A Streaming Cloud Platform for Real-Time Video Processing on Embedded Devices

Weishan Zhang*, Haoyun Sun*, Dehai Zhao†, Liang Xu‡, Xin Liu*, Huansheng Ning‡, Jiehan Zhou¶‖, Yi Guo *, Su Yang§

*School of Computer and Communication Engineering, China University of Petroleum (East China), Qingdao, China.

†Australian National University, Australia

‡College of Computer and Communication Engineering, Beijing University of Science and Technology, Beijing, China

¶University of Oulu, Finland

‖University of Toronto, Canada

§College of Computer Science and Technology, Fudan University, Shanghai, China

Abstract—Real-time intelligent video processing on embedded devices with low power consumption can be useful for applications like drone surveillance, smart cars, and more. However, the limited resources of embedded devices is a challenging issue for effective embedded computing. Most of the existing work on this topic focuses on single device based solutions, without the use of cloud computing mechanisms for parallel processing to boost performance. In this paper, we propose a cloud platform for real-time video processing based on embedded devices. Eight NVIDIA Jetson TX1 and three Jetson TX2 GPUs are used to construct a streaming embedded cloud platform (SECP), on which Apache Storm is deployed as the cloud computing environment for deep learning algorithms (Convolutional Neural Networks - CNNs) to process video streams. Additionally, self-managing services are designed to ensure that this platform can run smoothly and stably, in the form of a metric sensor, a bottleneck detector and a scheduler. This platform is evaluated in terms of processing speed, power consumption, and network throughput by running various deep learning algorithms for object detection. The results show the proposed platform can run deep learning algorithms on embedded devices while meeting the high scalability and fault tolerance required for real-time video processing.

Index Terms—embedded devices, low power consumption, deep learning, video processing, convolutional neural networks

1 INTRODUCTION

Embedded devices equipped with impressive computing capacities are being increasingly adopted in various domains. For example, some drones can monitor public places using aerial photography to help to maintain public safety, or can be used for goods delivery, and even for power grid monitoring to ensure their targets are working reliably. For such applications, there are challenging issues in the way of achieving good quality services for embedded video or image processing in real-time:

- **Low real-time recognition accuracy.** Only light-weight network structures [1] can run on drones in real-time due to hardware limitations; this limits the possibility of achieving high accuracy in a complex environment compared with deep network structures [2].
- **Bad performance of on-line recognition.** At this time, usually only simple tasks such as video collection and transferring are performed on board by drones and robots, and then the collected video is processed on a remote cloud server to conduct analysis. This makes it difficult to fulfill real-time requirements, and suffers latency that may cause loss of both life and assets. For example, the real-time recognition of an early stage fire around a power line may help to reduce potential damages.

Deep learning is currently applied successfully in various domains, such as natural language processing [3] and computer vision [4], and can even exceed human performance in some cases [5]. It has been applied for aerial video analysis due to its high accuracy in many computer vision tasks [6], [7], [8], [2]. However, these high-performance deep neural networks have large-scale weight parameters and require billions of floating point operations, which result in high power consumption. For example, AlexNet has 61M parameters (249MB of memory) and performs 1.5B high precision operations to classify one image. These numbers are even higher for deeper CNNs, e.g., VGG [7]. Even though heavy-weight tasks can be migrated to cloud servers [9], bandwidth, latency, and network availability are still major factors hindering the responsiveness of near real-time applications.

Efforts to address these issues have taken two directions: 1) network structure modification, and, 2) platform construction.

Binarization is an effective approach to reduce neural

Manuscript received July. 1, 2018; revised. Corresponding author: Weishan Zhang (email: zhangws@upc.edu.cn); Xin Liu (lx@upc.edu.cn); Jiehan Zhou (jiehan.zhou@oulu.fi)

network size. In Binarized Neural Networks (BNNs) [10], weights and activations are constrained to either +1 or -1, which requires 32 times less memory but gives 7 times the performance acceleration on MNIST data compared with the non-optimized model. XNOR-Net [11] approximates both weights and inputs to convolutional and fully connected layers with binary values, which offers a 58 times speed up in CPUs.

Google proposed MobileNets [12], which is a small, low latency model which can be easily matched to requirements of mobile and embedded vision applications. Apple's Core ML[1]is a foundational machine learning framework used by Apple products, enabling developers to build smart applications on embedded devices. Despite successful network size reduction, these models still cannot achieve as high accuracy as 32-bit DNNs [6] in processing color images.

Other work has focused on building embedded platforms and adopting deep learning algorithms. For example, Mao et al. [13] ran the Fast R-CNN on a Jetson TK1 platform[2]. Although additional modifications on the Fast R-CNN were made to fit TK1, the detection speed was very low (1.85 frames per second; hereafter, fps). Another study by Jagannathan et al. [14] ran a 7-layer CNN on TDA3x SoC for object classification, and the overall system performance was 15fps. The majority of existing work (Ratnayake et al. [15], Bako et al. [16], Nikitakis et al. [17], Chen et al. [18], Arva et al. [19]) used only a single embedded device to conduct video processing, which is obviously not capable of fulfilling real-time video processing (24fps). Gao et al. [20] built an embedded cluster for video processing, but it lacked a complete solution to address system stability and fault tolerance. Therefore, a powerful software/hardware platform is needed to support efficient embedded deep learning based real-time video processing.

We propose a streaming embedded cloud platform (SECP) for real-time intelligent video processing using embedded GPUs, supported by cloud computing. We apply both NVIDIA Jetson TX1 and TX2[3] to build the streaming embedded cloud platform for real-time video intelligent processing. On SECP, Apache Storm[4] is deployed as the runtime environment for deep learning algorithms to provide parallel processing. In order to make services both stable and fault-tolerant, some self-managing services are designed to conduct sensing, detecting, and scheduling functionalities, including a metric sensor, a bottleneck detector, and a scheduler. The metric sensor collects metrics from Storm to monitor the health status of the platform. The bottleneck detector finds computing bottlenecks, and the scheduler executes the decision to manage resources.

We conducted comprehensive evaluations of an SECP, including on-line video processing based on deep learning, capability to find bottlenecks, and ability to make reasonable decisions when there are abnormal situations. The contributions for this paper include:

- A novel streaming embedded cloud platform-SECP for real-time intelligent video processing is proposed that combines the power of stream processing and embedded GPUs' parallel processing capabilities, which can run deep neural networks efficiently.
- Self-managing services are designed to ensure SECP fault-tolerance for continuous stream processing by monitoring CPUs, GPUs and network throughput of the platform.
- A scheduler service applies a video frame emitting strategy to respond to faults or failures while minimizing data loss.

The rest of this paper is organized as follows. Section 2 gives a detailed description of the proposed SECP platform and its implementation. Section 3 evaluates the SECP. Section 4 presents related work. Conclusion and future work are given in section 5.

2 SECP Platform Overview

The SECP platform aims to build a streaming embedded cloud for intelligent real-time video processing, consisting of hardware, deep learning algorithms, and a streaming processing framework. In addition, self-managing services with a monitor and a scheduler are designed to ensure appropriate decisions on faults.

2.1 Hardware design of SECP

2.1.1 Embedded GPUs used in SECP

Deep neural networks are computing intensive for their heavy floating point calculation, which can be accelerated by GPUs. To ensure the performance of SECP, we use commercial embedded GPUs. The NVIDIA Jetson is an industry-leading embedded computing device, which currently has two versions: TX1 and TX2. Table 1 lists their main properties related to our work.

TABLE 1: Jetson TX1 versus Jetson TX2

	Jetson TX1	Jetson TX2
GPU	NVIDIA Maxwell 256 CUDA cores	NVIDIA Pascal 256 CUDA cores
CPU	64-bit A57 CPUs	64-bit Denver 2 and A57 CPUs
Memory	4GB 64-bit LPDDDR4	8GB 128-bit LPDDR4
Storage	16GB eMMC	32GB eMMC
Video Encode	4K×2K 30Hz	4K×2K 60Hz
Video Decode	4K×2K 60Hz	4K×2K 60Hz
Camera	1.4Gpix/s 1.5Gbps per lane	1.4Gpix/s 2.5Gbps per lane
Connectivity	1 Gigabit Ethernet, 802.11ac WLAN	

2.2 SECP Hardware Design

We designed two platforms for each version of NVIDIA Jetson device. A baseboard is designed for each platform to integrate three modules: power supply, communication, and computing. Then TX1 or TX2 GPUs can be plugged into the baseboard.

The computing module consists of three interfaces for each of the TX1/TX2 cores. The computing module enables horizontal scaling, as its computing cores can be added and removed arbitrarily according to computing demands. The communication module uses a Gigabit Ethernet RTL8370

1. https://developer.apple.com/machine-learning/
2. http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html
3. https://developer.nvidia.com/embedded-computing
4. http://storm.apache.org

switching chip to achieve rapid communication between computing cores. We also include a separate USB3.0 interface and a HDMI interface for platform debugging purposes.

2.3 Designing the SECP Software Components

2.3.1 Choosing Deep Learning Algorithms

The platform runs video object detection algorithms to process real-time videos. We compare a set of algorithms in terms of accuracy and speed running on an NVIDIA TITAN X using the Visual Object Classification Challenge 2007 (VOC2007) dataset. The results are listed in Table 2.

TABLE 2: Performance of various algorithms

Name	mAP (VOC2007)	fps (TITAN X)
R-CNN [21]	66.0%	0.02
SPPNet [22]	63.1%	0.43
Fast R-CNN [23]	66.9%	3.13
Faster R-CNN [24]	73.2%	7
YOLO [25]	63.4%	45
YOLOv2 [26]	76.8%	67
Tiny YOLO [25]	57.1%	207
SSD300 [27]	74.3%	46
SSD500 [27]	76.8%	19

The R-CNN (Girshick et al. [21]) based object detection algorithms are very slow because they have complex steps: 1) coarse-grained region proposal generation; 2) CNN feature extraction and object detection; and 3) fine-grained bounding box compression and regression. Although they fused multiple stages together and avoided redundant computations in later works [22], [23], [24], these algorithms are too slow to fulfill real-time requirements.

SSD [27] can achieve both high accuracy and high speed using an NVIDIA TITAN X GPU. Although the TITAN X's computing capacity exceeds that of NVIDIA Jetson, it is not practical to reproduce such performance on an embedded platform.

YOLO [25] considers object detection as a regression problem to spatially separated bounding boxes and associated class probabilities. A single neural network is used to predict bounding boxes and class probabilities directly from full images in one evaluation. This method achieves both good accuracy and high speed so that it is suitable to be transformed in an embedded platform. In the rest of the paper, we build the object detection algorithms based on YOLO v2.

2.3.2 Parallel GPU processing using the Storm framework

In order to overcome the limitations of a single embedded GPU, in our work we built an embedded cloud to overcome the computing capacity on embedded devices so that devices can process videos in parallel. In such an embedded cloud, each device is responsible for only one portion of the video data partitioned over the SECP, and the final results from different devices are merged.

We utilize a real-time stream processing framework (Apache Storm) to design the parallel video processing capabilities for SECP. The reason we chose Storm as the underlying cloud computing infrastructure is that it can handle an event with a sub-second latency, which is much faster than other options like Spark Streaming[5] which can incur a second-order delay. A Storm job called topology consists of spouts, bolts, tuples, streams and grouping specifications.

In the SECP, spouts receive data and emit it to other computing nodes called bolts. First, a spout pulls video stream from a camera. Then a spout coverts the video stream to video frames. A spout also has a message queue to emit the video frames to other bolts. Bolts are computing nodes. Since the processing time varies between different video frames, by default, every bolt receives the next video frame upon the completion of the current video frame. Video frames are emitted from the message queue to three bolts randomly. Then the video frames are processed in parallel by three bolts. Finally, to reconstruct the video correctly we add time stamps to every frame in the message queue to maintain their order in the video. All processing steps on Storm are called Storm tasks.

Given this architecture, we built a video object detection job as a Storm topology. We deploy the YOLO v2 as our object detection algorithm on three bolts, and the three bolts are deployed on three computing GPUs respectively. Finally, the results are merged on the bolt in the third layer. Please refer to Figure 2 which demonstrates how this works.

2.3.3 SECP Metric Sensor

To monitor the status of a video processing job, we designed a metric sensor to collect status of Storm tasks. The collected measurement data are stored for analysis and decision making. We mainly consider the following metrics:

- *Latency*
  Latency represents the delay between the starting time from the video frame emitted from a spout, and the ending time when the results are output. Latency measures our platform's video processing speed.
- *Queue length*
  The length of message queue indicates if the video data collection speed matches the video processing speed. An empty queue indicates the processing platform is idle and waiting for video data to be collected, and a long queue shows the processing platform is overwhelmed and cannot cope with the current speed of the data collection. Both situations require the platform to be tuned.
- *CPU utility*
  CPU utility is the ratio of the CPU time consumed in one task to its total time interval.
- *GPU utility*
  Similarly, GPU utility is the ratio of the GPU time consumed in one task to its total time interval.
- *Traffic*
  Traffic measures the data bulk passing through the topology in a given time period.

Storm provides APIs for accessing topology-level and cluster-level metrics. Hardware level metrics (such as GPU utility) are obtained by Jetson's APIs. To develop a general metric sensor, for all the metrics at different levels, we designed an API model of the metric sensor as shown in Figure 1.

5. http://spark.apache.org/streaming/

First, an interface called IMetric is defined for collecting metrics, which is implemented by GPUMetric, CPUMetric, and TrafficMetric. CPUMetric and TrafficMetric extend BaseRichBolt and BaseRichSpout of Storm APIs respectively. GPUMetric extends TegraStates of the NVIDIA Jetson APIs. Second, we set TopologyMetricConsumer as the default metric consumer to receive metric messages in a time interval. It calls TopologyMetricEvent that composes ComponentMetricEvent to get metrics. All the metrics are collected in an interval of one second.

2.3.4 The Bottleneck Detector

The bottleneck detector detects abnormal events. A self-learning detection algorithm is ideal, but not practical for two reasons. First, the detection algorithm needs to be light-weight. Second, the training data requires a significant sample size with manual labels of abnormal events. This prevents use of a supervised learning algorithm. Instead, we adopted a simple statistical method.

We calculate the mean value for each metric by equation (1).

$$MetricMean_k = \frac{\sum_{i=1}^{T} \frac{Metric_k^t}{MaxMetricValue_k}}{T} \quad (1)$$

Where k represents the k-th metric and t is the time point. We conduct normalization that divides each metric by its theoretical max value to make sure each metric has the same scale. T is the number of time interval and usually set to 10, which can achieve quick and accurate response. Another statistic method is variance calculated by equation (2).

$$\sigma_k^2 = \frac{\sum_{i=1}^{T}(Metric_k^t - MetricMean_k)^2}{T} \quad (2)$$

We set two thresholds, $MeanThreshold = 0.9$ and $VarianceThreshold = 0.0025$ for the bottleneck detector. Only when $MetricMean_k > MeanThreshold$ and $\sigma_k^2 < VarianceThreshold$, we determine the kth metric is facing a bottleneck. $\sigma_k^2 > VarianceThreshold$ represents the cluster is not stable enough. The platform should make suitable decisions according to the bottleneck detection results.

2.3.5 Task Scheduler

The scheduler acts as the decision executor that calls Storm APIs to configure Storm parameters and adjusts the video frame emitting strategy under the situation of a bottleneck. The aim is to keep the SECP available with the least loss of data.

Figure 2 shows an example. In part (a), a computing core crashes causing a bolt to shut down. The video frames in the spout message queue are blocked and the latency increases.

More seriously, if the blocked video frames keep accumulating in the message queue, the spout will end up with an 'out of memory' exception. Although a defender is designed to empty the memory by force when the message queue is full, video frames are lost. In this situation, the bottleneck detector identifies the abnormalities in memory, GPU utility, and traffic, and the scheduler drops one frame while emitting two frames, as shown in part (b). The idea of this strategy is that continuous frames contain more information than single frames, especially in an object detection task.

The schedule scheme is described in Algorithm 1. The video frames are divided into blocks of fixed length. If the computing nodes are powerful enough to process the video data, we do not need to drop any frames. But for the situation of low computing capacity (e.g. some computing nodes crash), as the length of message queue increases, the number of dropped frames will go up. When the crashed computing nodes recovered, the dropped frames will decrease. This scheme can guarantee that SECP can run stably with the lowest data loss using dynamic adjustment of dropped frames.

Algorithm 1 Schedule algorithm of SECP

Require:
CL: Current length of message queue
$MaxL$: Maximum length of message queue
$MinL$: Minimum length of message queue
$Block = B_n$: A block is a set of video frames
BL: Length of a block
df = 0 : Number of dropped frames
Ensure:
for all $B_n \in Block$ **do**
  if $CL > MaxL$ **and** $df < BL$ **then**
    $df += 1$
  end if
  if $CL < MinL$ **and** $df > 0$ **then**
    $df -= 1$
  end if
  for $t = 0$ **to** $t = df$ **do**
    $B_n.delete()$
  end for
  for $t = 0$ **to** $t = BL - df$ **do**
    VIDEO PROCESS($B_n.pop()$)
  end for
end for

2.4 SECP platform

The SECP platform is shown as a component diagram in Figure 3. The metric sensor and scheduler depend on Storm directly. The metric sensor calls Storm APIs to get each task's metrics and the scheduler calls Storm configuration to configure it. The bottleneck detector reads metrics obtained by the metric sensor and provides an interface for the scheduler to decide on which strategy is to be executed. Storm calls camera APIs to pull video streams from the camera. A video processing interface is designed for Storm to call video processing algorithms such as image recognition and object detection.

The SECP platform has multiple nodes with the same computing capability and configuration, which means that it is easy to make these nodes work in parallel. When there is a video input, the platform first clips it into frames and each frame is given a unique time stamp to facilitate the final result merging process. We use f_n to denote the video frames where n is the time stamp. The clipped video frames

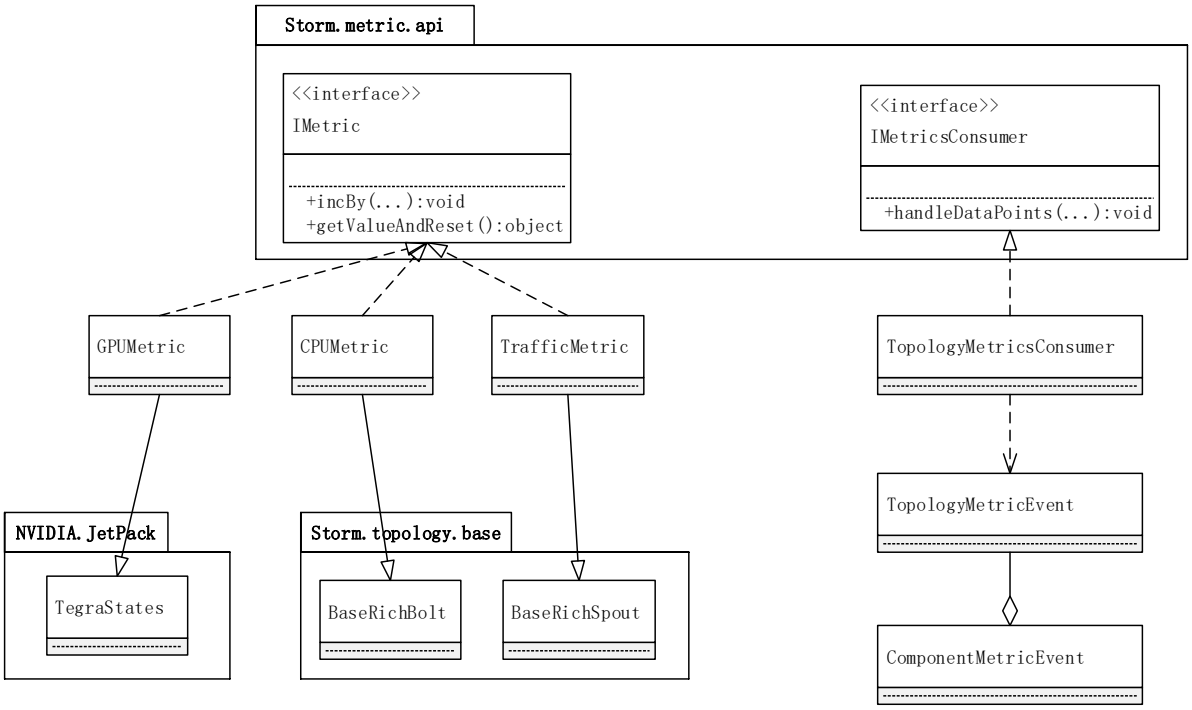

Fig. 1: The API model of the metric sensors

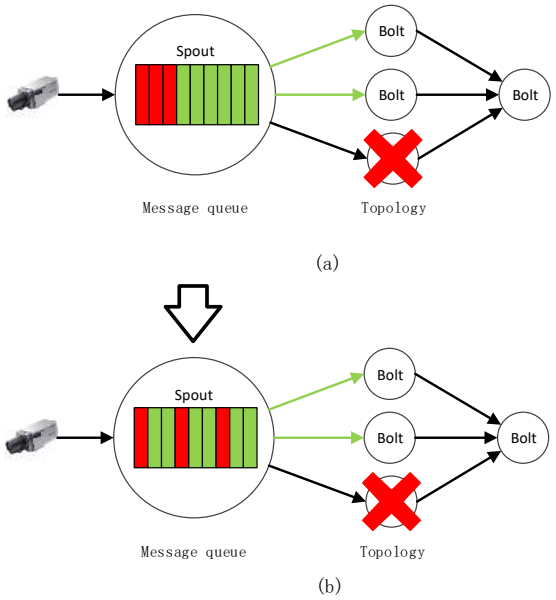

Fig. 2: A sample of decision execution
(a): a computing core is crashed. (b): our strategy to emit video frames in message queue. Red grids represents the blocked and dropped video frames, and green grids represents the video frames that will be emitted to other nodes.

are put into a queue and all computing nodes retrieve video frames from this queue in parallel. Because of a tiny time difference between different frames caused by network fluctuation or image complexity, there would be variance between the input order and the output order. For example, the input is f_1, f_2, f_3 and the output is r_2, r_1, r_3. The time stamp will help to correct this problem to have a result of r_1, r_2, r_3.

The SECP platform is designed for running deep learning algorithms, such as YOLO. The main computing of YOLO is convolutional computing, which occupies over 90% of the whole process. Therefore, accelerating convolutional computing can speed the running up. According to our former experiments [28], GPU is suitable for computation intensive tasks, which motivate us to use GPU-accelerated approach for improving the processing performance. The computing intensive portion (e.g. convolutional computing) can run a large number of GPU cores in parallel using CUDA[6]. The Jetson TX1 module has 256 CUDA cores. This is able to dramatically speed up computing applications. In addition, we installed a deep learning environment and deployed YOLO on each module, and made each module run independently but able to run in parallel to boost performance.

We further show the relation of these components as a package diagram in Figure 4. The monitor package contains three sub-package and uses the streaming processing framework (Storm) APIs to get metrics. The Storm stream processing framework depends on two packages directly. One is a video capture package that contains camera class and pre-processing class such as resize, and uses HIKVISION camera APIs to pull video stream. The second is the video processing package, which provides an intelligent processing interface and a traditional processing interface. These two interfaces are implemented by deep learning

6. https://developer.nvidia.com/cuda-zone

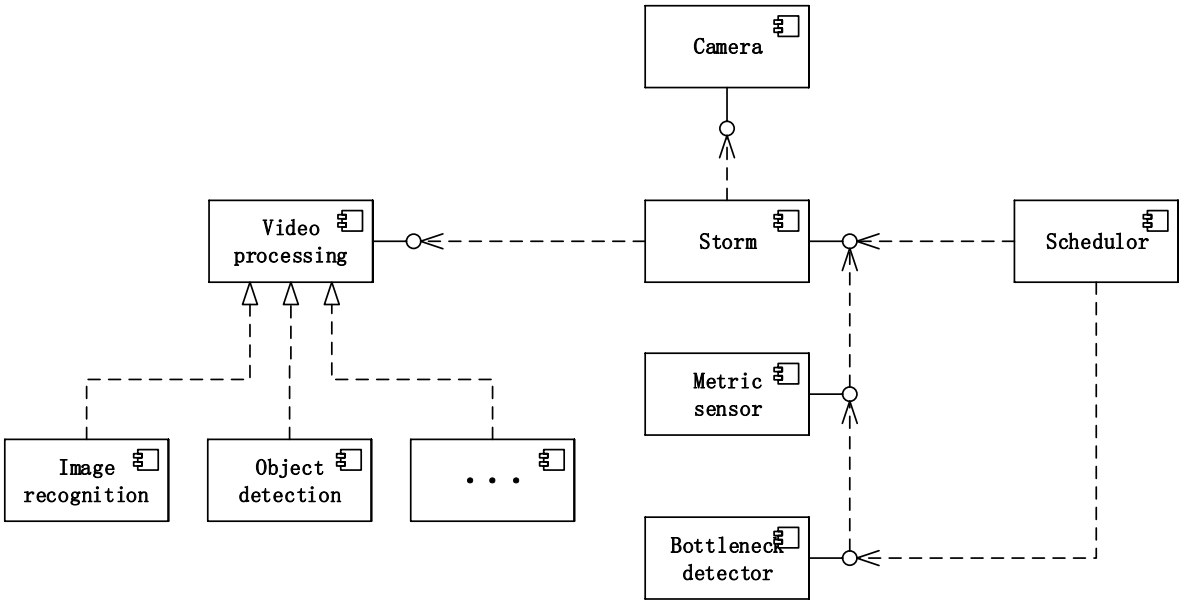

Fig. 3: Component diagram of overall architecture

algorithms, such as image recognition, and conventional algorithms such as canny edge detection. We have applied a set of toolkits on the algorithms including darknet[7] and OpenCV[8].

Figure 5 shows the deployment of the platform architecture. The platform has three computing cores and a switch. Each computing core has a GPU and a CPU cluster, on which we arrange different types of software, as shown in Table 3.

Supervisor3, Supervisor4 and Supervisor5 run object detection algorithms, so they are deployed on GPUs on three computing cores. Supervisor1, deployed on the second computing Core's CPU, merges results and stores them. Supervisor2 and Monitor are deployed on the other two computing core's CPUs respectively. The video capture node is an IP camera. All the nodes are connected with a switch to swap data.

TABLE 3: Software deployment

Computing core1	CPU	Supervisor2, Nimbus
	GPU	Supervisor3
Computing core2	CPU	Supervisor1
	GPU	Supervisor4
Computing core3	CPU	Monitor
	GPU	Supervisor5

3 Evaluation

In this section, we evaluate the performance of two types of SECPs built with Jetson TX1 and Jetson TX2. And we focus mainly on testing the scalability and fault tolerance of the platforms using YOLO V2. All the network models are trained on NVIDIA TITAN X in advance. The video stream is pulled from HIKVISION IP camera with a resolution of 1920×1080. The running environment is given in Table 4.

TABLE 4: Running environment

Software	Version
Storm	1.1.1
ZooKeeper	3.4.9
OpenCV	3.1.0
JDK	1.8
JetPack	3.1
CUDA	5.5

3.1 Jetson TX1 platform

In this experiment, we used eight Jetson TX1 cores to construct the platform. This is bigger than the proposed three cores platform because we aimed to explore the scalability and the bottleneck of the SECP. Three kinds of YOLOs with different network scales were running on the platform. Table 5 shows the properties of three kinds of YOLOs.

TABLE 5: Properties of three kinds of YOLOs

Name	Tiny	Medium	Large
Convolutional layer	9	15	24
Fully connected layer	3	3	3
Weight	172M	357M	1036M
Accuracy	57.1%	60.7%	63.4%

When the platform is started, all the computing cores are started in standby mode. Every computing core has one worker, which is in charge of running tasks. We add these running workers gradually and the three YOLOs' results are shown in Table 6, Table 7 and Table 8 respectively. The SECP platform in standby mode has a power consumption of approximate 20W, which is the running cost of basic system. Figure 6 shows the processing speed of three YOLOs. Obviously, it goes up continuously as the numbers of

7. https://pjreddie.com/darknet/
8. http://opencv.org/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2894621, IEEE Transactions on Cloud Computing

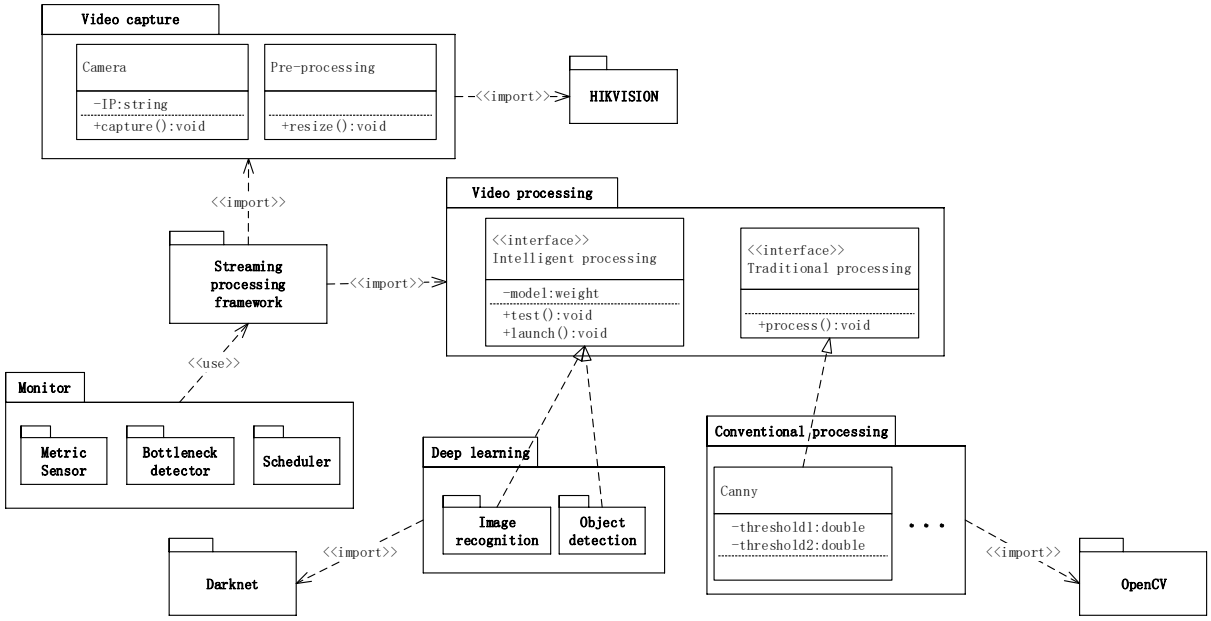

Fig. 4: Package diagram of overall architecture

workers increase, and Tiny YOLO is much faster than the other two YOLOs. We can also get the same conclusion from Figure 7 and Figure 8. However, in these three figures, Tiny YOLO changes more significantly than Medium YOLO and Large YOLO, and Medium YOLO's measurements are very similar with Large YOLO's. In addition, we found that the GPU utility was near 100% when running Medium YOLO. That is to say, this platform cannot run larger network models than Medium YOLO without latency. This experiment demonstrates that the platform has a high scalability, and that the bottleneck results from its computing capability.

3.2 Jetson TX2 platform

We applied three Jetson TX2 cores to construct SECP and ran Tiny YOLO on it. TX2 supports five kinds of running modes for various applications and we can configure it using NVIDIA's command tool called NVPModel. Table 9 lists the details of the configuration of five modes. This tool does not support hot switching, so we have to configure the platform before launching tasks. The results are shown in Table 10

Figure 9 and Figure 10 show the performance of the platform in five modes. Mode 0 had the highest processing speed because of its highest GPU frequency, but it also consumes the most energy, so its energy efficiency is not high. Mode 1 is the most power-saving mode, but the speed is a little bit low. Mode 3 has the highest energy efficiency with relative high processing speed, which is the best choice.

Another experiment on the Jetson TX2 platform evaluates fault tolerance using three time intervals. In the first 1-10 minutes, the platform runs normally. At the 11^{th} minute, core2 is shut down manually to simulate an accident. This condition lasts 10 minutes until the scheduler executes the repair strategy described in section 2.3.3 at 21^{th} minute. We configured Jetson TX2 to work in mode 0 and run Tiny YOLO on the SECP.

As shown in Table 11, in the first 10 minutes, the average latency to process a frame is 86.5ms, which means three cores can reach a processing speed of 35fps. When core2 is shut down, the average latency increased to 102.0ms due to decreased computing resources. After 10 minutes, the scheduler executes the repair strategy, bringing the average latency down to 75ms, but the SECP could process only 2/3 of the data of a normal situation.

We can analyze the changes of three cores by the four metrics (message queue, CPU utility, GPU utility, latency) that are shown in Figures 11, 12, 13 and 14 respectively. For normal status, the platform's processing speed is faster than the video frame pulling speed (30fps), so the message queue is empty. Core1 is the data transfer node which pulls video stream from the camera, decodes it and emits video frames to the other two cores' GPUs as well as to its own GPU, and these operations run on CPU. Therefore, core1's CPU utility is always the highest but its processing latency is always the lowest. When core2 is shut down, all the tasks including data swap, object detection, result merging etc. on it are moved to core3, causing core3 to have both high CPU utility and high GPU utility scores. In addition, core3's processing latency increases because two object detection tasks are competing for core3's computing resources. The reduced computing capacity causes video frames to block the message queue and the length of the queue would keep increasing if nothing is done to fix it. At the 20^{th} minute, when the scheduler executes the repair strategy, all core2's previous tasks on core3 are killed and the message queue is emptied. According to this strategy, the frame emitting method will be adjusted to make the platform run smoothly. Although the computing capacity became weaker due to the (simulated) hardware fault, the SECP platform recovers to a

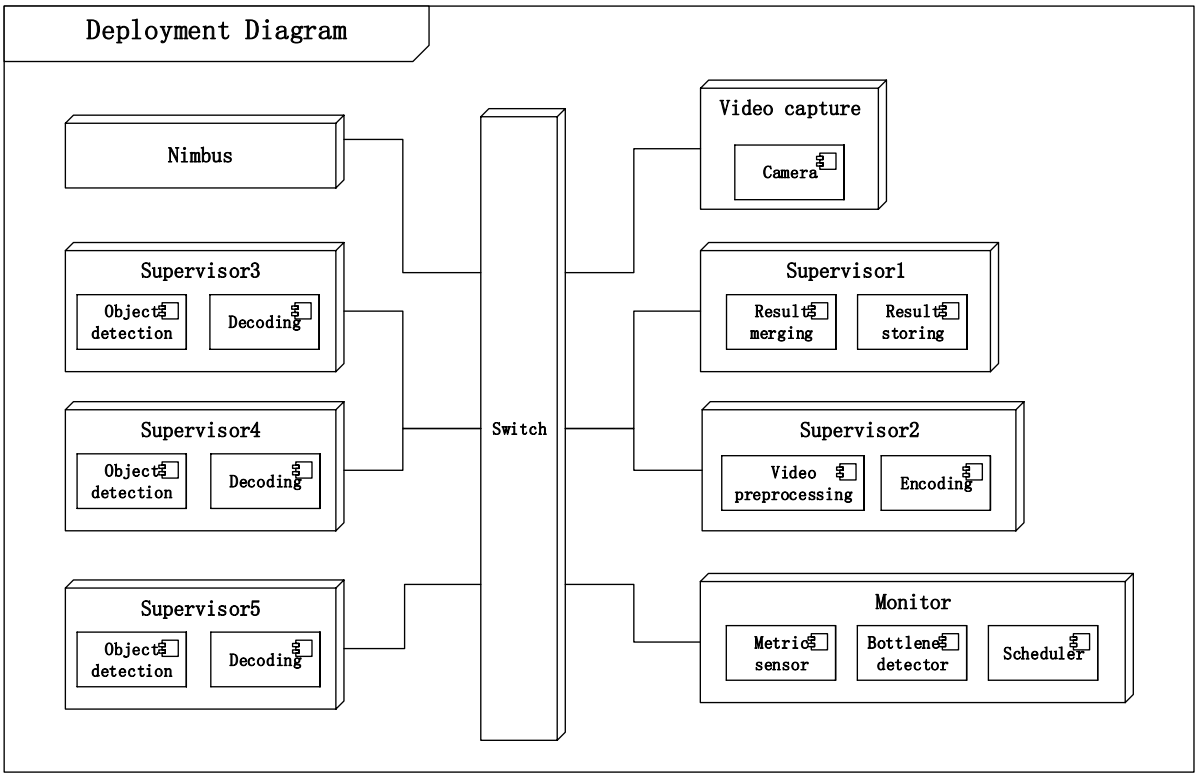

Fig. 5: Deployment diagram of overall architecture

TABLE 6: Performance of tiny YOLO on TX1 platform

Number of cores	0	1	2	3	4	5	6	7	8
Processing speed (fps)	0	11	23	34	45	57	67	79	90
Power consumption (W)	19.3	28.4	37.1	46.1	54.7	62.5	69.5	76.2	83.3
Network throughput (M/s)	0	6.81	12.99	19.16	24.74	30.47	35.63	39.64	45.17

TABLE 7: Performance of medium YOLO on TX1 platform

Number of cores	0	1	2	3	4	5	6	7	8
Processing speed (fps)	0	4	8	12	15	19	23	27	31
Power consumption (W)	20.5	30.0	39.9	50.2	61.2	71.7	83.8	94.7	104.8
Network throughput (M/s)	0	2.48	4.95	7.04	9.27	11.35	13.60	15.62	17.92

TABLE 8: Performance of large YOLO on TX1 platform

Number of cores	0	1	2	3	4	5	6	7	8
Processing speed (fps)	0	3	6	9	12	16	19	22	25
Power consumption (W)	21.0	30.5	40.5	50.8	61.9	73.3	84.9	96.0	107.3
Network throughput (M/s)	0	1.86	3.71	5.57	7.43	9.26	11.12	12.98	14.63

normal status.

In order to verify that TX1 and TX2 are efficient for object recognition, the YOLO v2 Tiny network and the VOC 2007 data set are used to test and compare the performance of the three platforms (including GTX Titan X). The specific results are shown in the following Figure 15:

As can be seen from Figure 15, although the performance of GTX TITAN X is very good, its single high energy consumption is very unsuitable for low power scenarios. TX1 and TX2 are more energy efficient with the same accuracy as GTX TITAN X, and TX2 is the best.

3.3 Discussion

In the experiment on the Jetson TX1 SECP, three kinds of YOLOs have different number of convolutional layers but the same number of fully connected layers. Because the main calculation overhead in CNNs comes from convolution, we modified only the convolutional layers of the YOLOs. In the experiment on the Jetson TX2 SECP, an abnormal status lasted 10 minutes until the scheduler executes repair strategy. This is too long for practical applications, but this was adopted on purpose for this experiment so that we could observe the results clearly. In practical applications,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2894621, IEEE Transactions on Cloud Computing

TABLE 9: NVPModel mode definition

Mode	Mode name	Denver2	Frequency	ARM A57	Frequency	GPU frequency
0	Max-N	2	2.0GHz	4	2.0GHz	1.30GHz
1	Max-Q	0	-	4	1.2GHz	0.85GHz
2	Max-P Core-All	2	1.4GHz	4	1.4GHz	1.12GHz
3	Max-P ARM	0	-	4	2.0GHz	1.12GHz
4	Max-P Denver	2	2.0GHz	0	-	1.12GHz

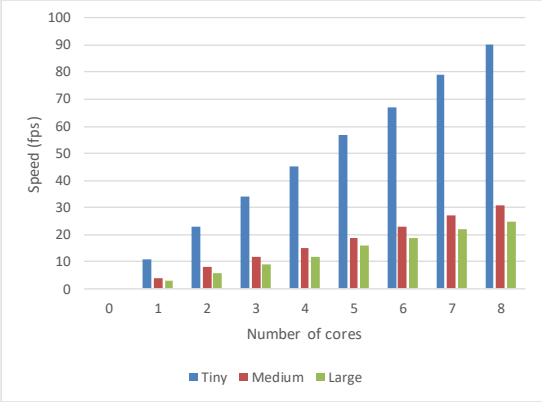

Fig. 6: Processing speed of three kinds of YOLOs

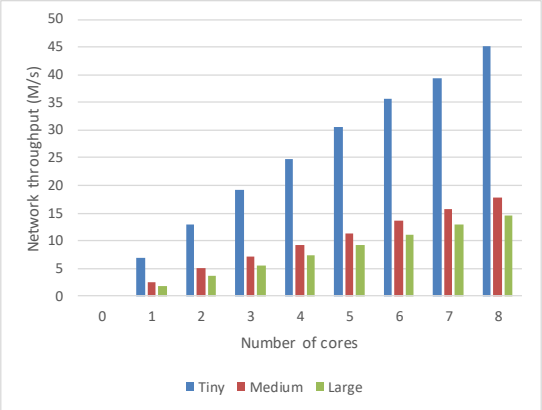

Fig. 8: Network throughput of three kinds of YOLOs

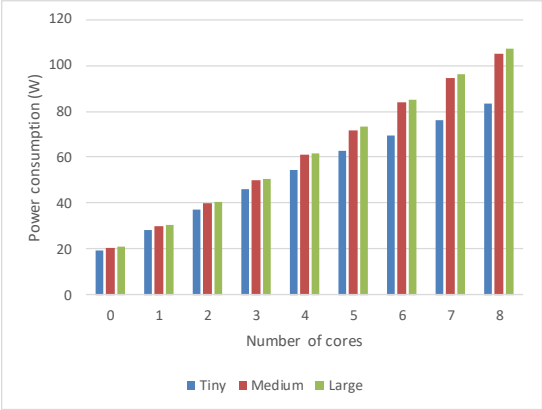

Fig. 7: Power consumption of three kinds of YOLOs

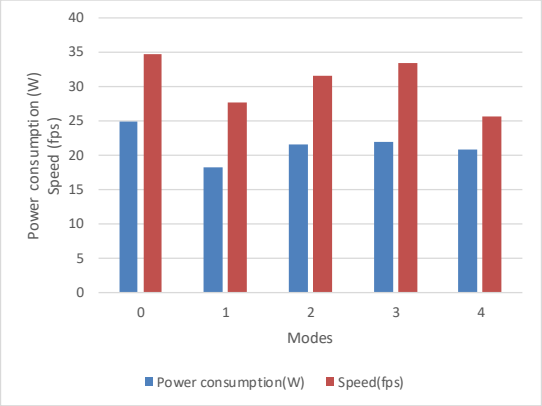

Fig. 9: Power consumption and speed in five modes

the scheduler can execute the strategy as soon as possible (i.e., within 10 seconds). We designed two kinds of system architectures for two main reasons: 1) we demonstrate the generalization ability of the platform because two kinds of platforms have the same software environment and both of them can run stably; 2) the experiment results can give some guidance in choosing a suitable platform in practical applications, since TX2 based platform is stronger but TX1 based platform is more economical.

In the SECP platform, only limited number of bolts is used to process video (one bolt on each node), and all these bolts are running in full capacity from our tests. Our results show the performance - speed, power consumption and network throughput - increases linearly with increasing module number, and the SECP platform has no bottleneck issue with 8 modules. Theoretically, the most likely cause of bottleneck is the network bandwidth. The last single bolt

TABLE 10: Performance of SECP in five modes

Mode	0	1	2	3	4
Power consumption (W)	25.0	18.2	21.5	21.9	20.8
Speed (fps)	35	28	32	33	26
Energy efficiency (fps/W)	1.39	1.52	1.47	1.52	1.23

TABLE 11: Overall performance of fault tolerance

Time interval (min)	1-10	11-20	21-30
Latency (ms/frame)	86.5	102.0	75.0
Data amount (frame)	18080	17640	12060

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2894621, IEEE Transactions on Cloud Computing

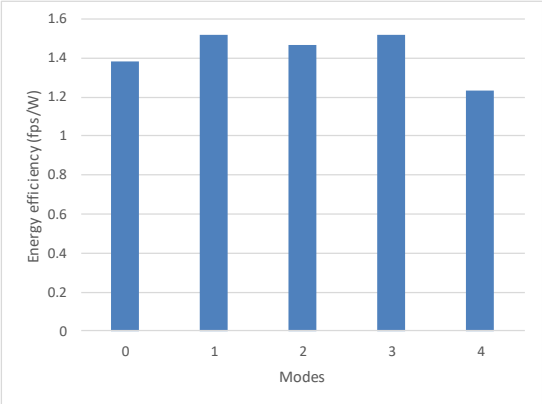

Fig. 10: Five modes' energy efficiency (fps/W)

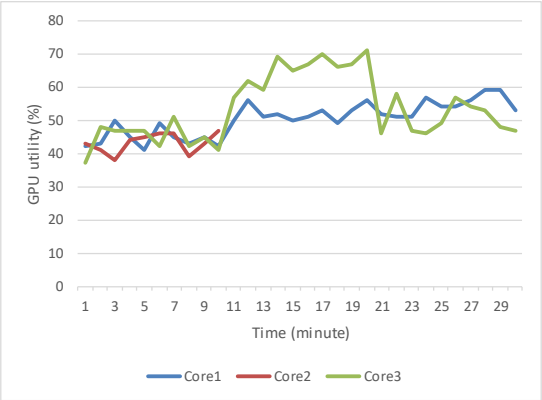

Fig. 13: GPU utility (%)

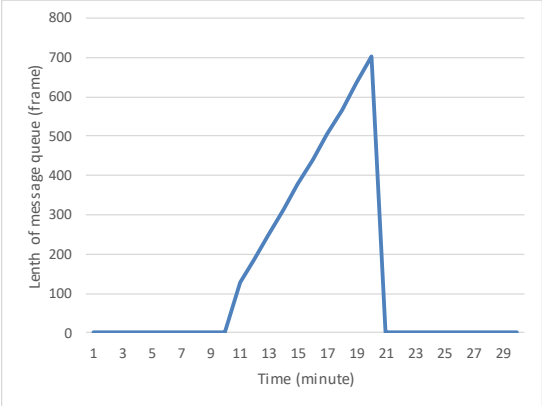

Fig. 11: Length of message queue (frame)

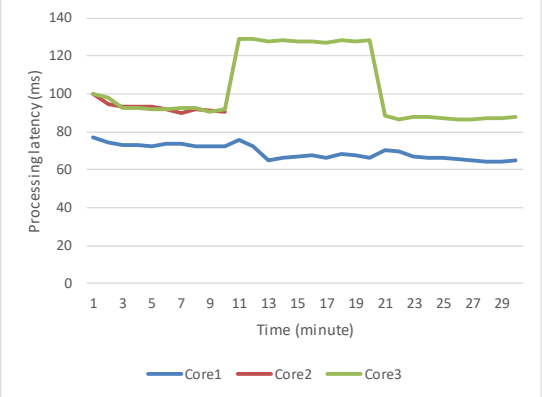

Fig. 14: Processing latency (ms)

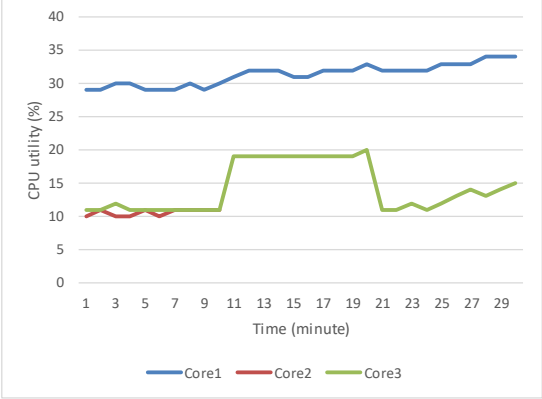

Fig. 12: CPU utility (%)

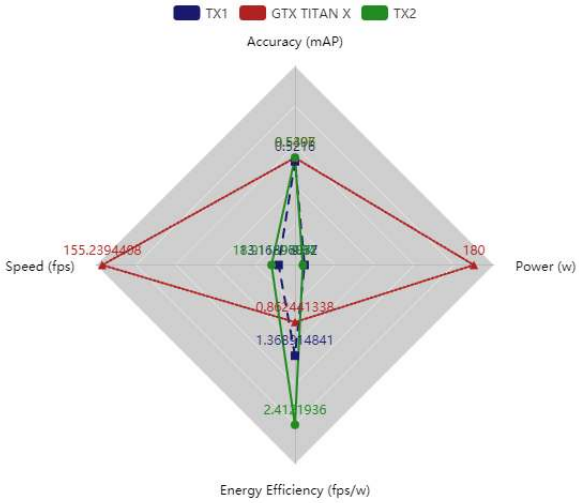

Fig. 15: Comparing TX1, TX2 and GTX Titan X

in the SECP is used to collect all the video frame information to organize the recognition results in the chronological order of the original video frames, where one single bolt is enough. The amount of calculation in this single bolt is very small (which takes much less than 1 ms). Therefore, the last single bolt will not be a bottleneck. In the real-world environment, SECP must determine the number of bolts doing parallel processing during deployment. Usually, we reserve a standby node for fault recovery. The SECP is monitoring the status of worker nodes and will reassign the

task to the standby node if a node is down.

In this research, a scale-out approach could be the right answer because we have many small nodes. In addition, object detection is not a simple task. It needs lots of memory and computation at each node. This redundant deployment of neural network models may add cost in higher power consumption, but it does not need to pass copies of the data, which involves additional network serialization and de-serialization.

There are various kinds of deep neural network architectures, such as CNNs, Recurrent Neural Networks (RNNs), and Deep Belief Networks (DBNs), and many kinds of input data (e.g. video, audio and text), but they are all essentially tensor calculation. In this work, we take object detection as an example to evaluate the performance of the platform, and the results show the platform can support CUDA well, which means the SECP platform can support a large variety of deep learning algorithms, and other computing intensive applications that require the corresponding software (e.g. CUDA). Our work shows that SECP can be a general platform for AI applications.

4 Related work

There is considerable research about embedded platforms for video processing. Ratnayake et al. [15] proposed a resource and power optimized FPGA-based configurable architecture for video object detection by integrating noise estimation, Mixture-of-Gaussian background modeling, motion detection and thresholding. Their Virtex-5 FPGA-based embedded platform achieved real-time processing of HD-1080p video streams at 30fps. However, their work was actually video background subtraction instead of object detection like YOLO as we used. Our platform can not only support conventional video processing algorithms such as background subtraction, but also support complex deep learning algorithms.

Bako et al. [16] presented an embedded implementation of a hardware-efficient method for motion information extraction from video signal, using an FPGA circuit-based system for on-line Sobel edge detection and edge displacement-based optical flow computation. The total execution time on a Xilinx Spartan-6 FPGA was approximately 1.6ms. This research only used a single embedded device to conduct Sobel edge detection, which is much simpler than the intelligent video processing like the object detection that we are doing.

Mao et al. [13] transferred and tailored deep neural networks to embedded devices by using an embedded Jetson TK1 CPU+GPU platform to conduct real-time object detection. They paid much attention to improving Fast R-CNN to accelerate the algorithm. However, the hardware they used was too weak to support real-time object detection and they did not use multiple TK1s to construct a platform as we did, and so they achieved a performance of only 1.85 fps. Our SECP is both more powerful and more flexible because it can not only run deep neural networks efficiently but also has high scalability and fault tolerance, making it suitable for practical applications.

Jagannathan et al. [14] introduced TI's low power TDA3x Soc which was based on a heterogeneous, scalable architecture to detect objects from images. They used HOG features and AdaBoost cascade classifier to detect objects, and CNN to classify the detected objects. Compared with our work, this research had two main drawbacks: 1) the object detection method was multi-step rather than end-to-end, and, 2) their single computing core was not powerful. The overall system performance was 15fps, but the input images were very low resolution (32×32). Our platform can solve HD-1080p video with the highest speed of 79 fps running on three TX2s.

Gao et al. [20] proposed a scalable distributed object detection framework based on an embedded manycore cluster architecture. The basic object detection algorithm was parallel process images by cascade classifier and local binary pattern operator. The framework was implemented on a Xilinx Zynq SoC and Adapteva Epiphany combined heterogeneous manycore platform named Parallella. The application of an embedded cluster made the object detection algorithm run in parallel, providing 7.8 times speedup over a dual-core ARM. However, they could only achieve 2 fps when solving 1920×1080 resolution images. In addition, they did not consider fault tolerance and stability of the system as we have done in this paper.

There is also work that took advantage of cell phones to build mobile clouds called Mobile Storm [29], in which the work flow of a real-time stream processing job was modeled, then decomposed into several tasks so that the job can be executed concurrently and in a distributed manner on multiple mobile devices. It was implemented on Android phones and conducted on real-time HD video processing applications. This work is very similar with ours as it deployed Storm on embedded devices for real-time video processing. In addition, it also has high scalability by adding worker nodes. The results showed that the cluster with 5 nodes could handle a high resolution video stream at 19 fps. However, this cluster was evaluated using only face detection.

Some work is targeting elastic management of cloud resources. For example, Aljawarneh et al. [30] optimized the performance of big spatial data queries on top of Apache Spark, which also supported advanced management functions including a self-adaptable load-balancing service to self-tune framework execution. A comprehensive survey is conducted for stream processing by Assuno et al. [31] for achieving efficient resource management decisions based on current load, especially for edge and cloud computing, where our proposed SECP can serve as edge nodes. These works motivate our design of the self-management services in order to improve the reliability of SECP.

Kang et al. [32] proposed NOSCOPE targeting querying large scale of videos. It can reduce the cost of neural network video analysis by up to three orders of magnitude via an inference-optimized model search. Its purpose is to scale to thousands of hours of video, possibly from thousands of data feeds, for the purposes of large-scale video classification, but real-time performance is not its target. It used powerful GPUs to achieve good performance, which is not possible for the embedded devices we are testing.

Han et al. [33] demonstrated that they can achieve even higher accuracy while significantly reducing the computation and the size of models. However, this approach is not flexible enough. Some domain specific models have

performed well and could be used directly on the proposed platform. It is unwise to rewrite every model using a compression way. We used a simple but effective method to find bottlenecks of our SECP, which calculates the mean and variance of each metric, rather than adopting complex methods as the work by Zhang et al. [34] and Chen et al. [35]. Because the Storm cluster is small, light-weight algorithms are best for achieving energy efficiency.

5 Conclusions and Future Work

Real-time video processing on embedded devices can be applied to many domains, including drones and smart cars. However, the limited resources of embedded devices make this a big challenge. The existing work usually use a single device without the support of cloud computing technologies. In this paper, we design a deep learning platform on embedded devices using stream processing cloud computing, which is called streaming embedded cloud platform, where NVIDIA Jetson TX1 and TX2 development boards are used, and Apache Storm is deployed as cloud computing environment for deep learning algorithms (Convolutional Neural Networks) to process video streams. Some self-managing services are designed to make sure the platform can run smoothly and stably. When facing accidents such as a computing core crash, a strategy can be executed to keep the platform running continuously with the least data loss.

We have evaluated the SECP platform in terms of processing speed, power consumption, and network throughput by running different kinds of deep learning algorithms for object detection. The results show the proposed platform can meet the requirement of real-time video processing with high scalability and fault tolerance, and is usable for running deep learning algorithms on embedded devices to realize real-time video processing.

In the future, we will pay more attention to video processing algorithm modification rather than using existing algorithms directly. We are considering designing algorithms specifically for the platform in order to take full advantage of computing resources to achieve higher speeds. Furthermore, it is advantageous that the SECP can communicate with cloud servers, where more tasks and decisions can be made.

Acknowledgments

This research is supported by the Program on Innovation Method Fund of China (Grant No. 2015010300), the Key Research Program of Shandong Province (No. 2017GGX10140) and also supported by Fundamental Research Funds for the Central Universities.

References

[1] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] C. D. Manning, "Computational linguistics and deep learning," *Computational Linguistics*, 2016.

[4] V. Kustikova and P. Druzhkov, "A survey of deep learning methods and software for image classification and object detection," *OGRW2014*, p. 5, 2014.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[9] L. Zhu, S. Tan, W. Zhang, Y. Wang, and X. Xu, "Validation of pervasive cloud task migration with colored petri net," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 89–101, 2016.

[10] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[11] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[12] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[13] H. Mao, S. Yao, T. Tang, B. Li, J. Yao, and Y. Wang, "Towards real-time object detection on embedded systems," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[14] S. Jagannathan, K. Desappan, P. Swami, M. Mathew, S. Nagori, K. Chitnis, Y. Marathe, D. Poddar, and S. Narayanan, "Efficient object detection and classification on low power embedded systems," in *Consumer Electronics (ICCE), 2017 IEEE International Conference on*. IEEE, 2017, pp. 233–234.

[15] K. Ratnayake and A. Amer, "Embedded architecture for noise-adaptive video object detection using parameter-compressed background modeling," *Journal of Real-Time Image Processing*, vol. 13, no. 2, pp. 397–414, 2017.

[16] L. Bako, S. Hajdu, S.-T. Brassai, F. Morgan, and C. Enachescu, "Embedded implementation of a real-time motion estimation method in video sequences," *Procedia Technology*, vol. 22, pp. 897–904, 2016.

[17] A. Nikitakis, S. Papaioannou, and I. Papaefstathiou, "A novel low-power embedded object recognition system working at multi-frames per second," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 1s, p. 33, 2013.

[18] X. Chen, W. Li-Feng, and M. Qing-Lei, "A video sensing oriented format-compliant entropy coding encryption scheme and embedded video processing system," *DEStech Transactions on Computer Science and Engineering*, no. aice-ncs, 2016.

[19] G. Arva and T. Fryza, "Embedded video processing on raspberry pi," in *Radioelektronika (RADIOELEKTRONIKA), 2017 27th International Conference*. IEEE, 2017, pp. 1–4.

[20] F. Gao, Z. Huang, S. Wang, and X. Ji, "A scalable object detection framework based on embedded manycore cluster," in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2016 International Conference on*. IEEE, 2016, pp. 142–145.

[21] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Region-based convolutional networks for accurate object detection and segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 1, pp. 142–158, 2016.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," in *European Conference on Computer Vision*. Springer, 2014, pp. 346–361.

[23] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[24] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[25] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the*

IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 779–788.
[26] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.
[27] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.
[28] W. Zhang, P. Duan, W. Gong, Q. Lu, and S. Yang, "A load-aware pluggable cloud framework for real-time video processing," *IEEE Trans. Industrial Informatics*, vol. 12, no. 6, pp. 2166–2176, 2016. [Online]. Available: https://doi.org/10.1109/TII.2016.2560802
[29] Q. Ning, C.-A. Chen, R. Stoleru, and C. Chen, "Mobile storm: Distributed real-time stream processing for mobile clouds," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*. IEEE, 2015, pp. 139–145.
[30] I. M. Aljawarneh, P. Bellavista, A. Corradi, R. Montanari, L. Foschini, and A. Zanotti, "Efficient spark-based framework for big geospatial data query processing and analysis," in *2017 IEEE Symposium on Computers and Communications, ISCC 2017, Heraklion, Greece, July 3-6, 2017*, 2017, pp. 851–856. [Online]. Available: https://doi.org/10.1109/ISCC.2017.8024633
[31] M. D. de Assuno, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1 – 17, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804517303971
[32] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: Optimizing neural network queries over video at scale," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1586–1597, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137664
[33] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
[34] W. Zhang, P. Duan, L. T. Yang, F. Xia, Z. Li, Q. Lu, W. Gong, and S. Yang, "Resource requests prediction in the cloud computing environment with a deep belief network," *Software: Practice and Experience*, vol. 47, no. 3, pp. 473–488, 2017.
[35] S. Chen, M. Ghorbani, Y. Wang, P. Bogdan, and M. Pedram, "Trace-based analysis and prediction of cloud computing user behavior using the fractal modeling technique," in *Big Data (BigData Congress), 2014 IEEE International Congress on*. IEEE, 2014, pp. 733–739.

PLACE PHOTO HERE

Weishan Zhang Weishan Zhang is a full professor, deputy head for research of Department of Software Engineering, China University of Petroleum. He got PhD from Northwestern Polytechnical University, China in 2001. His current research interests are big data platforms, pervasive cloud computing, and service oriented computing. Weishan has published over 100 papers and his current H-index according to Google scholar is 16.

Dehai Zhao Dehai Zhao has obtained master degree at China University of Petroleum, majoring in software engineering. Now he is studying at Australian National University for Ph.D. His main research interests include deep learning, software engineering and big data processing, and he has published over 10 papers in these fields.

Haoyun Sun Haoyun Sun is a master student working on big data processing, computer vision, and software engineering. He has published two papers in these areas.

Yi Guo Yi Guo is a master student working on big data processing platforms, and software engineering. She has published two papers in these areas.

PLACE PHOTO HERE

Xin Liu Xin Liu is an associate professor at College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China. She received her PhD from Nankai University, China in 2012. Her research interests include social networks, data mining, big data processing.

Jiehan Zhou Jiehan Zhou is currently a senior researcher in University of Oulu. He got his PhD in computer engineering from University of Oulu, Finland and PhD in Manufacturing Automation from Huazhong University of Science and Technology, China. His current research interests are big data platforms, Internet of Things, pervasive cloud computing, He has published over 100 papers.

Su Yang Su Yang is a professor with Dept. of Computer Science and Engineering, Fudan University. His research interests are mainly pattern recognition, social computing, machine vision, and data mining. He is the PI of a number of NSFC projects including "Graphical Symbol recognition in natural scenes" and "Detection of abnormal collective behaviors via movement and communication pattern analysis".

PLACE PHOTO HERE

Huansheng Ning Huansheng Ning received a B.S. degree from Anhui University in 1996 and Ph.D. degree in Beihang University in 2001. Now, he is a professor and vice dean of School of Computer and Communication Engineering, University of Science and Technology Beijing, China. His current research focuses on Internet of Things, cyber-physical modeling. He is the founder of Cyberspace and Cybermatics and Cyberspace International Science and Technology Cooperation Base. He serves as an associate editor of IEEE System Journal. He gained the IEEE Computer Society Meritorious Service Award in 2013, IEEE Computer Society Golden Core Award in 2014.