A Pattern-Based API for Mapping Applications to a Hierarchy of Multi-Core Devices

Jia Guo
Computer Science and Engineering
Ohio State University
Columbus OH 43210
Email: guo.980@osu.edu

Radu Teodorescu
Computer Science and Engineering
Ohio State University
Columbus OH 43210
Email: teodores@cse.ohio-state.edu

Gagan Agrawal
Computer and Cyber Sciences
Augusta University
Augusta GA 30912
Email: gagrawal@augusta.edu

***Abstract*— Recent years have witnessed an evolution of Internet of Things (IoT) devices. This has lead to the emergence of (related) paradigms of Edge/Fog computing, where the goal is to exploit the power of interconnected heterogeneous devices together with distributed/cloud computing. In Edge/Fog computing, one of the challenges is automatically distributing the work between different devices to reduce application latency. At the same time, with increasing transistor density and the end of Denard scaling, even small edge devices have parallelism. Thus, we need a programming model that can help distribute the work between different devices and yet parallelize operations on each device.**

Motivated by the popularity of MapReduce(-like) frameworks, we develop a pattern-based high-level programming API targeting computer vision applications for the Edge/Fog paradigm with parallelism within devices. Based on this API, parallelization, workload distribution, and optimizations that account for resource limitations of IoT devices, are implemented. Our evaluation with three image processing applications shows that while using a single device, we achieve 17-45% speedup over OpenCV, one of the most popular frameworks for image processing. In addition, we further gain benefits from distributing the work between multiple devices.

I. Introduction

This work is motivated by two popular trends. First, with increasing transistor density and the end of Dennard scaling, parallelism has become extremely common. While standard desktops have been multi-core for more than a decade now, Smartphones like IPhones and Android phones commonly have 4 and 8 cores, respectively. In recent years, small *edge devices* like a Raspberry Pi or NXP's i.MX 8M Nano also have 4 cores. For example, Raspberry Pi 3 B, a single-board computer, has quad-core 64-bit ARM v8 processor, which is able to complete complicated tasks such as robot control on its own. Clearly, exploiting such parallelism is crucial for applications that have substantial computing and/or data processing requirements.

The second trend is towards Internet of Things (IoT) and emergence of applications that target such an environment. With the rapid development of smart devices and wireless technologies in recent years, paradigms of *Edge* or *Fog* computing have emerged. The idea is to use the edge to cloud continuum of processing devices for applications on IoT. Bonomi, *et al.* [5] and Iorga *et al.* [15] give a detailed description of the structure and applications of Edge/Fog computing. Some of the key considerations can be summarized as:

- **Heterogeneity**: The devices in Edge/Fog computing are of very different processing capabilities, ranging from small sensors to cloud resources.
- **Hierarchical Structure**: Hierarchical structure is supported in Edge/Fog environment, different layers from edge to center function as a continuum, and multiple edge devices are likely to connect to a single more powerful device (See Figure 1).
- **Predominance of Wireless Access**: At the edge of the network, sensors and devices are mostly connected by a wireless network.
- **Need for Low Latency**: Many applications executed on such environments are interactive and require (near) real-time response. In general, we can say that reducing the latency of response is an important consideration.

Overall, development of applications for the Edge/Fog computing paradigm is presenting new challenges. With edge devices like a Raspberry Pi having substantial computing power, instead of using such devices only for collection and forwarding of information, it is more reasonable to fully utilize Pi's computing power and schedule a certain fraction of the workload at the edge of network. At the same time, it is important to exploit parallelism within each device or node. Because scheduling of operations may not occur until the runtime, application stages should be written so that they can be efficiently parallelized on different types of devices. As stated above, the range of processing devices available differ considerably in their architecture and size of resources.

Though there has been several efforts on developing programming APIs for edge/fog paradigms [14], [27], [31], the work does not consider automatic partitioning of work between the devices, or parallelization of processing within a device. In this paper, we focus on developing a pattern-based high-level programming API for the Edge/Fog environment and applications we characterized above. The pattern based API is a generalization of what has been very successfully done with MapReduce and its implementation in frameworks like Hadoop and Spark [36]. Our work builds on top of a variant of MapReduce [16], [33], which has much better efficiency and lower memory requirements. Moreover, unlike previous work with MapReduce/Spark, in this paper, our focus is on image processing applications. In these problems,

978-1-7281-6095-5/20/$31.00 ©2020 IEEE
DOI 10.1109/CCGrid49817.2020.00-92

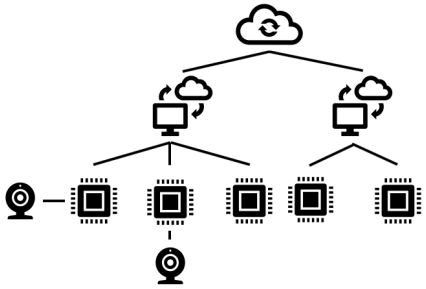


Fig. 1: Topology of a Hierarchical IoT System

a stream of images are collected from edge devices and processed in a distributed fashion in an hierarchical structure of devices. One common use case involves *smart cameras* connected to a local PC and/or a cloud server. The smart camera itself can have substantial computing power, however, the processing power of the devices closer to the edge of network tends to be lower than those at the center. Thus, the challenge is to design an API that enables parallelism on different devices, splitting of work across devices, and optimizations that involve knowledge of both application and specific type of system.

The contributions of this work are:

- API Design: We propose a set of high-level pattern-based APIs to describe image processing and reduction operations, which enables parallelization and optimization of applications on different platforms.
- Parallelization of Operations: We are able to parallelize our target applications on both an edge device (Raspberry Pi) and a normal multi-core desktop/node.
- Optimizations for Specific Devices: We optimize the processing considering the resources on the device. For example, caching of intermediate applications can help improve performance, but such caching has to be cognizant of difference in memory availability between devices of different types.
- Distribution of Work Between Devices: By recognizing how certain operations are independent of each other, processing is automatically split between edge devices and a normal PC.

Using our framework, we have implemented three popular image processing applications. Some of the key results from our detailed evaluation are as follows. First, when comparing against OpenCV, one of the most popular image processing frameworks, we obtain a speedup of 17-45%, while parallelizing the same code on different devices. Second, we further improve performance by over 20% by dividing the processing between edge devices and a central device, as compared to simply processing all data at the central device. It should also be noted that we are able to parallelize the same code on devices with different hardware architectures and computing capabilities.

II. API Design

As stated in the previous section, our goal is to have a simplified programming API for distributed environments where the work is to be distributed across devices with very different capabilities, with all or most of them having parallelism. We particularly build on top of success of frameworks like Smart [33] and Spark [36], which allow complex applications to be specified using high-level and simple primitives. However, unlike these works, we are focusing on image processing applications common in IoT scenarios, and we want to obtain high performance as well. Towards the latter goal, we use OpenMP and Posix Sockets at the back-end to support efficient implementation of functions. Additionally, Intel SSE and ARM Neon are also used to optimize the code in several places.

In this section, we describe the API we have developed. Table I describes the set of functions supported in our framework. They can be viewed as a set of operators applied to an image/data point. Broadly, one can think of the processing as a series of operators applied one after another to an image or a (set of) pixels. Data parallelism is enabled within the execution of each operator, and task/pipelined parallelism is enabled when one strong device (e.g. PC) is processing intermediate results concurrently from multiple child devices.

The first set of functions are based on the MapReduce idea. However, for better performance and lower memory requirements, we use a variant. Specifically, unlike functional idea of MapReduce, a reduction structure is provided for the users to implement MapReduce-like processing. This structure is first introduced by Jiang *et al.* [16], and improved by Wang *et al.* [33] as a C++ based MapReduce alternate. In our previous works, the implementations of this idea have been shown to be efficient, both in terms of execution time and memory. Specifically, a comparison with Spark has shown up to $92\times$ speedup [33]. Following work [13] shows that this major improvement comes from reduced intermediate result size and improved locality, as is crucial for small devices. In the image processing scenario, it can be applied to several application stages, as we will discuss in the next section.

This reduction-style processing structure comprises three functions: *genKey*, *accumulate*, and *merge*, working on a *reduction object* class. A *reduction object* is essentially a user-defined accumulator for intermediate results to merge on. For example, in histogram application, it is defined as a counter for the votes associated with a specific bin(s). As for the three functions, Listing 1 illustrates the implementation of a histogram-like reduction procedure using this API. In the *genKeys* stage (line 3-8), runtime takes *ids[]* as *keys* to access all the histogram bins related to an input data chunk. Then in the *accumulate* stage (line 11-40), each pixel votes to corresponding bins in histogram. Finally, all histograms from different threads are collected and merged to one final output with the *merge* function (line 36-39). At the back-end, the framework maintains a hash-map of keys to reduction objects. The process of reduction works as following. For each element of input, a key (or multiple keys) is generated by *genKey*, based on which the element is accumulated to corresponding accumulator(s). The *genKey-accumulate* process happens in parallel for input elements. After all the inputs are processed, the accumulators of different threads are merged to a final result. Two additional functions *truncate*

$\boldsymbol{V}$: general data $\boldsymbol{M}$: image matrix $\boldsymbol{C}$: co-coordinates $\boldsymbol{K}$: key $\boldsymbol{Seq < A >}$: a list of elements of type A $\boldsymbol{f}$: functions defined in framework $\boldsymbol{Acc < K >}$: Accumulator of tuple with the same key $\boldsymbol{< A, B >}$: a tuple of type A and B		
genKey	$V \rightarrow K \quad (Seq < K >)$	Generate one (or multiple) key(s) K for each input element V
accumulate	$< K, V > \times Acc < K > \rightarrow Acc < K >$	Accumulate an input pair $< K, V >$ to the accumulator $Acc < K >$ corresponding to K
merge	$Acc_1 < K > \times Acc_2 < K > \rightarrow Acc_1 < K >$	Merge two accumulators with the same K to the first accumulator $Acc_1 < K >$
filter	$V \rightarrow Bool$	Generate a boolean to decide whether a given input element V should be filtered
count	$Seq < V > \rightarrow Long$	Count the number of data elements in a sequence $Seq < V >$
transform	$V \rightarrow V'$	Perform flexible user-defined operation to transform one element to another element
sample	$M \times double \rightarrow M$	Sample an image M with a sampling factor *double*
window	$M \times f \times Seq < C > \rightarrow V$	Perform operation f on a sequence of windows defined by $Seq < C >$ in image M
pyramid	$M \times sample \times Seq < double > \rightarrow Seq < M >$	Generate a pyramid representation for M based on the $sample$ function and a sequence of factors
convolution	$M \times C \rightarrow V$	Perform a convolution on the given coordinate C in image M

TABLE I: Functional Description of the API

and *emission* are also provided so that the user can define how the input should be divided to elements and when the reduction result can be emitted. This reduction process can happen in local device or extended to the distributed version.

The *count* and *filter* are operators provided simply to count the number of data points and filter them out, respectively. The *transform* operator transforms the images/data points sequentially into another data type. Users can define flexible procedures such as SVM classification process in this operator. *Sample* is an operator to down-sample images by a given factor.

One non-trivial operator we introduce is the *window* operator, where the goal is to enable parallelism for sliding window operations with caching capability. Specifically, to use this operator on an image, a sequence of window coordinates and the operations to apply on the windows needs to be defined. In implementing this, the framework will concurrently invoke the operations on a *set* of windows. This set can either comprise a single queue of windows, which are to be processed concurrently (with synchronizing barriers); or could comprise several queues where windows in the same queue are strictly processed sequentially. These two patterns enable flexible scheduling of operations that are based on sliding windows, while enabling potential optimizations that will be discussed later.

Another non-trivial operator is Pyramid, which takes an instance of sample operator and a list of down-scaling factors. These down-scaling factors are used to produce a sequence of down-scaled images, which can be processed in parallel. This operator meets the need of popular image processing applications, as we will describe in the next section. Our implementation also uses this framework to divide the work between different types of devices.

The last set of operations is as follows. The convolution operator is used to apply a predefined convolution kernel on an image or inside a window. The communication between devices in our framework follows an *edge to center* pattern. Two functions, upstream and broadcast, are defined to push intermediate images/data points to the parent device or broadcast necessary configurations in Json format to the children devices. A key observation in the IoT topology we defined in I is that the most powerful devices and user interfaces are in the center of the topology; therefore, merge and emission of results almost always happens at the center or parent. This implies a unidirectional propagation of results and a top-down broadcast of control messages, which can be described by these two functions.

III. Applications

In this section, we introduce three applications that our framework supports and optimizes for the IoT scenario. They are all object detection algorithms, though they all use different types of features: Haar-like features [17], [26], HOG (histogram of gradient) features [8], and LBP (local binary pattern) [23], respectively. Our goal here is to show how the APIs introduced in the last section can support these applications.

In each of these three applications, the first few steps are quite similar. Periodically, raw images are retrieved from either the camera modules or local image files. These images are sampled and converted to *gray-scale* format. In our framework, these steps are simple implementation of filter and transform functions. After these steps, a multi-scale down sampling is performed, which will be elaborated later in Section IV-A. On top of the above pre-processing, a sliding window detector is applied to detect the target objects in different parts of the image. This sliding window detector, which is essentially a pre-trained 0/1 classifier inspecting each window (or, region of interest, *ROI*) of $height \times width$ in the original image, consists of two components: *feature extraction* and *classification*. In the following discussion, we will first give a brief description of three feature extraction techniques associated with the applications and their implementation in our framework. Towards the end of this section, we show how the classification step is implemented.

A. Feature Extraction

Feature extraction is the process of converting an ROI to lower dimensional vectors. The three features we mentioned above, Haar-like, LBP, and HOG, are widely used for object detection, as they are available through popular libraries like OpenCV, Scikit-Image, and others. The advantages of using these features are that they are easy to train with relatively small amount of data and the inference process is not resource-intensive either. Thus, besides being very

popular for object detection, they are also a good choice for edge devices.

Haar-like: Haar-like feature extraction is developed by Papageorgiou *et al.* [26], and enhanced by Jones *et al.* [17]. This feature extraction algorithm performs convolution in a given ROI using Haar-like features, which are a set of rectangular convolution kernels (shown in Figure 2). The evaluation of these convolutions are accelerated by a 2-D *integral image*, with which the sum of pixels in a rectangular area $(x1, x2] \times (y1, y2]$ can be calculated conveniently with the equation

$$Sum = I[x2, y2] - I[x1, y2] - I[x2, y1] + I[x1, y1]$$

Then the convolution can be easily evaluated by multiplying corresponding weights to the summation of pixels in the kernel-specified rectangular areas.

The implementation of integral image calculation and convolution operation in our framework is though transformation API and convolution functions, respectively. The transform API converts the original image to integral images. The weight in each Haar-like kernel and the convolution process is defined using the convolution API. One of the key aspects of our work is that we facilitate result reuse across adjacent windows, as will be discussed in Section IV-B.

HOG: The HOG feature extraction is proposed by Dalal, *et al* [8]. To calculate the HOG features of an ROI, we first calculate the X and Y gradient of each pixel as

$$d_x = p[x+1, y] - p[x-1, y]$$
$$d_y = p[x, y+1] - p[x, y-1]$$

with which we can get the magnitude and direction of the gradient of each pixel by

$$|d| = \sqrt{d_x^2 + d_y^2}$$
$$\theta = \frac{d_x}{\sqrt{d_x^2 + d_y^2}}$$

Then for each (non overlapping) $8 \times 8px$ cell, we calculate from all 64 pixels a histogram as shown in the bottom right of Figure 3, where the direction of gradient is split into 9 bins, and the vote of each pixel is decided by corresponding magnitude of gradient. On top of this, all neighbouring 2×2

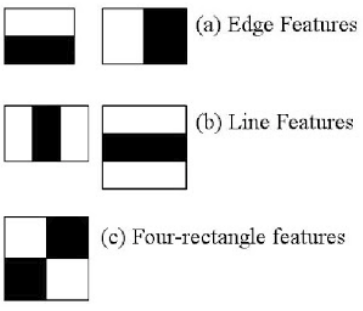


Fig. 2: Haar-like features

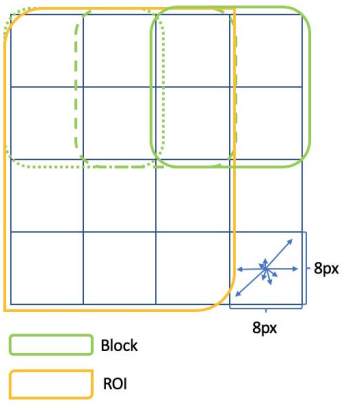


Fig. 3: HOG feature cells, blocks, and ROI

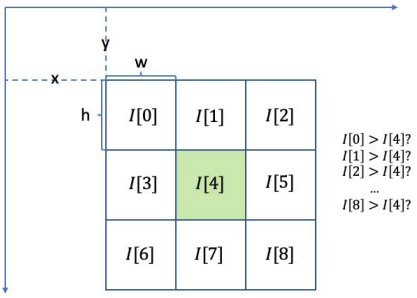


Fig. 4: MB-LBP feature illustration

cells are grouped as blocks (two adjacent blocks have 50% overlap), as shown in Figure 3). In each block, 4 histograms of cells are concatenated and normalized as the block feature. In the end, all the features of blocks whose top left cells lie in the given ROI are concatenated as the final ROI-level feature for classification.

A normal implementation of this procedure contains 4 nested loops. But with appropriate loop unrolling and caching, it can be implemented in our framework using two reduction structures. As shown in the high level code Listing 1, the first reduction structure (line 1-40) concurrently calculates the gradient in different cells, during which the direction and magnitude of gradient result will be instantly accumulated to the corresponding histogram of each block (line 15-30). In the second reduction structure (line 42-45), the normalized feature vector for each block is calculated and cached based on the result of previous reduction. The final HOG feature for each window can be achieved directly by concatenating the cached block-level normalized vectors, as described in the *windowClassifier* (line 48-65).

LBP: The LBP extraction is proposed by Ojala, *et al.* [23], and improved several times. A popular version, Multi-block Local Binary Patterns (MB-LBP), is a set of features described by quadruples, (x, y, w, h). Each quadruple specifies 9 rectangles in the ROI as shown in Figure 4. Using the integral image, we can easily calculate the sum of pixels in each rectangle, after which the sums of the surrounding 8

rectangles are compared with the sum of the central one. The comparison result (1/0) can be concatenated to a 8 bit integer, e.g. 10010111, which is the LBP feature. The implementation of LBP feature extraction is similar to Haar feature extraction, even simpler is that no convolution is involved.

B. Classification

When the features are extracted from ROI, they are input to the classification algorithms where the final output is generated. The classification algorithm frequently used together with Haar-like features and LBP is called *cascades of boosted classifiers*, with resulting applications called Haar-cascade and LBP-cascade. The concept of this type of classifier is that instead of using all the Haar/LBP features in the ROI at once, we divide them into an ordered series of tests (or stages). In each test, a chosen subset of the features are evaluated to determine if the given ROI should be excluded from the result. In other words, only when all tests are passed we accept the ROI as a target object. Since the early tests are designed to reject most of the negative data points, this classification technique enables early rejection of ROIs, avoiding unnecessary evaluation of all features in an ROI.

This classification process is implemented as a tree-shaped control sequence based on sliding window API in our framework. For HOG-based classification, we use a trained SVM model as the classifier, which has a even simple implementation of vector multiplications.

Listing 1: HOG feature extraction with our API

```
class BlockHistogram : public Reduction {
  //Generate the Block IDs as keys.
  void gen_keys(Data& cell, vector<int>& keys) override {
    int ids[4] = {-1,-1,-1,-1};
    //Pixels in each cell contribute to up to four blocks
    calculateBlockIDs(cell.x, cell.y, ids);
    for(int i=0;i<4;i++) keys.push_back(ids[i]);
  }

  // Accumulate pixels in a cell on histogram.
  void accumulate(Data& cell, vector<tuple<int ,RedObj
      ↪ *>>& histograms) override {
    //This loop can be optimized with SIMD instructions
    for(int y = cell.y; x<cell.yend; y++){
      for(int x = cell.x; x<cell.xend; x++){
        //Evaluate the derivatives
        float dx,dy;
        dx=mat[x+1][y]-mat[x-1][y];
        dy=mat[x][y+1]-mat[x][y-1];
        //Calculate bin IDs and votes
        int bin1=angleCeil(dx,dy);
        int bin2=angleFloor(dx,dy);
        //One pixel votes for two bins
        float vote1=splitVote(dx,dy,bin1)
        float vote2=splitVote(dx,dy,bin2)
        //Accumulate the results to histogram
        for(auto& h: histograms){
            h.second()->getBin(x,y)[bin1]+=vote1
            h.second()->getBin(x,y)[bin2]+=vote2
            ...//detailed interpolation process and pixel
                ↪  weight omitted
        }
      }
    }
  }

  // Merge histogram2 into histogram1 on bin[].
  void merge(unique_ptr<RedObj>& histogram1, const RedObj
      ↪ & histogram2) override {
    for (int i = 0; i < histogram1.NUM_BINS; ++i)
      histogram1->bin[i] += histogram2->bin[i];
  }
}

class BlockNormalization : public Reduction {
    ...
    //This reduction process takes care of normalizing
        ↪ the block-level histograms caching the result
}


class windowClassifier: Operation{
  //This operation is called in window() to run sliding
      ↪ window classification
  ...
  void execute(Param& param){
    int iBlock=param.cordinate.x;
    int jBlock=param.cordinate.y;
    double feature[] = new double[param.FEATURE_LEN];

    //fill in feature with cached results of blocks
    int count=0;
    for (;iBlock<..;iBlock++)
      for(;jBlock<...;jBlock++){
        param.blockCache.get(i,j,feature+count);
        count+=param.BLOCK_FEATURE_LEN;
      }
    if(svm(feature)) param.out.put(param.cordinate);
  }
}
```

IV. Optimizations Implemented in the Framework

In this section, we propose two significant optimizations for image processing applications that target the IoT environment. The goal behind these two methods is to enhance the application performance and resource utilization when devices with different processing power co-exist in the topology, as described in I.

A. Load Balancing Multi-scale Detection

Our first optimization involves achieving better performance through workload distribution for *multi-scale* object detection algorithms, considering the topology that has several edge devices and either a personal computer (or cloud or cloudlet). There are two reasons why we will like to distribute the work between the devices. First, even small edge devices, such as a Raspberry Pi, have a significant amount of of processing power. For example, in our environment, 6 Raspberry Pis' total computing power is 40% of the computing power of a family PC equipped with i7-6700K.

The second reason for workload distribution is that more processing the edge reduces the amount of data that has to be transferred from edge devices to the PC. Data transmission can be a significant source of battery consumption at edge devices, which can be a critical factor when edge devices are not connected to power.

The workload distribution itself needs to consider multiple factors such as resource utilization, power consumption, and latency of operations. Current implementations of the object-detection algorithms have been ported to edge devices. However, to the best of our knowledge, there is no implementation that divides the work between the devices for the PC-edge environment. With the goal of utilizing the resources on different types of devices, we focus on work distribution. Because we are dealing with real-time image processing jobs, minimizing the latency of processing is the target for our workload distribution strategy.

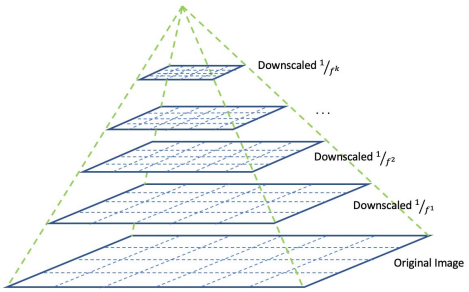


Fig. 5: Pyramid representation of an image

To enable a balanced and easy-to-implement distribution of the workload, we can take advantage of the multi-scale detection structure in the common image processing applications, which is exposed through our API. Specifically, the object detection solutions we discussed above are expected to find out all objects of interest in a given image, regardless of the position or size. To do this, a sliding window of detection must be applied on each image from a set of images. This set of images, called pyramid (representation), comprises the original image and images geometrically down-scaled from the original image, as shown in Figure 5.

Two observations can be made. First, each scaled image in the pyramid can be processed independently, in other words, task parallelism has been exposed through our API. Second, the number of windows (all of fixed size) that can be applied to an image goes down exponentially as the image scales down. This gives us the convenience of dividing the multi-scale object-detection workload to devices in different layers of the topology. As the processing power of central PC is much higher than those of the edge devices, by assigning images of larger scales to PC, we can leave a reasonable amount of work on edge, enhancing both the performance and resource utility. Specifically, our implementation is as follows. Once the image is retrieved from the camera, certain fast pre-processing steps, e.g. equalization and gray-scale conversion, are applied on the edge device, after which a copy of the image is sent to the PC. Now, in parallel, the image is down-sampled and processed on both PC and Raspberry Pi, according to some pre-calculated split of scales. For example, we are down scaling a image of $568 \times 320px$ with a factor of $\frac{1}{1.1}$, yielding a sequence of images in the size of 1 (original), $\frac{1}{1.1}$, $\frac{1}{1.1}^2$, $\frac{1}{1.1}^3$, and so on, up to $\frac{1}{1.1}^{24}$. Say, in a 2-layer topology of PC and Raspberry Pis, one PC is running as the back-end of 6 Pis, giving an roughly 1:2 processing power ratio for one Raspberry Pi to the resource serving it on PC end. Based on this estimation, we can put the images of scales of 1 (original), $\frac{1}{1.1}$, $\frac{1}{1.1}^2$, and up to $\frac{1}{1.1}^{10}$ on the PC, and $\frac{1}{1.1}^{11}$ to $\frac{1}{1.1}^{24}$ on the Raspberry Pi. This gives a roughly 2.5:1 ratio for load on PC to the load on 6 Raspberry Pis. By doing this, the job on Raspberry Pi and PC will finish at about the same time, giving most balanced scheduling of the workload. In a more complicated topology, this division of workload can be done automatically after a pre-run benchmark of processing power at a different level.

This division of workload is done at the runtime. By providing a series of (down) sampling factors to the distributed pyramid API, the framework will automatically decide a reasonable the split of workload between edge devices and PC. This can be achieved either with user-defined processing power ratio or the from result of a previous load test using the same application. Additionally, on top of this fixed load-balancing module, a dynamic adjustment of workload can be implemented by monitoring the job finish time on different devices. In addition to load balancing multi-scale detection jobs, we also observe similar pyramid structure in fused-layer CNN [1], as is a potential generalization of this optimization.

B. Caching of Intermediate Results

The second optimization is proposed specifically to enhance the performance when convolution operations are performed in sliding windows. As we have described in III-A, the convolution kernels in Haar-like features have perfect rectangle boundaries, and the weight in each rectangle are the same. Thus, the result of one particular convolution kernel varies only by a small amount when the window slides one step in a particular direction. As shown in Figure 6 (left), for horizontally oriented kernels, convolution result in red circles stay the same for horizontally adjacent windows. An analogous property holds for vertically oriented kernels as shown in Figure 6 (middle). Diagonally shaped kernels can always be divided to two horizontal/vertical kernels as shown in Figure 6 (right).

With this observation, we can build a cache to reuse the results in adjacent steps. First, row by row and column by column integral images, IR and IC, are introduced to facilitate the calculation of new steps. More importantly, for each non-diagonal Haar-like kernel, we cache the entire convolution result at window $(x-1, y)$ as $Conv_{x-1,y}$ and by-step convolution results $S_{(x-1)\%w,y}$, as is shown in Figure 7. For horizontal oriented features, the result for the same kernel in the next step, $Conv_{x,y}$, can be calculated from $Conv_{x-1,y}$ by using the following expression:

$$Conv_{x-1,y} - S_{(x-1)\%w,y} + w1 \times (IC_{y_1 y_4}) + w2 \times (IC_{y_2 y_3})$$

In this part, SIMD instructions are used to calculate $IC_{x_1 x_4}$ and $IC_{x_2 x_3}$ in advance for adjacent windows.

Vertical features can be calculated similarly from $W[x, y-1]$. The management of this cache and the reuse of results are implemented behind the window operation and convolution kernel functions. Moreover, to use this caching mechanism, the sliding windows are executed in a wave-front style, i.e., in a sequence of (0,0), (1,0) (0,1), (2,0) (1,1) (0,2), and so on. This sequence makes sure that when (x, y) is executed, the results for $(x-1, y)$ and $(x, y-1)$ are available.

The LBP algorithm could also take advantage of this optimization. As discussed in III-A, the calculation of the MP-LBP feature is still essentially calculation of sum of rectangle areas in sliding windows. Therefore, by simply caching 9 sums and by-step sums for each feature can we deduce the results similarly for the same feature in the next window.

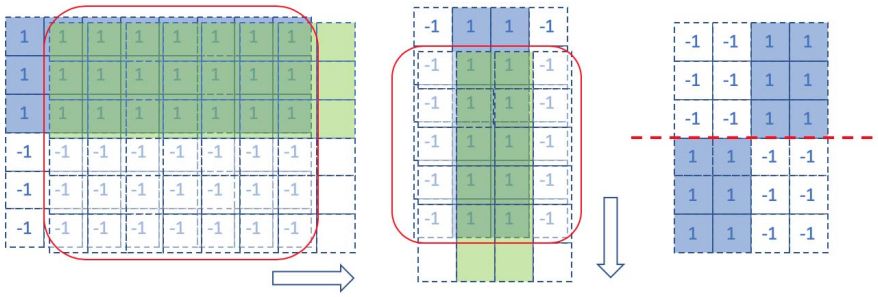


Fig. 6: Result Reuse in Adjacent Windows

On top of this, we can further improve the performance of this caching by limiting the number of features to be stored. This is because first, on edge devices, the resources such as cache and RAM are limited in size and speed compared to standard PC and thus they may not be sufficient to support the entire cache. Second, according to III-B, the cascaded classification uses an early rejection of ROIs. As the result, for most of the windows, merely first few stages are frequently executed. With this observation, we can limit the caching to first few stages so that little resource will be spent on caching features less likely to be used.

V. Experimental Results

In this section we evaluate the performance of our framework. Our first set of experiments focus on evaluating the overall performance and scalability of our framework on two different platforms. For these experiments, our baseline is OpenCV [18], a widely used C++ based library for computer vision. OpenCV has implemented these algorithms in standalone mode with TBB/OpenMP for parallelism support, which is used for mapping to multiple cores. Our next set of experiments focus on evaluating the impact of optimizations, including the ability to partition the work between edge and central devices.

A. Experiment Setup and Applications

Our experiments are conducted using two different platforms. The first is a 16 core multi-core machine with 3.0Ghz Intel(R) Xeon(R) Platinum 8000 series processors and over 90GB memory, that we accessed on the Amazon Elastic Compute Cloud (Amazon EC2). The second platform is an IoT computing setup with one common desktop and six Raspberry Pi devices. The desktop has one Intel(R) i7-6700k 4-core CPU running at 4.0Hz, with 24GB RAM and 1000Mbps Ethernet connection to a wireless router. The router is an AC2600 4x4 Dual Band Wi-Fi router with gigabit Ethernet connection. Each of the Raspberry Pi has a Quad core 64-bit processor clocked at 1.2GHz, 1GB LPDDR2 SRAM, and 802.11n Wireless LAN connection to the router. Each of Raspberry Pis only communicates with the desktop, forming a hierarchical structure described in I.

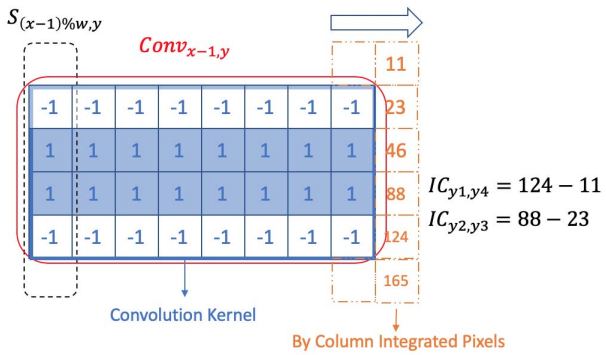


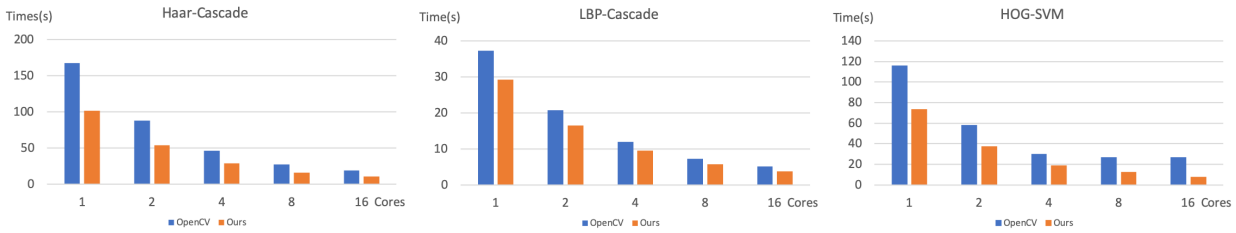
Fig. 7: Caching for Haar Features

We compiled all test applications with OpenCV 4.1.2, GCC version 7.4, and GOMP on Ubuntu 18.04 (desktop), latest Raspbian (Raspberry Pi), and Deep Learning AMI Version 25.3 (AWS).

The three applications we are using are those described in III. Each involves gray-scale conversion followed by object detection with Haar-like, LBP, and HOG features, respectively. While performing object detection, the pre-trained models are from OpenCV's haarcascade_frontalface_default.xml, lbpcascade_frontalface.xml, and hog.cpp, respectively. The first two algorithms are for frontal face detection with cascaded classifiers, and the third one is for people detection with SVM. To allow repeatable experiments, we use test data with 1000 images preprocessed to $480 \times 320px$ from the IMDB-WIKI data set [29]. With the goal of repeatable experiments and comparison across different configurations and frameworks, we didn't take input from Pi's camera module directly. Specifically, for cascaded detectors, the number of stages executed in each ROI heavily depends on the image itself, as described in III-B, and thus, the workload is dependent upon the set of images processed.

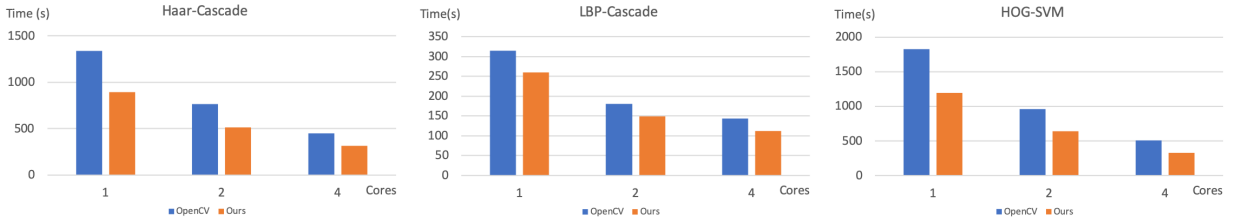
B. Performance Comparison

First, we test the scalability of OpenCV and our framework on a 16-core multi-core and Raspberry Pi. As we can see in Figure 8, the applications using our framework scales well as the number of cores increases on the multi-core machine. On one core, our implementation of Haar-Cascade, LBP-Cascade, and HOG-SVM is faster by 39%, 21%, and 37% percent over OpenCV. The main reason for this performance difference is we leverage the *reduction object* programming model and the optimizations in Section IV-B



(a) Haar-Cascade (b) LBP-Cascade (c) HOG-SVM

Fig. 8: Comparing Scalability of Our Framework with OpenCV on AWS $Time(s)/Cores$



(a) Haar-Cascade (b) LBP-Cascade (c) HOG-SVM

Fig. 9: Comparing Scalability of Our Framework with OpenCV on Raspberry-Pi $Time(s)/Cores$

In scaling from 1 core to 16 cores, each time the number of cores doubles, the performance gain of our implementations are $\{1.89\times, 1.87\times, 1.75\times, 1.52\times\}$, and $\{1.76\times, 1.74\times, 1.66\times, 1.49\times\}$, respectively, for Haar-Cascade and LBP-Cascade. Overall relative speedups with 16 cores are 9.46 and 7.57 for these two applications. The performance gain of OpenCV's Haar-Cascade and LBP-Cascade on 2-8 cores falls into the range of $1.64\times$ to $1.9\times$, showing similar relative speedups as our framework. For these two applications, in scaling from 8 to 16 cores, OpenCV's performance gain is around $1.42\times$, lower than our implementation. Overall, relative speedups with 16 cores are 8.61 and 7.22 for these applications with OpenCV.

A more distinct trends is seen with HOG-SVM. In scaling from 1 to 4 cores, relative speedups with our framework and OpenCV are 3.88 and 3.90. However, in scaling HOG-SVM from 4 to 16 cores, we see a much more obvious average performance gain of $(1.55\times)$ using our framework, compared with OpenCV's $1.05\times$. This gain is from the efficient reduction structure we are using to implement the evaluation of histogram. Since the histogram process is very widely used in various feature extraction and histogram equalization algorithms, the *reduction object* paradigm supported in our current framework (and previously introduced in MATE and Smart frameworks [16], [33]) can be very helpful in developing image processing applications. In comparison, OpenCV implements histogram with intermediate results including partial derivatives, angle, and magnitude matrices buffered in memory, which requires additional space and reduces locality. More broadly, previous work has demonstrated the advantage of using this reduction object paradigm in clustering, online training, and other types of applications; therefore, generalization of our framework in similar IoT applications is promising.

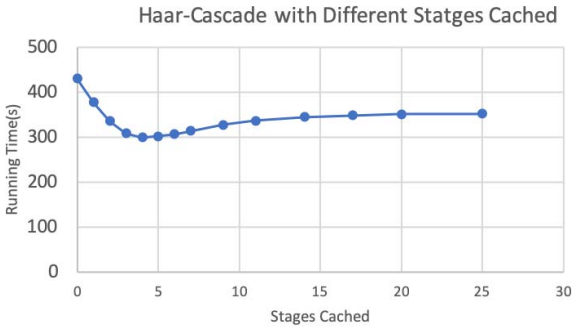


Fig. 10: Running Time of Haar-Cascade with Different Number of Stages Cached in Cascade Classifier

We also performed similar scalability test on Raspberry Pi. As shown in Figure 9, on one core, our implementation is faster by 34%, 17%, and 35% percent over OpenCV, respectively, for these three applications. Compared with the performance on the multi-core server, the performance improvement for Haar-Cascade and LBP-Cascade is lower. The reason is that we are caching some results in the optimizations for these two algorithms. With less cache size, cache speed, and memory performance, the optimizations will not execute as efficiently as on desktops. In scaling from 1 to 4 cores, our implementation and OpenCV has similar performance. For the three applications, the speed up is $\{2.84\times, 2.31\times, 3.66\times\}$ for our framework and $\{2.99\times, 2.18\times, 3.58\times\}$ for OpenCV.

Figure 10 shows the how the performance of Haar-like

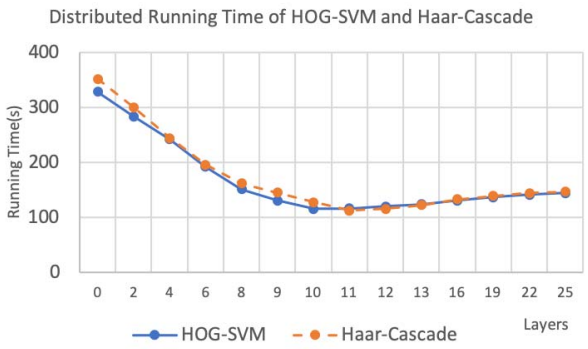


Fig. 11: Running Time for Distributed Application with Different Layers of Images Processed on Desktop

algorithm varies on the edge device when the number of stages to to be cached are varied. As we have discussed in IV-B, edge devices tend to have limited resources, whereas the later stages of cascaded classifiers are not reached in most detection tasks. Therefore, the number of stages in which the framework will cache the intermediate results for adjacent windows are limited, which will also control the peak size of intermediate results. From it is obvious that caching only first 4 of 25 stages yields the best performance, with around 15% gain as compared to caching all stages (25 stages cached). This corresponds to a 33% gain when compared with OpenCV's implementation. When more information is cached, the running time increases due to limited cache size and memory performance on Raspberry Pi.

Finally, Figure 11 illustrates the ability of our framework to distribute workload between the edge and the central devices. To enable setup for this experiment, we compared the performance of our framework between 4 core execution on a single Raspberry Pi with 4 core execution on desktop. The running time for Haar-Cascade, LBP-Cascade, and HOG, respectively, is $\{22.14s, 7.70s, 23.21s\}$ on PC, and $\{351.71s, 117.83s, 325.50s\}$ on Raspberry Pi. This result shows that the performance on desktop for all three applications are around 15 times better than Raspberry Pi. Thus, for dividing up the processing between edge devices and the central device, the central device can be seen has having 2.5 times higher processing power than 6 edge devices.

In the load balancing experiment, we run Haar-Cascade and HOG-SVM in a distributed fashion on the desktop and Raspberry Pi. Specifically, the desktop processes the bottom (more compute-intensive) layers (see Figure 5) of the pyramids from the images collected on each of the 6 Raspberry Pis. As we can see, when more bottom layers are assigned to the desktop the processing time decrease dramatically. For the Haar-Cascade algorithm, after the bottom 11 layers are assigned to desktop, the time begins to increase slowly, as the desktop is more heavily loaded. Similarly the minimum processing time of HOG-SVM happens when 10 images are assigned to desktop. This confirms to the layer assignment estimation we gave in Section IV-A. When processing a large number of images, our framework can decide on the load distribution automatically.

VI. Related Work

There has only been limited and relatively preliminary work on programming model development for Edge/Fog computing. Authors in [27] develop a distributed framework named MediaBroker for live stream management and data transformations. The main focus of their work is the type-aware data transport and the system for describing types of streaming data. Hong *et al.* [14] proposes a high level programming model called Mobile Fog that targets a large number of distributed heterogeneous devices. Each Mobile Fog process in their model handles the workload from a certain Geospatial region and it supports a dynamic distribution of heavy workload within the same network hierarchy. However, they didn't consider workload optimization between different levels of network hierarchy due to their different computation abilities. Satyanarayanan *et al.* propose a decentralized cloud computing architecture called GigaSight [31] to support edge video analytics using virtual machine-based cloudlets. The main motivation of their work is to address the issues of a huge amount of video streams sent to cloud with strong enforcement of privacy preferences. Although a multi-step pipeline is considered during video analytics, the automated deployment of those steps is not their focus. In summary, we are not aware of previous work on Edge/Fog settings that considers either of image processing application, parallelization on devices with different capabilities, and/or automatic distribution of work between edge and central devices.

Parallelization of image processing applications has been extensively studied in the past. Computer vision pioneer Rosenfeld described parallel image processing approach for a cellular processor in 1983 [28]. In the decade that followed, cluster (or network) of workstations became more popular target [20]. More recently, CUDA has been used for parallel image processing [35]. A survey of algorithms developed over several decades can be seen from Braun *et al.* [7] and Uhr [32]. OpenCV has lately become a popular framework for image processing that includes support for parallelization [18]. OpenCV has been used in the past on Raspberry Pi for image processing [6].

The idea of using patterns or *skeletons* for developing parallel applications has been used in the past [11], [12], [22]. Our work is specific to image processing applications, shared memory parallelization (including on edge devices) and distribution of work between edge and central devices.

There is a large body of work on task parallel programming models [2], [9], [21], [24], [25]. The task parallelism exploited in our framework is specific to the pyramid structure in image processing applications and distributed of work between edge and central devices. As we evolve our framework and consider other application classes, ideas from more general task frameworks can be used.

Similarly, there exists a large body of work on scheduling DAG-described tasks on multi-processor environments [4], [10], [19], [34]. These algorithms have been designed to optimize the task scheduling problems given computation power and bandwidth limits. Specifically, [19] proposed a static algorithm, Dynamical Critical-Path Scheduling, to schedule

tasks on fully connected identical processors. [34] provided a more compact static algorithm based on topological sort. [30] demonstrated a hybrid algorithm for heterogeneous processors. More recently, with the emergence of battery-powered edge devices, the energy concern is brought to this picture. One published approach [3] is energy aware and it could dynamically scale down voltage on devices to bring down both computation power and energy consumption. Our work is different in starting with a program written in pattern-based API (as opposed to a task graph), and combining task and data parallelism. However, scheduling of tasks is limited because of the applications and target platform we consider.

VII. Conclusions

Internet of Things (IoT) and associated applications are going to have a large impact on society and businesses. Developing applications for this emerging paradigm involves new challenges. Specifically, we have argued how we need to exploit data and task/pipelined parallelism for these applications. Focusing on a specific class of applications (computer vision), we have developed a pattern-based API that help develop applications for mapping to these platforms. Our results have shown that we can effectively parallelize and scale across cores on both edge and central devices, outperforming popular vision framework OpenCV in each case. We are also able to reduce latency by dividing the work between edge and central devices.

References

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 22. IEEE Press, 2016.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3] S. Baskiyar and R. Abdel-Kader. Energy aware dag scheduling on heterogeneous systems. *Cluster Computing*, 13(4):373–383, 2010.

[4] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 27–34. IEEE, 2010.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[6] S. Brahmbhatt. *Embedded Computer Vision: Running OpenCV Programs on the Raspberry Pi*, pages 201–218. Apress, Berkeley, CA, 2013.

[7] T. Bräunl, S. Feyrer, W. Rapf, and M. Reinhardt. *Parallel image processing*. Springer Science & Business Media, 2013.

[8] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. 2005.

[9] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *2008 37th International Conference on Parallel Processing*, pages 586–593. IEEE, 2008.

[10] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, Technical report, 2006.

[11] W.-c. Feng, H. Lin, T. Scogland, and J. Zhang. Opencl and the 13 dwarfs: a work in progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 291–294. ACM, 2012.

[12] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[13] J. Guo and G. Agrawal. Achieving performance and programmability for mapreduce (-like) frameworks. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 314–323. IEEE, 2018.

[14] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, pages 15–20, 2013.

[15] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi. Fog computing conceptual model. Technical report, 2018.

[16] W. Jiang, V. Ravi, and G. Agrawal. A Map-Reduce System with an Alternate API for Multi-Core Environments. In *Proceedings of Conference on Cluster Computing and Grid (CCGRID)*, 2010.

[17] M. Jones and P. Viola. Fast multi-view face detection. *Mitsubishi Electric Research Lab TR-20003-96*, 3(14):2, 2003.

[18] A. Kaehler and G. Bradski. *Learning OpenCV*. O'Reilly Media, Inc., 2014.

[19] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, 7(5):506–521, 1996.

[20] C.-k. Lee and M. Hamdi. Parallel image processing applications on a network of workstations. *Parallel Computing*, 21(1):137–160, 1995.

[21] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Acm Sigplan Notices*, volume 44, pages 227–242. ACM, 2009.

[22] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, 2002.

[23] T. Ojala, M. Pietikäinen, and D. Harwood. A comparative study of texture measures with classification based on featured distributions. *Pattern recognition*, 29(1):51–59, 1996.

[24] S. L. Olivier, B. R. De Supinski, M. Schulz, and J. F. Prins. Characterizing and mitigating work time inflation in task parallel programs. *Scientific Programming*, 21(3-4):123–136, 2013.

[25] S. L. Olivier and J. F. Prins. Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, 38(5-6):341–360, 2010.

[26] C. P. Papageorgiou, M. Oren, and T. Poggio. A general framework for object detection. In *Sixth International Conference on Computer Vision (IEEE Cat. No. 98CH36271)*, pages 555–562. IEEE, 1998.

[27] U. Ramachandran, M. Modahl, I. Bagrak, M. Wolenetz, D. J. Lillethun, B. Liu, J. Kim, P. W. Hutto, and R. Jain. Mediabroker: A pervasive computing infrastructure for adaptive transformation and sharing of stream data. *Pervasive and Mobile Computing*, 1(2):257–276, 2005.

[28] A. Rosenfeld. Parallel image processing using cellular arrays. *Computer*, (1):14–20, 1983.

[29] R. Rothe, R. Timofte, and L. V. Gool. Imdb-wiki 500k face images with age and gender labels.

[30] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111. IEEE, 2004.

[31] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.

[32] L. Uhr. *Parallel computer vision*. Elsevier, 2014.

[33] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang. Smart: A mapreduce-like framework for in-situ scientific analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 51. ACM, 2015.

[34] M.-Y. Wu, W. Shu, and J. Gu. Efficient local search far dag scheduling. *IEEE Transactions on parallel and distributed systems*, 12(6):617–627, 2001.

[35] Z. Yang, Y. Zhu, and Y. Pu. Parallel image processing based on cuda. In *2008 International Conference on Computer Science and Software Engineering*, volume 3, pages 198–201. IEEE, 2008.

[36] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.