OSP Final Project:
Implementing Automatic Document Scanning using OpenCV

Haeyoon Cho°

Computer Science & Engineering, Ewha Womans University

요 약

This project aims to provide automatic document scanning using OpenCV library. When provided an image which contains the document, the document will be detected and be saved in either grayscale or binary as if it is scanned, without distortion. The overall process will be explained in this report, together with the sample output image.

1. Problem to Solve

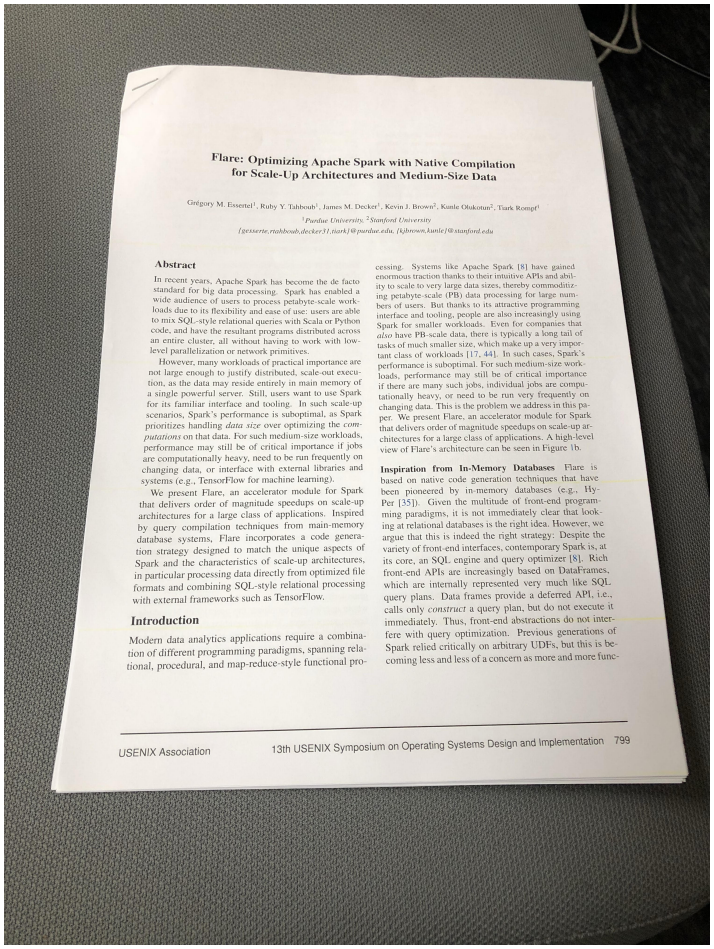
The goal of the project is to implement a program that performs automatic document scanning, where the user can choose saving option for the output image, either in grayscale or binary. There are some requirements in order to do this correctly. We should properly estimate the document boundary, and transform the image without distortion. To enhance the visibility, contrast adjustment is also recommended. In this report, I am going to explain the overall process to realize the goal with the functions that I used. Note that the functions are either implemented on my own or provided by OpenCV [1]. For explaination with intermediate output images, I am going to use the image in Figure 1.

2. Implementation

I used iPhone8 to take the picture of the document, and resized it to certain ratio in order to reduce time in performing the calculations.

2.1 Edge Detection

To detect the document boundary, edge detection has to take place beforehand. For this, I applied Canny edge detection technique, using the OpenCV-provided function: `Canny` [2]. But before that, we need some pre-processing to get the desired output. First, since Canny edge detection works well on grayscale image, we have to convert it to grayscale. Moreover, Canny edge detector is quite sensitive to noise, which makes it hard to find the desired output edges. So we have to apply filtering to smooth the image. I used Gaussian filtering technique in this case, adjusting the kernel for boundary processing. Finally we are ready to apply the `Canny` function to the grayscale, blurred image. This is very simple with just a single line of code like in Listing 1.



Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data

Grégory M. Essertel[1], Ruby Y. Tahboub[1], James M. Decker[1], Kevin J. Brown[2], Kunle Olukotun[2], Tiark Rompf[1]

[1]Purdue University, [2]Stanford University

{gesserte,rtahboub,decker31,tiark}@purdue.edu, {kjbrown,kunle}@stanford.edu

Abstract

In recent years, Apache Spark has become the de facto standard for big data processing. Spark has enabled a wide audience of users to process petabyte-scale workloads due to its flexibility and ease of use: users are able to mix SQL-style relational queries with Scala or Python code, and have the resultant programs distributed across an entire cluster, all without having to work with low-level parallelization or network primitives.

However, many workloads of practical importance are not large enough to justify distributed, scale-out execution, as the data may reside entirely in main memory of a single powerful server. Still, users want to use Spark for its familiar interface and tooling. In such scale-up scenarios, Spark's performance is suboptimal, as Spark prioritizes handling *data size* over optimizing the *computations* on that data. For such medium-size workloads, performance may still be of critical importance if jobs are computationally heavy, need to be run frequently on changing data, or interface with external libraries and systems (e.g., TensorFlow for machine learning).

We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. Inspired by query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures, in particular processing data directly from optimized file formats and combining SQL-style relational processing with external frameworks such as TensorFlow.

Introduction

Modern data analytics applications require a combination of different programming paradigms, spanning relational, procedural, and map-reduce-style functional processing. Systems like Apache Spark [8] have gained enormous traction thanks to their intuitive APIs and ability to scale to very large data sizes, thereby commoditizing petabyte-scale (PB) data processing for large numbers of users. But thanks to its attractive programming interface and tooling, people are also increasingly using Spark for smaller workloads. Even for companies that *also* have PB-scale data, there is typically a long tail of tasks of much smaller size, which make up a very important class of workloads [17, 44]. In such cases, Spark's performance is suboptimal. For such medium-size workloads, performance may still be of critical importance if there are many such jobs, individual jobs are computationally heavy, or need to be run very frequently on changing data. This is the problem we address in this paper. We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. A high-level view of Flare's architecture can be seen in Figure 1b.

Inspiration from In-Memory Databases Flare is based on native code generation techniques that have been pioneered by in-memory databases (e.g., HyPer [35]). Given the multitude of front-end programming paradigms, it is not immediately clear that looking at relational databases is the right idea. However, we argue that this is indeed the right strategy: Despite the variety of front-end interfaces, contemporary Spark is, at its core, an SQL engine and query optimizer [8]. Rich front-end APIs are increasingly based on DataFrames, which are internally represented very much like SQL query plans. Data frames provide a deferred API, i.e., calls only *construct* a query plan, but do not execute it immediately. Thus, front-end abstractions do not interfere with query optimization. Previous generations of Spark relied critically on arbitrary UDFs, but this is becoming less and less of a concern as more and more func-

USENIX Association 13th USENIX Symposium on Operating Systems Design and Implementation 799

Figure 1: Sample image for demo.

Listing 1: Edge Detection using Canny

```
Canny(blurred, canny, 75, 200);
```

Starting from the left, `blurred` is the grayscale blurred image varible, `canny` is the output destination variable that shows the edges in the image in binary, and the remaining are the threshold needed for the procedure. Figure 2 is the sample output of the edge detection.

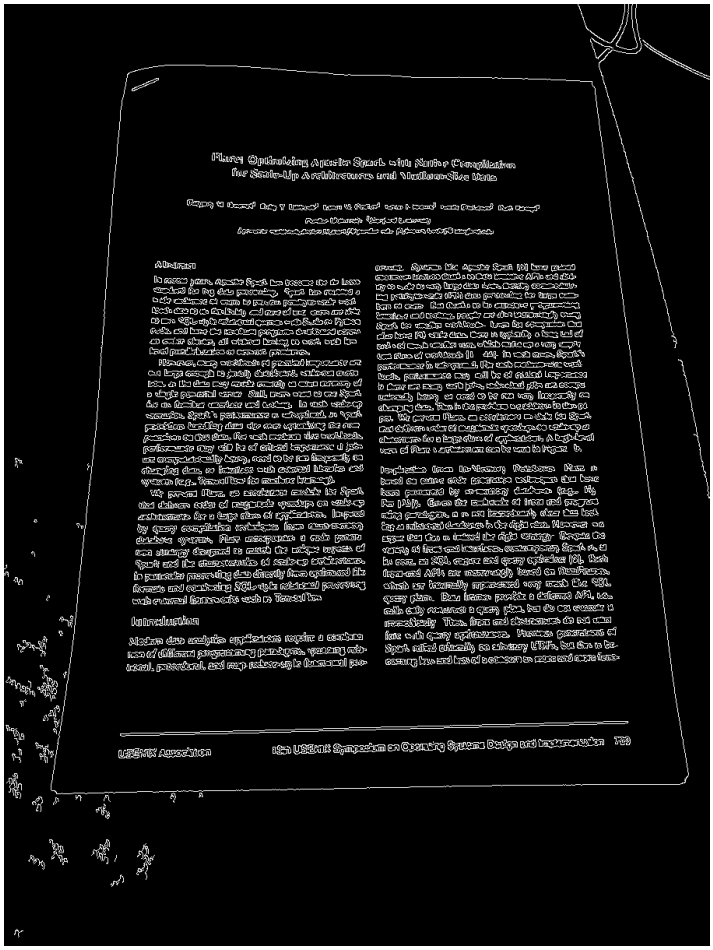


Figure 2: Edge detection output.

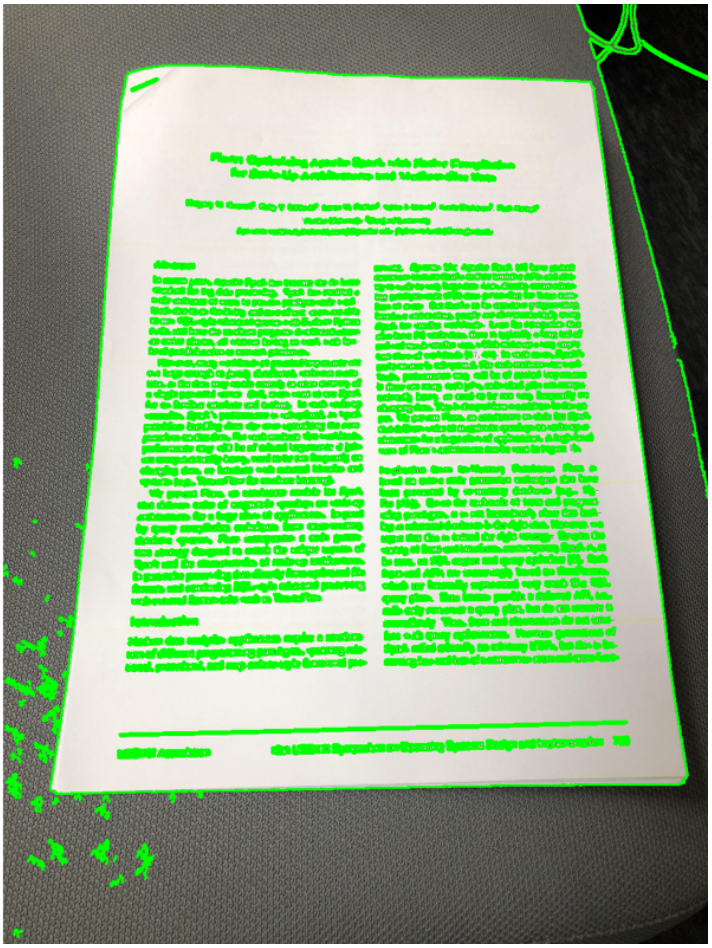


Figure 3: Contour detection output.

2.2 Detecting Document Boundary

Now, using the edge-detected binary image output we are going to detect the boundary of the document. First step is to find all the contours using `findContours` [3] function in OpenCV. This function emits vector of set of points that are found as a contour in the image, like in Listing 2.

Listing 2: Edge Detection using Canny

```
vector<vector<Point>> contours;
vector<Vec4i> hierarchy; // not used.
findContours(canny, contours, hierarchy,
    RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0));
```

So using the `canny`, which is the output of the edge detection, the set of points that make up the contour in the image are saved in the `contours` variable. `hierarchy` variable is not used in our code, and the other parameter are basic setup for the function use. The `contours` variable will contain too many set of points that are out of our interest like in Figure 3.

Then how are we going to select the document boundary among many contours detected? We have two clues to do this. First one is that we can say that the biggest contour is most likely to be the document boundary, and second one is that document will be in a rectangular shape, which means it has exactly four points. Let me explain this in code-level in Listing 3.

Listing 3: Boundary Detection

```
sort(contours.begin(), contours.end(),
    compare_area);

vector<Point> points;
for (vector<Point> contour : contours) {
    vector<Point> temp;
    // Calculate the perimeter of the contour
    double perimeter = arcLength(contour, true);
    // Using the perimeter, approximate the
        polygon of the contour
    approxPolyDP(contour, temp, 0.02 * perimeter,
        true);

    // Starting from the biggest polygon, if it
```

```
        has four points, break the loop
    if (temp.size() == 4) {
        points = temp;
        break;
    }
}
```

To add some explanation, we sort the contours according to its area to find the biggest contour. I implemented a comparison function, `compare_area`, using `contourArea` [4]. Then using the contours, we should approximate the polygon of the contour to estimate the shape of the contour. `arcLength` [5] and `approxPolyDP` [6] is used in the process. They calculate the perimeter of the contour and approximate the polygon using the perimeter, respectively. The approximation emits the corner points of the polygon as an output. When the polygon has exactly four points, then we break the loop since we finally found the biggest rectangle in the image. The approximated rectangle is detected just like in Figure 4.

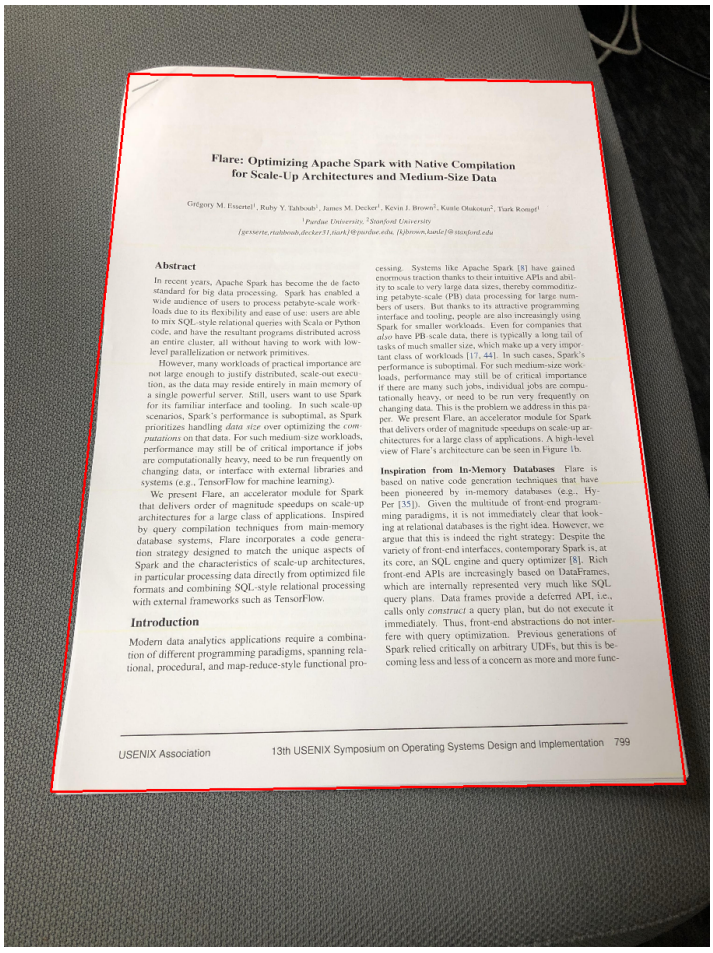


Figure 4: Document boundary estimation.

2.3 Homography Transformation

We finally detected the boundary of the document and now it is time to transform the image using homography matrix. This is something like warping the perspective. First, we have to solve for the homography matrix to warp the image. In order to do to this, we need at least 4 corresponding points. We have already solved for the 4 source points, which are the corner points of the image boundary. To match them with destination points, I tried to sort the points according to x coordinate, which gives the order of bottom-left, top-left, top-right, bottom-right for trapezoid. However, to cover other rectangular shapes other than trapezoid, I used another algorithm to detect the order of the corner points. It is pretty simple, top-left corner would be the point where the sum of the value of x coordinate and y coordinate is smallest and bottom-right would be the biggest. For top-right corner and bottom-left corner, we can use the difference of the value of x and y coordinate. Then as we know the order of the points, you can just match the points in the order you want with the destination points. I assumed that the input image's shape, whether it is in portrait or landscape manner, is identical to the document in the picture and so used the points of the input. (For the case where the shape is different, I just resized it by changing the length of row and column after getting the transformed image.) So now that we have the 4 matches of corresponding points, we compute the homography matrix using singular value decomposition, demonstrated in Listing 4.

Listing 4: Homography matrix calculation

```
template <typename T>
Mat cal_H(vector<Point2f> srcPoints,
    vector<Point2f> dstPoints, int
    number_of_points) {

    Mat A(2 * number_of_points, 9, CV_64F,
        Scalar(0));

    for (int i = 0; i < number_of_points; i++) {
        // initializing A
        // ...
    }
    Mat w, u, vt;
    SVD::compute(A, w, u, vt, SVD::FULL_UV);
    Mat v = vt.t();

    Mat temp = v.col(v.cols - 1);
    Mat H(3, 3, CV_64F);
    // reshaping temp to H
```

```
    // ...
    return H;
}
```

Note that I abbreviated the process of initializing matrix `A` and reshaping matrix `H` for simplification. As you can see, I used `SVD::compute` [7] for singular value decomposition. So by using this implemented function, we obtain the homography matrix.

Finally, using the homography matrix we can warp the image. To implement the function for warping, I applied inverse warping technique and bilinear interpolation technique. At first, it didn't work well, so I blamed the bilinear interpolation and implemented bicubic interpolation. However it showed no good result, and I found a wrong implementation of bilinear interpolation. At last, I could implement a function showing an output as good as the OpenCV-provided function. The implementation is shown in Listing 5 in pseudo code. I assumed that the input image is in grayscale for this time. The output for the transformation is in Figure 5

Listing 5: Inverse warping with bilinear interpolation

```
Mat warping(Mat input, Mat T) {
    Mat result;
    // ... initialize
    T = T.inv();
    for (i = 0; i < result.rows; i++) {
        for (j = 0; j < result.cols; j++) {
            // inverse warping for homography
            z = (T.at(2, 0) * j + T.at(2, 1) * i +
                T.at(2, 2));
            y = ((T.at(0, 0) * j + T.at(0, 1) * i +
                T.at(0, 2)) / z);
            x = ((T.at(1, 0) * j + T.at(1, 1) * i +
                T.at(1, 2)) / z);

            y1 = floor(y); y2 = ceil(y);
            x1 = floor(x); x2 = ceil(x);

            mu = y - y1; lambda = x - x1;
            // bilinear interpolation
            if ( ... ) { // in window range
                step_1 = mu * input.at(x1, y2) +
                        (1 - mu)*input.at(x1, y1);
                step_2 = mu*input.at(x2, y2) +
                        (1 - mu)*input.at(x2, y1);
                step_3 = lambda * step_2 +
                        (1 - lambda)*step_1;

                result.at(i, j) = step_3;
            }
        //return result after the loop
```

2.4 Binary Image Support

In order to provide the option for binary image, I implemented a function performing adaptive thresholding. Given a certain threshold, if the pixel values exceed the threshold, it will be marked black, else in white. Let me skip the code since this is not that complicated to implement. We just add up the pixel values in the kernel to decide whether we assign black or not. The output is shown in Figure 6

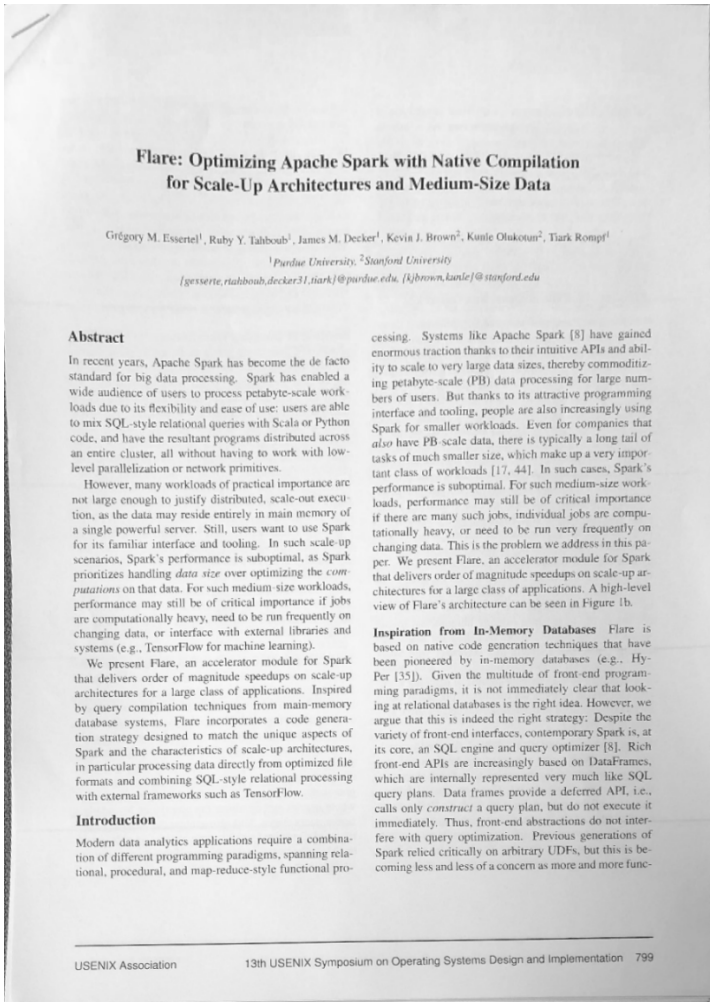
3. Results and Analysis

The results of sample demo image are shown in Figure 5 and Figure 6, grayscale and binary respectively. As you can see, the images are undistorted and quite clear to recognize. Additionally, I attached the result of the cases where I ran the images in the `Dataset.zip`, from `pic00.jpg` to `pic11.jpg`. Since the original image takes too much time to run, I resized the pictures to 30% of its original size. So the output isn't that clear. For `pic01.jpg`, I resized it to 50% and the output was clear. Let me skip with the binary images from `pic02.jpg`. And also my code isn't applicable to every size of the image. Appropriate parameters need to be set according to the image size.

4. Miscellaneous

The demo video about the usage of this project can be found in the link here [8] as reference. Table below shows functions that I used in this project, and whether I implemented it on my own or not.

사용한 함수	출처
Gaussian Blur	직접 구현
Canny Edge Detection	OpenCV
Contour Detection	OpenCV
Perimeter calculation	OpenCV
Polygon Approximation	OpenCV
Homography Calculation	직접 구현
Singular Value Decomposition	OpenCV
Warping with Homography	직접 구현
Adaptive Thresholding	직접 구현



Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data

Grégory M. Essertel[1], Ruby Y. Tahboub[1], James M. Decker[1], Kevin J. Brown[2], Kunle Olukotun[2], Tiark Rompf[1]

[1]Purdue University, [2]Stanford University

{gesserte,rtahboub,decker31,tiark}@purdue.edu, {kjbrown,kunle}@stanford.edu

Abstract

In recent years, Apache Spark has become the de facto standard for big data processing. Spark has enabled a wide audience of users to process petabyte-scale workloads due to its flexibility and ease of use: users are able to mix SQL-style relational queries with Scala or Python code, and have the resultant programs distributed across an entire cluster, all without having to work with low-level parallelization or network primitives.

However, many workloads of practical importance are not large enough to justify distributed, scale-out execution, as the data may reside entirely in main memory of a single powerful server. Still, users want to use Spark for its familiar interface and tooling. In such scale-up scenarios, Spark's performance is suboptimal, as Spark prioritizes handling *data size* over optimizing the *computations* on that data. For such medium-size workloads, performance may still be of critical importance if jobs are computationally heavy, need to be run frequently on changing data, or interface with external libraries and systems (e.g., TensorFlow for machine learning).

We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. Inspired by query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures, in particular processing data directly from optimized file formats and combining SQL-style relational processing with external frameworks such as TensorFlow.

Introduction

Modern data analytics applications require a combination of different programming paradigms, spanning relational, procedural, and map-reduce-style functional processing. Systems like Apache Spark [8] have gained enormous traction thanks to their intuitive APIs and ability to scale to very large data sizes, thereby commoditizing petabyte-scale (PB) data processing for large numbers of users. But thanks to its attractive programming interface and tooling, people are also increasingly using Spark for smaller workloads. Even for companies that *also* have PB-scale data, there is typically a long tail of tasks of much smaller size, which make up a very important class of workloads [17, 44]. In such cases, Spark's performance is suboptimal. For such medium-size workloads, performance may still be of critical importance if there are many such jobs, individual jobs are computationally heavy, or need to be run very frequently on changing data. This is the problem we address in this paper. We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. A high-level view of Flare's architecture can be seen in Figure 1b.

Inspiration from In-Memory Databases Flare is based on native code generation techniques that have been pioneered by in-memory databases (e.g., HyPer [35]). Given the multitude of front-end programming paradigms, it is not immediately clear that looking at relational databases is the right idea. However, we argue that this is indeed the right strategy: Despite the variety of front-end interfaces, contemporary Spark is, at its core, an SQL engine and query optimizer [8]. Rich front-end APIs are increasingly based on DataFrames, which are internally represented very much like SQL query plans. Data frames provide a deferred API, i.e., calls only *construct* a query plan, but do not execute it immediately. Thus, front-end abstractions do not interfere with query optimization. Previous generations of Spark relied critically on arbitrary UDFs, but this is becoming less and less of a concern as more and more func-

USENIX Association 13th USENIX Symposium on Operating Systems Design and Implementation 799

Figure 5: Document boundary estimation.

Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data

Grégory M. Essertel[1], Ruby Y. Tahboub[1], James M. Decker[1], Kevin J. Brown[2], Kunle Olukotun[2], Tiark Rompf[1]

[1]Purdue University, [2]Stanford University

{gesserte,rtahboub,decker31,tiark}@purdue.edu, {kjbrown,kunle}@stanford.edu

Abstract

In recent years, Apache Spark has become the de facto standard for big data processing. Spark has enabled a wide audience of users to process petabyte-scale workloads due to its flexibility and ease of use: users are able to mix SQL-style relational queries with Scala or Python code, and have the resultant programs distributed across an entire cluster, all without having to work with low-level parallelization or network primitives.

However, many workloads of practical importance are not large enough to justify distributed, scale-out execution, as the data may reside entirely in main memory of a single powerful server. Still, users want to use Spark for its familiar interface and tooling. In such scale-up scenarios, Spark's performance is suboptimal, as Spark prioritizes handling *data size* over optimizing the *computations* on that data. For such medium-size workloads, performance may still be of critical importance if jobs are computationally heavy, need to be run frequently on changing data, or interface with external libraries and systems (e.g., TensorFlow for machine learning).

We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. Inspired by query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures, in particular processing data directly from optimized file formats and combining SQL-style relational processing with external frameworks such as TensorFlow.

Introduction

Modern data analytics applications require a combination of different programming paradigms, spanning relational, procedural, and map-reduce-style functional processing. Systems like Apache Spark [8] have gained enormous traction thanks to their intuitive APIs and ability to scale to very large data sizes, thereby commoditizing petabyte-scale (PB) data processing for large numbers of users. But thanks to its attractive programming interface and tooling, people are also increasingly using Spark for smaller workloads. Even for companies that *also* have PB-scale data, there is typically a long tail of tasks of much smaller size, which make up a very important class of workloads [17, 44]. In such cases, Spark's performance is suboptimal. For such medium-size workloads, performance may still be of critical importance if there are many such jobs, individual jobs are computationally heavy, or need to be run very frequently on changing data. This is the problem we address in this paper. We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. A high-level view of Flare's architecture can be seen in Figure 1b.

Inspiration from In-Memory Databases Flare is based on native code generation techniques that have been pioneered by in-memory databases (e.g., HyPer [35]). Given the multitude of front-end programming paradigms, it is not immediately clear that looking at relational databases is the right idea. However, we argue that this is indeed the right strategy: Despite the variety of front-end interfaces, contemporary Spark is, at its core, an SQL engine and query optimizer [8]. Rich front-end APIs are increasingly based on DataFrames, which are internally represented very much like SQL query plans. Data frames provide a deferred API, i.e., calls only *construct* a query plan, but do not execute it immediately. Thus, front-end abstractions do not interfere with query optimization. Previous generations of Spark relied critically on arbitrary UDFs, but this is becoming less and less of a concern as more and more func-

USENIX Association 13th USENIX Symposium on Operating Systems Design and Implementation 799

Figure 6: Binary output.

References

[1] "OpenCV." https://opencv.org/.

[2] "OpenCV - Canny." https://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=canny#canny.

[3] "OpenCV - findContours." https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#findcontours.

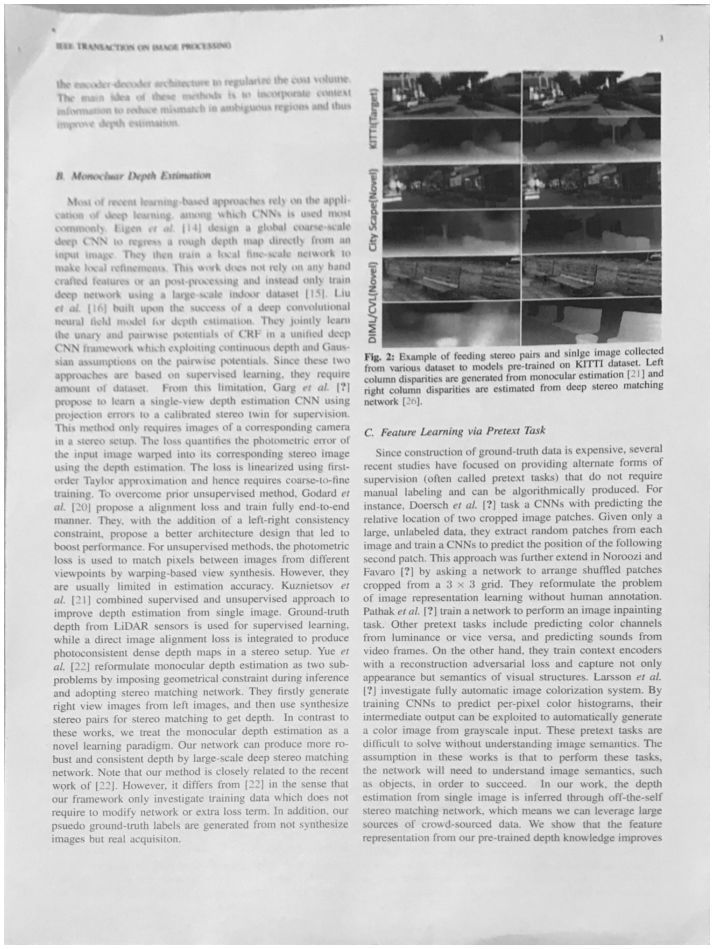
[4] "OpenCV - contourArea." https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#contourarea.

[5] "OpenCV - arcLength ." https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#arclength.

[6] "OpenCV - approxPolyDP." https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#approxpolydp.

[7] "OpenCV - SVD::compute." https://docs.opencv.org/3.4/df/df7/classcv_1_1SVD.html#a6a8081ff2e54fc43a7a844c2df58476c.

[8] "Demo Video." https://drive.google.com/file/d/1CDDrUnZm5WOmbDLBL5ek3vBcPT02EKwD/view?usp=sharing.



IEEE TRANSACTION ON IMAGE PROCESSING 3

the encoder-decoder architecture to regularize the cost volume. The main idea of these methods is to incorporate context information to reduce mismatch in ambiguous regions and thus improve depth estimation.

B. Monocluar Depth Estimation

Most of recent learning-based approaches rely on the application of deep learning, among which CNNs is used most commonly. Eigen *et al.* [14] design a global coarse-scale deep CNN to regress a rough depth map directly from an input image. They then train a local fine-scale network to make local refinements. This work does not rely on any hand crafted features or an post-processing and instead only train deep network using a large-scale indoor dataset [15]. Liu *et al.* [16] built upon the success of a deep convolutional neural field model for depth estimation. They jointly learn the unary and pairwise potentials of CRF in a unified deep CNN framework which exploiting continuous depth and Gaussian assumptions on the pairwise potentials. Since these two approaches are based on supervised learning, they require amount of dataset. From this limitation, Garg *et al.* [?] propose to learn a single-view depth estimation CNN using projection errors to a calibrated stereo twin for supervision. This method only requires images of a corresponding camera in a stereo setup. The loss quantifies the photometric error of the input image warped into its corresponding stereo image using the depth estimation. The loss is linearized using first-order Taylor approximation and hence requires coarse-to-fine training. To overcome prior unsupervised method, Godard *et al.* [20] propose a alignment loss and train fully end-to-end manner. They, with the addition of a left-right consistency constraint, propose a better architecture design that led to boost performance. For unsupervised methods, the photometric loss is used to match pixels between images from different viewpoints by warping-based view synthesis. However, they are usually limited in estimation accuracy. Kuznietsov *et al.* [21] combined supervised and unsupervised approach to improve depth estimation from single image. Ground-truth depth from LiDAR sensors is used for supervised learning, while a direct image alignment loss is integrated to produce photoconsistent dense depth maps in a stereo setup. Yue *et al.* [22] reformulate monocular depth estimation as two sub-problems by imposing geometrical constraint during inference and adopting stereo matching network. They firstly generate right view images from left images, and then use synthesize stereo pairs for stereo matching to get depth. In contrast to these works, we treat the monocular depth estimation as a novel learning paradigm. Our network can produce more robust and consistent depth by large-scale deep stereo matching network. Note that our method is closely related to the recent work of [22]. However, it differs from [22] in the sense that our framework only investigate training data which does not require to modify network or extra loss term. In addition, our psuedo ground-truth labels are generated from not synthesize images but real acquisiton.

Fig. 2: Example of feeding stereo pairs and sinlge image collected from various dataset to models pre-trained on KITTI dataset. Left column disparities are generated from monocular estimation [21] and right column disparities are estimated from deep stereo matching network [26].

C. Feature Learning via Pretext Task

Since construction of ground-truth data is expensive, several recent studies have focused on providing alternate forms of supervision (often called pretext tasks) that do not require manual labeling and can be algorithmically produced. For instance, Doersch *et al.* [?] task a CNNs with predicting the relative location of two cropped image patches. Given only a large, unlabeled data, they extract random patches from each image and train a CNNs to predict the position of the following second patch. This approach was further extend in Noroozi and Favaro [?] by asking a network to arrange shuffled patches cropped from a 3×3 grid. They reformulate the problem of image representation learning without human annotation. Pathak *et al.* [?] train a network to perform an image inpainting task. Other pretext tasks include predicting color channels from luminance or vice versa, and predicting sounds from video frames. On the other hand, they train context encoders with a reconstruction adversarial loss and capture not only appearance but semantics of visual structures. Larsson *et al.* [?] investigate fully automatic image colorization system. By training CNNs to predict per-pixel color histograms, their intermediate output can be exploited to automatically generate a color image from grayscale input. These pretext tasks are difficult to solve without understanding image semantics. The assumption in these works is that to perform these tasks, the network will need to understand image semantics, such as objects, in order to succeed. In our work, the depth estimation from single image is inferred through off-the-self stereo matching network, which means we can leverage large sources of crowd-sourced data. We show that the feature representation from our pre-trained depth knowledge improves

Figure 7: Grayscale output for pic00.jpg. The top left part was out-of-focus in the original image.

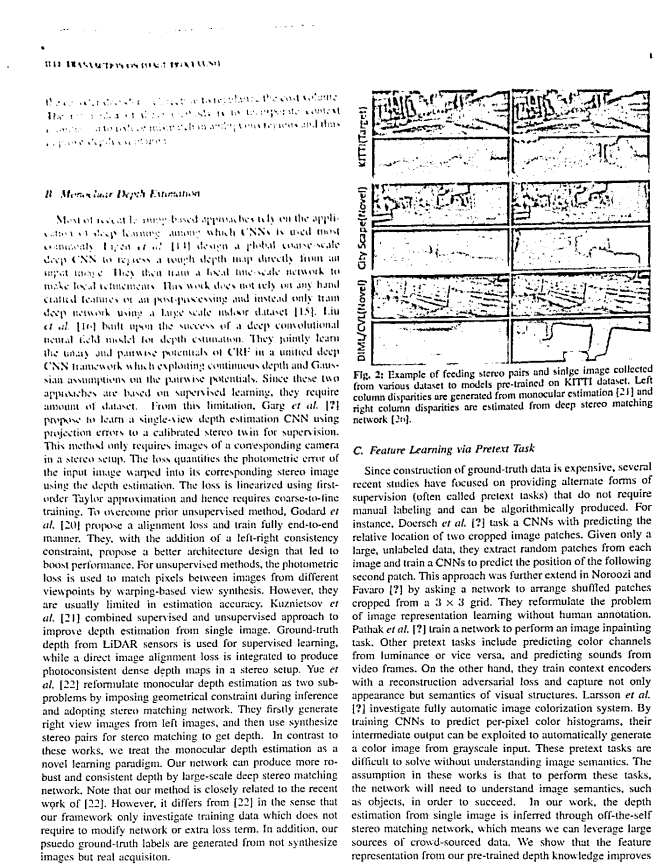


Figure 8: Binary output for pic00.jpg.

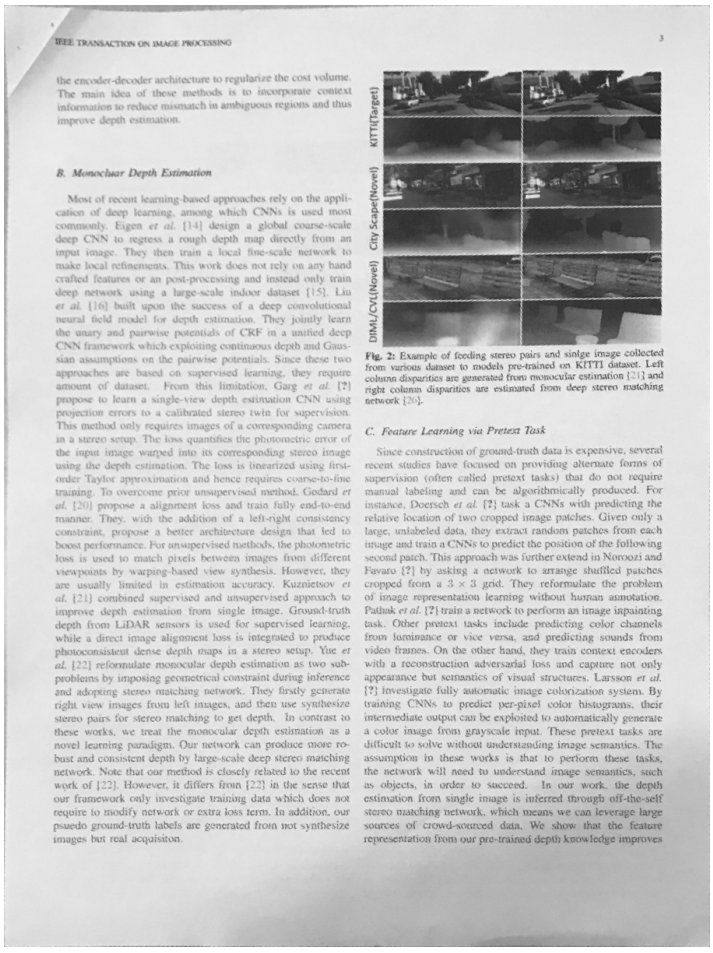


Figure 9: Grayscale output for pic01.jpg.

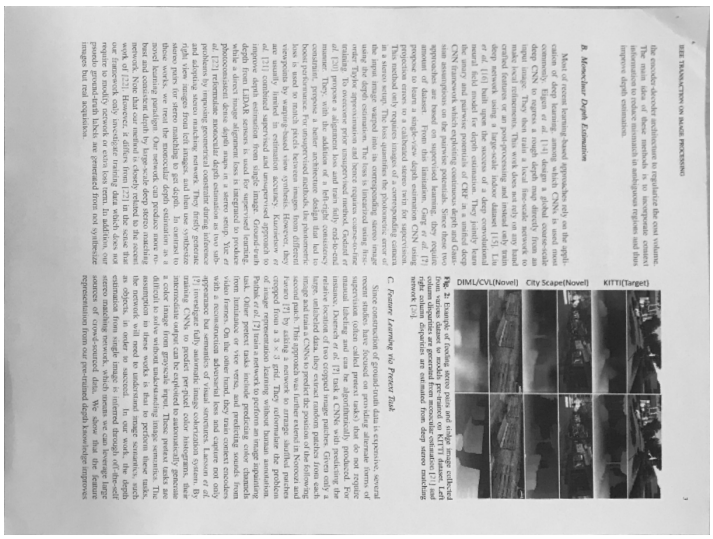
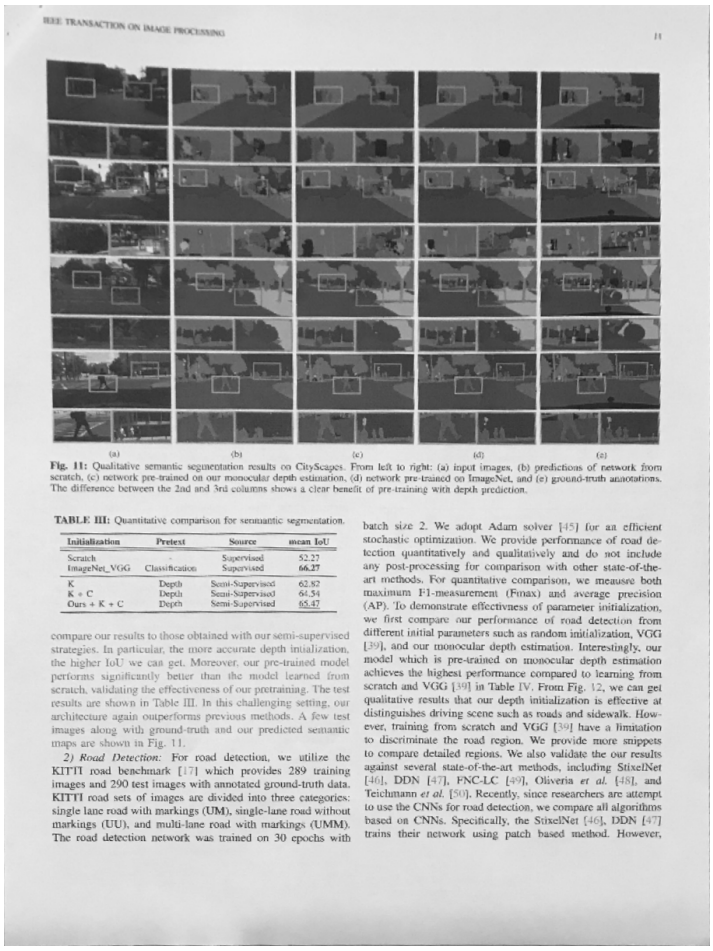


Figure 10: Grayscale output for pic02.jpg.

Figure 11: Grayscale output for pic03.jpg.

Figure 12: Grayscale output for pic04.jpg.

Figure 13: Grayscale output for pic05.jpg.



IEEE TRANSACTION ON IMAGE PROCESSING 11

(a) (b) (c) (d) (e)

Fig. 11: Qualitative semantic segmentation results on CityScapes. From left to right: (a) input images, (b) predictions of network from scratch, (c) network pre-trained on our monocular depth estimation, (d) network pre-trained on ImageNet, and (e) ground-truth annotations. The difference between the 2nd and 3rd columns shows a clear benefit of pre-training with depth prediction.

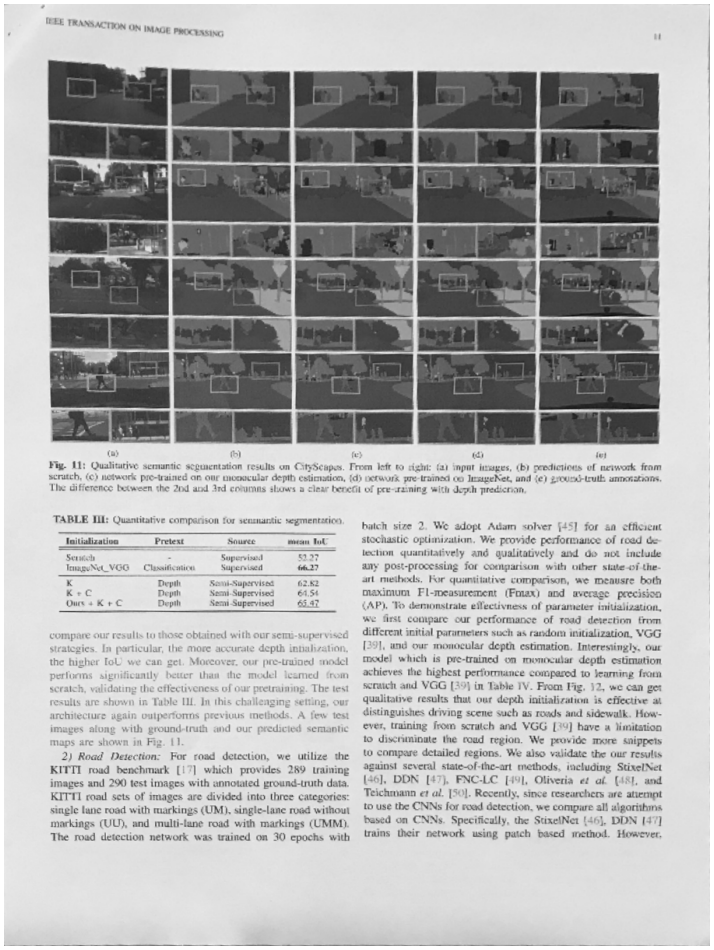
TABLE III: Quantitative comparison for semantic segmentation.

Initialization	Pretext	Source	mean IoU
Scratch	-	Supervised	52.27
ImageNet_VGG	Classification	Supervised	**66.27**
K	Depth	Semi-Supervised	62.82
K + C	Depth	Semi-Supervised	64.54
Ours + K + C	Depth	Semi-Supervised	65.47

compare our results to those obtained with our semi-supervised strategies. In particular, the more accurate depth initialization, the higher IoU we can get. Moreover, our pre-trained model performs significantly better than the model learned from scratch, validating the effectiveness of our pretraining. The test results are shown in Table III. In this challenging setting, our architecture again outperforms previous methods. A few test images along with ground-truth and our predicted semantic maps are shown in Fig. 11.

2) Road Detection: For road detection, we utilize the KITTI road benchmark [17] which provides 289 training images and 290 test images with annotated ground-truth data. KITTI road sets of images are divided into three categories: single lane road with markings (UM), single-lane road without markings (UU), and multi-lane road with markings (UMM). The road detection network was trained on 30 epochs with batch size 2. We adopt Adam solver [45] for an efficient stochastic optimization. We provide performance of road detection quantitatively and qualitatively and do not include any post-processing for comparison with other state-of-the-art methods. For quantitative comparison, we meausre both maximum F1-measurement (Fmax) and average precision (AP). To demonstrate effectivness of parameter initialization, we first compare our performance of road detection from different initial parameters such as random initialization, VGG [39], and our monocular depth estimation. Interestingly, our model which is pre-trained on monocular depth estimation achieves the highest performance compared to learning from scratch and VGG [39] in Table IV. From Fig. 12, we can get qualitative results that our depth initialization is effective at distinguishes driving scene such as roads and sidewalk. However, training from scratch and VGG [39] have a limitation to discriminate the road region. We provide more snippets to compare detailed regions. We also validate the our results against several state-of-the-art methods, including StixelNet [46], DDN [47], FNC-LC [49], Oliveria *et al.* [48], and Teichmann *et al.* [50]. Recently, since researchers are attempt to use the CNNs for road detection, we compare all algorithms based on CNNs. Specifically, the StixelNet [46], DDN [47] trains their network using patch based method. However,

Figure 14: Grayscale output for pic06.jpg.



IEEE TRANSACTION ON IMAGE PROCESSING 11

(a) (b) (c) (d) (e)

Fig. 11: Qualitative semantic segmentation results on CityScapes. From left to right: (a) input images, (b) predictions of network from scratch, (c) network pre-trained on our monocular depth estimation, (d) network pre-trained on ImageNet, and (e) ground-truth annotations. The difference between the 2nd and 3rd columns shows a clear benefit of pre-training with depth prediction.

TABLE III: Quantitative comparison for semantic segmentation.

Initialization	Pretext	Source	mean IoU
Scratch	-	Supervised	52.27
ImageNet_VGG	Classification	Supervised	**66.27**
K	Depth	Semi-Supervised	62.82
K + C	Depth	Semi-Supervised	64.54
Ours + K + C	Depth	Semi-Supervised	65.47

compare our results to those obtained with our semi-supervised strategies. In particular, the more accurate depth initialization, the higher IoU we can get. Moreover, our pre-trained model performs significantly better than the model learned from scratch, validating the effectiveness of our pretraining. The test results are shown in Table III. In this challeaging setting, our architecture again outperforms previous methods. A few test images along with ground-truth and our predicted semantic maps are shown in Fig. 11.

2) Road Detection: For road detection, we utilize the KITTI road benchmark [17] which provides 289 training images and 290 test images with annotated ground-truth data. KITTI road sets of images are divided into three categories: single lane road with markings (UM), single-lane road without markings (UU), and multi-lane road with markings (UMM). The road detection network was trained on 30 epochs with batch size 2. We adopt Adam solver [45] for an efficient stochastic optimization. We provide performance of road detection quantitatively and qualitatively and do not include any post-processing for comparison with other state-of-the-art methods. For quantitative comparison, we meausre both maximum F1-measurement (Fmax) and average precision (AP). To demonstrate effectivness of parameter initialization, we first compare our performance of road detection from different initial parameters such as random initialization, VGG [39], and our monocular depth estimation. Interestingly, our model which is pre-trained on monocular depth estimation achieves the highest performance compared to learning from scratch and VGG [39] in Table IV. From Fig. 12, we can get qualitative results that our depth initialization is effective at distinguishes driving scene such as roads and sidewalk. However, training from scratch and VGG [39] have a limitation to discriminate the road region. We provide more snippets to compare detailed regions. We also validate the our results against several state-of-the-art methods, including StixelNet [46], DDN [47], FNC-LC [49], Oliveria *et al.* [48], and Teichmann *et al.* [50]. Recently, since researchers are attempt to use the CNNs for road detection, we compare all algorithms based on CNNs. Specifically, the StixelNet [46], DDN [47] trains their network using patch based method. However,

Figure 15: Grayscale output for pic07.jpg.

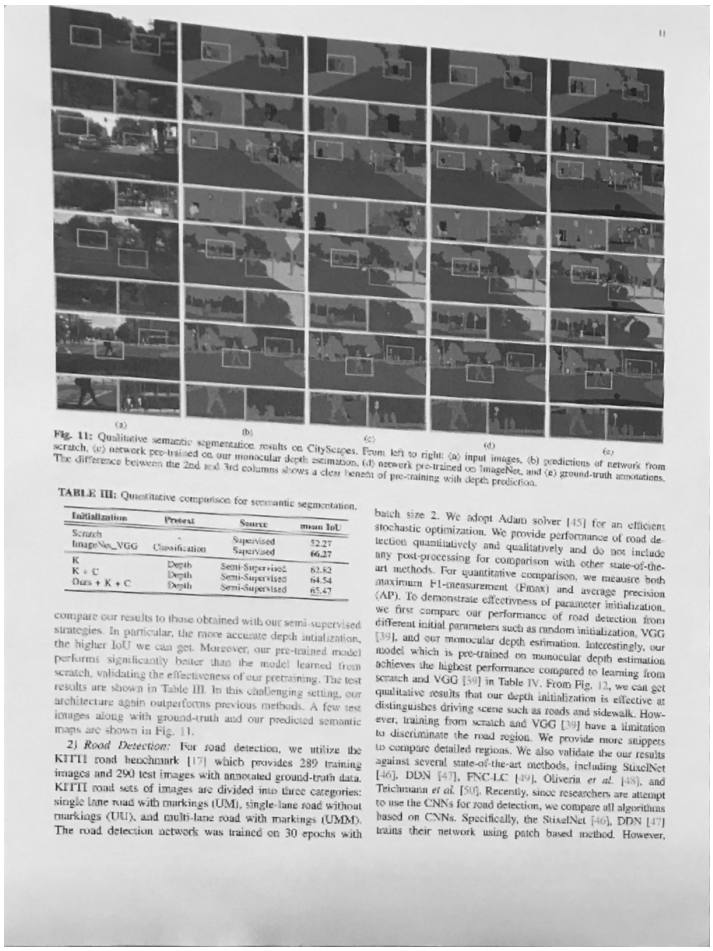


Fig. 11: Qualitative semantic segmentation results on CityScapes. From left to right: (a) input images, (b) predictions of network from scratch, (c) network pre-trained on our monocular depth estimation, (d) network pre-trained on ImageNet, and (e) ground-truth annotations. The difference between the 2nd and 3rd columns shows a clear benefit of pre-training with depth prediction.

TABLE III: Quantitative comparison for semantic segmentation.

Initialization	Pretext	Source	mean IoU
Scratch	-	Supervised	52.27
ImageNet_VGG	Classification	Supervised	66.27
K	Depth	Semi-Supervised	62.82
K + C	Depth	Semi-Supervised	64.54
Oats + K + C	Depth	Semi-Supervised	65.47

compare our results to those obtained with our semi-supervised strategies. In particular, the more accurate depth initialization, the higher IoU we can get. Moreover, our pre-trained model performs significantly better than the model learned from scratch, validating the effectiveness of our pretraining. The test results are shown in Table III. In this challenging setting, our architecture again outperforms previous methods. A few test images along with ground-truth and our predicted semantic maps are shown in Fig. 11.

2) Road Detection: For road detection, we utilize the KITTI road benchmark [17] which provides 289 training images and 290 test images with annotated ground-truth data. KITTI road sets of images are divided into three categories: single lane road with markings (UM), single-lane road without markings (UU), and multi-lane road with markings (UMM). The road detection network was trained on 30 epochs with batch size 2. We adopt Adam solver [45] for an efficient stochastic optimization. We provide performance of road detection quantitatively and qualitatively and do not include any post-processing for comparison with other state-of-the-art methods. For quantitative comparison, we measure both maximum F1-measurement (Fmax) and average precision (AP). To demonstrate effectiveness of parameter initialization, we first compare our performance of road detection from different initial parameters such as random initialization, VGG [39], and our monocular depth estimation. Interestingly, our model which is pre-trained on monocular depth estimation achieves the highest performance compared to learning from scratch and VGG [39] in Table IV. From Fig. 12, we can get qualitative results that our depth initialization is effective at distinguishes driving scene such as roads and sidewalk. However, training from scratch and VGG [39] have a limitation to discriminate the road region. We provide more snippets to compare detailed regions. We also validate the our results against several state-of-the-art methods, including StixelNet [46], DDN [47], FNC-LC [49], Oliveira *et al.* [48], and Teichmann *et al.* [50]. Recently, since researchers are attempt to use the CNNs for road detection, we compare all algorithms based on CNNs. Specifically, the StixelNet [46], DDN [47] trains their network using patch based method. However,

Figure 16: Grayscale output for pic08.jpg.

Figure 17: Grayscale output for pic09.jpg.

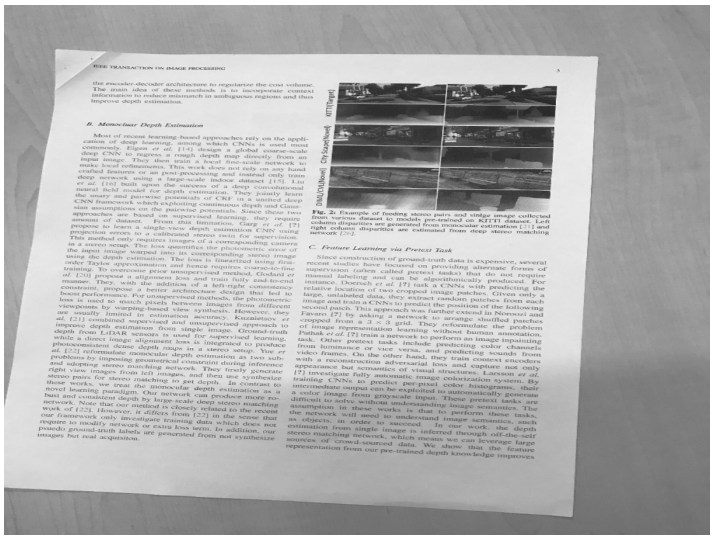


Figure 18: Grayscale output for pic10.jpg.

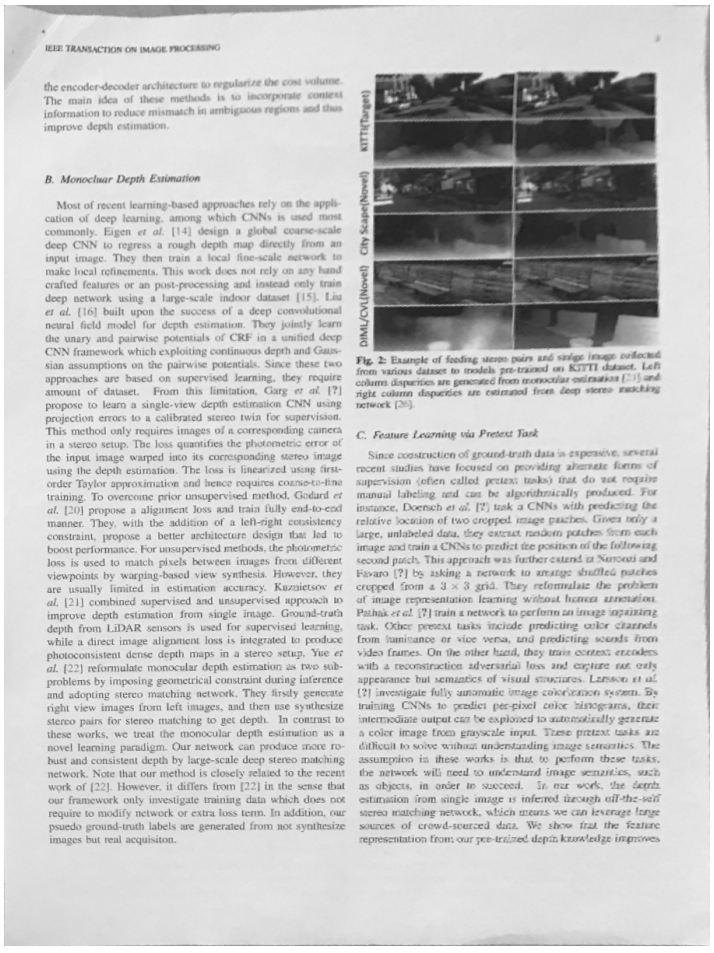


Fig. 2: Example of feeding stereo pairs and single image collected from various dataset to models pre-trained on KITTI dataset. Left column disparities are generated from monocular estimation [21] and right column disparities are estimated from deep stereo matching network [26].

IEEE TRANSACTION ON IMAGE PROCESSING

the encoder-decoder architecture to regularize the cost volume. The main idea of these methods is to incorporate context information to reduce mismatch in ambiguous regions and thus improve depth estimation.

B. Monocluar Depth Estimation

Most of recent learning-based approaches rely on the application of deep learning, among which CNNs is used most commonly. Eigen *et al.* [14] design a global coarse-scale deep CNN to regress a rough depth map directly from an input image. They then train a local fine-scale network to make local refinements. This work does not rely on any hand crafted features or an post-processing and instead only train deep network using a large-scale indoor dataset [15]. Liu *et al.* [16] built upon the success of a deep convolutional neural field model for depth estimation. They jointly learn the unary and pairwise potentials of CRF in a unified deep CNN framework which exploiting continuous depth and Gaussian assumptions on the pairwise potentials. Since these two approaches are based on supervised learning, they require amount of dataset. From this limitation, Garg *et al.* [?] propose to learn a single-view depth estimation CNN using projection errors to a calibrated stereo twin for supervision. This method only requires images of a corresponding camera in a stereo setup. The loss quantifies the photometric error of the input image warped into its corresponding stereo image using the depth estimation. The loss is linearized using first-order Taylor approximation and hence requires coarse-to-fine training. To overcome prior unsupervised method, Godard *et al.* [20] propose a alignment loss and train fully end-to-end manner. They, with the addition of a left-right consistency constraint, propose a better architecture design that led to boost performance. For unsupervised methods, the photometric loss is used to match pixels between images from different viewpoints by warping-based view synthesis. However, they are usually limited in estimation accuracy. Kuznietsov *et al.* [21] combined supervised and unsupervised approach to improve depth estimation from single image. Ground-truth depth from LiDAR sensors is used for supervised learning, while a direct image alignment loss is integrated to produce photoconsistent dense depth maps in a stereo setup. Yue *et al.* [22] reformulate monocular depth estimation as two sub-problems by imposing geometrical constraint during inference and adopting stereo matching network. They firstly generate right view images from left images, and then use synthesize stereo pairs for stereo matching to get depth. In contrast to these works, we treat the monocular depth estimation as a novel learning paradigm. Our network can produce more robust and consistent depth by large-scale deep stereo matching network. Note that our method is closely related to the recent work of [22]. However, it differs from [22] in the sense that our framework only investigate training data which does not require to modify network or extra loss term. In addition, our psuedo ground-truth labels are generated from not synthesize images but real acquisition.

C. Feature Learning via Pretext Task

Since construction of ground-truth data is expensive, several recent studies have focused on providing alternate forms of supervision (often called pretext tasks) that do not require manual labeling and can be algorithmically produced. For instance, Doersch *et al.* [?] task a CNNs with predicting the relative location of two cropped image patches. Given only a large, unlabeled data, they extract random patches from each image and train a CNNs to predict the position of the second patch. This approach was further extend in Noroozi and Favaro [?] by asking a network to arrange shuffled patches cropped from a 3×3 grid. They reformulate the problem of image representation learning without human annotation. Pathak *et al.* [?] train a network to perform an image inpainting task. Other pretext tasks include predicting color channels from luminance or vice versa, and predicting sounds from video frames. On the other hand, they train context encoders with a reconstruction adversarial loss and capture not only appearance but semantics of visual structures. Larsson *et al.* [?] investigate fully automatic image colorization system. By training CNNs to predict per-pixel color histograms, their intermediate output can be exploited to automatically generate a color image from grayscale input. These pretext tasks are difficult to solve without understanding image semantics. The assumption in these works is that to perform these tasks, the network will need to understand image semantics, such as objects, in order to succeed. In our work, the depth estimation from single image is inferred through off-the-self stereo matching network, which means we can leverage large sources of crowd-sourced data. We show that the feature representation from our pre-trained depth knowledge improves

Figure 19: Grayscale output for pic11.jpg