra K1. http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

[28] Nvidia. [n. d.]. Jetson Tegra X2. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html

[29] Robin Wentao Ouyang, Lance M Kaplan, Alice Toniolo, Mani Srivastava, and Timothy Norman. [n. d.]. Parallel and Streaming Truth Discovery in Large-Scale Quantitative Crowdsourcing. ([n. d.]).

[30] Panagiota N Panagopoulou and Paul G Spirakis. 2005. Efficient convergence to pure Nash equilibria in weighted network congestion games. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 203–215.

[31] Raspberry. [n. d.]. Raspberry Pi 3B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[32] Rafael Hector Saavedra-Barrera. 1992. *CPU performance evaluation and execution time prediction using narrow spectrum benchmarking*. Ph.D. Dissertation. University of California, Berkeley.

[33] Zohreh Sanaei, Saeid Abolfazli, Abdullah Gani, and Rajkumar Buyya. 2014. Heterogeneity in mobile cloud computing: taxonomy and open challenges. *IEEE Communications Surveys & Tutorials* 16, 1 (2014), 369–392.

[34] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009).

[35] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. 2016. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 258–269.

[36] Chenguang Shen and Mani Srivastava. 2017. Exploring Hardware Heterogeneity to Improve Pervasive Context Inferences. *Computer* 50, 6 (2017), 19–26.

[37] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[38] Nitin Sinha, Korrapati Eswari Pujitha, and John Sahaya Rani Alex. 2015. Xively based sensing and monitoring system for IoT. In *Computer Communication and Informatics (ICCCI), 2015 International Conference on*. IEEE, 1–6.

[39] Kristoffer Robin Stokke, Håkon Kvale Stensland, Carsten Griwodz, and Pål Halvorsen. 2015. Energy efficient video encoding using the tegra K1 mobile processor. In *Proceedings of the 6th ACM Multimedia Systems Conference*. ACM, 81–84.

[40] Hang Su and Dakai Zhu. 2013. An elastic mixed-criticality task model and its scheduling algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 147–152.

[41] Nathan Vance, Ryan Mackey, Daniel Zhang, and Dong Wang. 2018. Simulating Large-Scale Social Sensing Based Edge Computing Systems with Heterogeneous Network Configurations. In *2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE.

[42] Nathan Vance, Daniel Zhang, Yang Zhang, and Dong Wang. 2018. Privacy-aware Edge Computing in Social Sensing Applications using Ring Signatures. In *icpads*. IEEE.

[43] Dong Wang, Boleslaw K Szymanski, Tarek Abdelzaher, Heng Ji, and Lance Kaplan. 2018. The age of social sensing. *IEEE Computer* (2018).

[44] Drew Wicke, David Freelan, and Sean Luke. 2015. Bounty hunters and multiagent task allocation. In *Proceedings of the 2015 international conference on autonomous agents and multiagent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 387–394.

[45] Shanhe Yi, Cheng Li, and Qun Li. 2015. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*. ACM, 37–42.

[46] Daniel Zhang, Yue Ma, Yang Zhang, Suwen Lin, X Sharon Hu, and Dong Wang. 2018. A real-time and non-cooperative task allocation framework for social sensing applications in edge computing systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 316–326.

[47] Daniel Zhang, Yue Ma, Chao Zheng, Yang Zhang, X Sharon Hu, and Dong Wang. 2018. Cooperative-Competitive Task Allocation in Edge Computing for Delay-Sensitive Social Sensing. In *Proceedings of the Third ACM/IEEE Symposium on Edge Computing*. ACM.

[48] Daniel Yue Zhang, Chao Zheng, Dong Wang, Doug Thain, Xin Mu, Greg Madey, and Chao Huang. 2017. Towards scalable and dynamic social sensing using a distributed computing framework. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 966–976.

[49] Charles Zheng and Douglas Thain. 2015. Integrating containers into workflows: A case study using makeflow, work queue, and docker. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*. ACM, 31–38.

[50] Qi Zhu, Haibo Zeng, Wei Zheng, Marco DI Natale, and Alberto Sangiovanni-Vincentelli. 2012. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)* 11, 4 (2012), 85.

[51] Ahmet Özlü. 2018. Vehicle Detection, Tracking and Counting by TensorFlow. https://github.com/ahmetozlu/vehicle_counting_tensorflow