Energy-Optimized Mapping of Application to Smartphone Platform – A Case Study of Mobile Face Recognition

Yi-Chu Wang
Dept. of Electrical and Computer Engineering,
University of California, Santa Barbara, USA
yichuwang@umail.ucsb.edu

Kwang-Ting (Tim) Cheng
Dept. of Electrical and Computer Engineering,
University of California, Santa Barbara, USA
timcheng@ece.ucsb.edu

Abstract

Modern smartphones use heterogeneous multi-core SoC which includes CPU, GPU, DSP and various application-specific accelerators. It provides opportunities to realize compute-intensive applications on a battery-powered and resource-limited mobile device by assigning each sub-task to the most suitable computing core. To meet the performance requirement with minimized energy consumption, the algorithm also needs to be characterized to identify its adaptability to the performance and energy/power trade-off. In this paper, we use face recognition as an application driver and Nvidia’s Tegra SoC/platform as a target platform to explore the strategies of application-to-platform mapping for energy minimization and performance optimization. We demonstrate that tuning the algorithms for the platform can significantly reduce the computational complexity to meet the real-time performance requirement with very little compromise in the recognition accuracy. We further demonstrate that utilizing the mobile GPU inside the Tegra SoC for feature extraction, the most compute-intensive task in this application, can achieve 51% reduction in runtime and 50% reduction in total energy consumption, in comparison with an implementation which uses the CPU only.

1. Introduction

With the pervasive presence of low-cost, high quality cameras on mobile devices, we are witnessing the beginning of an explosive growth of embedded computer vision applications, such as mobile augmented reality [1], mobile image search and real-time image content analysis [2]. Most vision tasks rely heavily on floating-point calculation and are memory intensive. Realizing computer vision applications in a mobile embedded environment with satisfactory real-time performance remains a very challenging task. Due to this limitation, applications targeting mobile platforms often need to remove computationally expensive operations from the algorithms [3], or to rely on the clouds for the heavy computation [4].

Modern smartphone platforms have already integrated heterogeneous cores, including multiple CPUs, GPU, DSP, and application-specific accelerators, in a single chip (e.g. TI’s OMAP3 [5], Nvidia’s Tegra2 [6], and Qualcomm’s SnapDragon [7]) which is powerful enough to tackle many compute-intensive tasks. It is believed that the trend of incorporating more cores, in terms of both the count and the variety, in a single mobile SoC will continue to accelerate. In order to explore the true benefits of the increasing computing power available in a heterogeneous mobile platform, it is necessary to revisit the vision algorithms to identify the opportunity of better utilizing the platform’s heterogeneous resources for both performance improvement and energy minimization. While it’s desirable to make these algorithm-adaptation and optimization processes as general as possible, they are inevitably platform-dependent. Consider the utilization of a GPU as an example. Almost all existing GPU-accelerated vision algorithms assume the use of desktop- or server-based GPUs and do not take into account the impact on power/energy consumption. Porting such implementations to a mobile GPU may not yield much, or any, acceleration due to the drastic differences in GPU architecture, system memory bus architecture, and the available memory bandwidth between a mobile GPU and a desktop GPU [8]. Therefore, the optimization strategies of using a mobile GPU to accelerate a general-purposed vision task could be very different, which is yet to be studied.

The power and energy consumption is the most critical design consideration for smartphone applications. The overall objective of mapping a vision application to a mobile platform should be to maximize per-energy user experience. In this paper, we explore the strategies of mapping a computer vision application to a smartphone platform and use face recognition as the application driver for this study. The ability of achieving high-accuracy face recognition in real-time right on a smartphone could enable popular applications, such as automatic face annotation of photos and videos, without the need of high-bandwidth communications and cloud computing. We use Nvidia’s Tegra SoC/platform [6], which is specifically designed for smartphones and tablets, as the target platform in our study. Our study focuses on two aspects: (1) algorithm-level adaptation, and (2) utilization of on-chip mobile GPU for both performance and energy optimization. We demonstrate that there is a wide range of available trade-offs among performance, energy consumption, and accuracy for this vision task. Based on the performance and accuracy requirements, an energy-optimized design which meets these requirements can then be chosen for implementation.

The rest of the paper is organized as follow: In Section 2, we briefly discuss the general strategies of application-to-platform mapping. Section 3 explains our application driver, a face recognition system, and analyzes its workloads. In Section 4, we investigate the tuning knobs in the face recognition algorithm for exploration of trade-offs among the accuracy and the execution time. Section 5 investigates the use of an embedded GPU for performance and energy optimization. The experimental setup of our study and results are presented in Section 6. Finally we
conclude our study and the future directions.

2. Algorithm-to-platform mapping strategy

The vision algorithms running on a desktop or server are often optimized for accuracy and runtime performance. The process of porting them to an embedded system involves identifying suitable tuning knobs available in the algorithm which allow exploration of performance, runtime and energy trade-offs. After the tradeoffs are fully characterized with respect to various settings of the tuning knobs, an operating point which minimizes the total energy while meeting the accuracy and performance requirements can then be chosen for implementation.

As mobile platforms have evolved from single-core CPUs, to multi-core CPUs, to heterogeneous multi-core SOCs, porting vision algorithms to such platforms must be extended to take heterogeneous computing into account. While using multiple and/or different cores to complete a single task could improve the performance, it usually also increase the system power consumption. Depending on the ratio of performance improvement to power increase, the total energy may increase or decrease. It is essential to investigate various strategies for heterogeneous computing for their impacts on performance and total energy consumption. Figure 1 shows an overview of our algorithm-to-platform mapping strategy for a smartphone platform. It consists of two parts: one is to identify platform-independent algorithm tuning knobs for performance-tuning, and the other is to identify the platform characteristic. These two pieces of information are inputs to the implementation stage in which we iteratively vary the algorithmic tuning knobs and/or the task partition, and then estimate the resulting performance and power consumption. This approach is empirical and iterative which stops when a satisfactory implementation is found.

![Figure 1: Energy-minimized algorithm to platform mapping strategy](image1)

3. Face recognition system

Face recognition identifies known people in a given photo. It is a key task for several photo and video sharing and management applications. A face recognition system usually consists of four steps which are illustrated in Figure 2 (a): (1) Face detection, which scans the entire image to identify face regions. (2) Face landmark localization, which identifies the face landmark regions such as the eyes, the nose, and the mouth, and then resizes and registers the face region accordingly. (3) Face feature extraction, which represents a face region by its features that are sufficiently invariant or robust to the variation of illumination, pose, facial expression, and occlusion. (4) Face feature classification, which compares the face feature to the training face set and in turn assigns a name of the most similar identity to the query face.

Figure 2(b) shows an exemplar face annotation application on smartphones. The face regions are identified, recognized and tagged with names automatically on a newly taken photo, or one from the photo gallery in a smartphone. Once a face is recognized and tagged, the user can choose to add it to the face database and/or to link it to the user’s address book. Furthermore, the photo with tagged face(s) could be uploaded to a photo-sharing website such as Picasa in real-time using the smartphone’s Wi-Fi or 3G network connectivity.

![Figure 2: (a) Steps in a face recognition system. (b) One application scenario on a smartphone.](image2)

3.1. Face feature extraction

The state-of-the-art face detection technology can achieve a great detection rate with fast computation. It has been widely deployed to various consumer products. Face recognition, however, demands very high computational resources, including both CPU utilization and memory, in order to achieve high discriminative power and robustness. Among the many global and local face representations, the Gabor-based feature descriptor [10] is considered as one of the best methods for face recognition. The Gabor wavelet representation of an image is the convolution of the image with a family of Gabor kernels, as defined in [10]. A typical Gabor-based feature descriptor uses 40 different Gabor kernels, which include 5 different scales and 8 different orientations, to extract the structure corresponding to multiple frequencies (scales) and multiple orientations in a local area. In order to reserve more local information, the Gabor-face is further divided into several non-overlapping local patches, each of which has different discriminative power for the purpose of face recognition. An example is shown in Figure 3, where the face is divided into 20 non-overlapped areas. Each local patch is then processed by Principle Component Analysis (PCA) [19] and Linear Discriminant Analysis (LDA) [20] to reduce its feature dimensionality and forms the final face feature descriptor. The use of Gabor feature combined with the LDA-based
recognition method has been reported to achieve 93.83% accuracy on the traditional face recognition dataset FERET [11].

3.2. Computational requirement

Directly porting the codes designed for a desktop platform to an Android powered Tegra platform and running the program on the Tegra CPU (details will be described in Section 6) takes 8.5 seconds to detect and recognize a person. Table 1 shows both the execution time breakdown and memory requirement of the face recognition pipeline. In this implementation, Android facedetector API [12] is used to identify the face regions in a given photo, and an AdaBoost based eye localization method [21] is used to identify the landmark regions. Face feature classification is performed by using the K-nearest-neighbor method. The PCA and LDA transformation matrix of each face patch is trained individually. This long runtime clearly indicates that the face recognition workload is too heavy for a smartphone platform and it is desirable to accelerate it by at least 5x for offering a satisfactory user experience.

Table 1: Execution time breakdown and memory requirements of the face recognition system running on Tegra platform

<table>
<thead>
<tr>
<th>Task</th>
<th>Time in secs. (% of total time)</th>
<th>Type of required data and size of memory requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face detection</td>
<td>1.5 (17.6%)</td>
<td>Face model: hundreds of KB</td>
</tr>
<tr>
<td>Landmark detection</td>
<td>0.7 (8.2%)</td>
<td>Eye model: tens of KB</td>
</tr>
<tr>
<td>Feature extraction:</td>
<td>5.1 (60.0%)</td>
<td>Gabor kernel: tens of KB</td>
</tr>
<tr>
<td>Gabor wavelet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature extraction:</td>
<td>1.0 (11.8%)</td>
<td>Pre-trained PCA/LDA matrix:</td>
</tr>
<tr>
<td>Dimension reduction</td>
<td></td>
<td>tens of MB</td>
</tr>
<tr>
<td>Feature classification</td>
<td>0.2 (2.4%)</td>
<td>Data base: up to hundreds of KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(if database has thousand faces)</td>
</tr>
<tr>
<td>Total</td>
<td>8.5 (100.0%)</td>
<td>~37MB</td>
</tr>
</tbody>
</table>

4. Tuning the algorithm parameters

The profiling results indicate that the two feature extraction steps account for more than 70% of the total processing time and over 90% of the memory requirements. One the other hand, although the memory requirement and computation time of face feature classification is low, it grows linearly with the number of training faces in the database. We therefore identify the tuning knobs in these three steps and explore the recognition rate and execution complexity trade-offs resulting from adjustment to these knobs: (1) selection of Gabor kernels, (2) selection of local patches, and (3) reduction of feature dimension.

Selection of Gabor kernels. As described in Section 3, a Gabor-based face descriptor involves convolving a face region with a series of Gabor kernels which consist of 5 different scales and 8 different orientations. Since Gabor wavelet transform is a time consuming process of the face recognition system, reducing the number of kernels could reduce the processing time. An experimental study of using selected subsets of these Gabor kernels is summarized in Figure 4(a). The results show, using all 40 kernels achieves a recognition rate of 92.85%. Using one half of the kernels (skipping orientations 1, 3, 5, and 7), the recognition rate drops to 85.04%. If we keep all orientations but skip kernels of scales 1 and 3, then the recognition rate drops to 87.32%. Skipping part of both orientations and scales results in a significant drop of accuracy to 79.48%. These results show that the recognition rate is highly sensitive to the choices of kernels and we likely need to use most of the kernels in order to achieve high accuracy.

Selection of local patches. Figure 3 shows the spatial partition of a face region into 20 non-overlap local patches. The computational complexity of this step and the required memory size for the transformation matrix is proportional to the number of local patches used for feature extraction. For desktop applications, all 20 local patches are extracted for classification.

It would not be surprising that certain regions are more discriminative, or more robust with respect to variation in facial expression, than others. For example, eyes, nose, mouth are easier to separate different people, while cheek are more similar for most people. On the other hand, although some patches are less discriminative by themselves alone, using them might complement other patches. We chose 10 different subsets of local patches, which are illustrated in Figure 3, to evaluate the recognition accuracy. For each case, the patches marked with oblique lines in the face are not extracted for recognition. The number of selected local patches is incrementally decreased, with Choice 1 having all 20 patches selected and Choice 10 having only 2 patches in the eye area selected. The specific subset of patches selected for each choice is based on the patches' discriminative power. The recognition results are plotted in Figure 4(b) in which the horizontal index corresponds to the index of the choice shown in Figure 3. It is interesting that, using only half of the 20 patches (i.e. Choice 5) results in an accuracy drop of only 1%. Figure 4(b) shows several different curves, each of which corresponds to a case with a different number of training samples per person. More training samples result in a higher accuracy and all curves reveal a similar accuracy trend with respect to these 10 choices of local patch selection.

![Figure 3: Ten different local patch combinations used in the exploration of algorithm tuning trade-off.](image-url)

Reduction of feature dimension. As the size of the face database increases (i.e. the total number of all training images), the time for feature classification as well as the required memory will increase accordingly. For resource-limited mobile embedded systems, it is desirable that the face feature is as compact as possible so that the memory and runtime requirements per training face are minimized. For each patch with a size of 16x16 pixels, its Gabor representation has a dimension of 16x16x40 = 10,240 when using 40 Gabor kernels. The feature dimension reduction step reduces its dimension to 120. Figure 4(c) shows the impact on the recognition accuracy for further reduction of the feature dimension. Five most discriminative combinations of local face patches in previous study (Figure 3) are used for this experiment. The result is also very promising: when feature dimension is reduced to one fourth of the original feature...
dimension (i.e. 30, instead of 120), the accuracy drops only 1.48%, from 92.42% to 90.94%.

4.1. Summary of algorithm parameters selection

We identify three parameters to adjust the face recognition algorithm’s performance. The recognition rate is highly sensitive to the number and choices of the Gabor kernels, which are used in the first stage of feature extraction. However, as this stage is the most compute-intensive, the overall execution time can be reduced if the number of Gabor kernels is reduced. The number and choices of local patches which are used in the second stage of feature extraction, affect the recognition rate as well. By properly selecting only one half of the local patches, the recognition rate drops only slightly (by 1%). As this stage relies on pre-trained PCA and LDA matrices for each local patch, the overall memory usage is reduced if fewer local patches are used. Finally, the reduction of feature dimension, which is used in the feature classification stage, has little impact on the recognition rate. However, this stage also has least computation and memory requirements, in comparison with the previous two stages.

Table 2: Summary of face recognition algorithm’s tuning parameters and their impacts on recognition rate.

<table>
<thead>
<tr>
<th>Parameter</th>
<th># of Gabor kernels</th>
<th># of local patches</th>
<th># of feature dim.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution stage</td>
<td>Feature extraction step (1): Gabor wavelet</td>
<td>Feature extraction step (2): dimension reduction</td>
<td>Feature classification</td>
</tr>
<tr>
<td>Impact on recog. rate</td>
<td>High (drop linearly)</td>
<td>Medium (drop slightly)</td>
<td>Low (drop only when feature dim. is skipped)</td>
</tr>
<tr>
<td>Impact on system req.</td>
<td>Reduce computation time</td>
<td>Reduce memory</td>
<td>Reduce both computation time and memory</td>
</tr>
</tbody>
</table>

5. Utilizing mobile GPU

A comparison of the commodity GPU cores inside various platforms is shown in Figure 5. The GPU embedded in a mobile platform has significantly less computing power, less memory bandwidth and lower power consumption. Another major difference is the programming API. OpenGL ES 2.0 [13] is the primary graphics library for handheld and embedded devices with a programmable GPU. The commonly used high-level API for a desktop environment, such as CUDA [14] or OpenCL [15], is not supported in the embedded platform yet. To utilize the mobile GPU as a general-purpose accelerator, programmers have to map the algorithms to the graphics domain, and writing the programs using the graphics API, OpenGL ES, to configure the vertex and fragment shaders.

In addition, OpenGL ES API exposes less controllability to the low-level hardware, and hence makes it less flexible to use the GPU for general purpose computing. For example, the graphics APIs in the current versions of OpenGL ES do not have the “scatter” operation (i.e. write to an arbitrary memory location), or thread-level synchronization.

Figure 5: Comparison of GPU cores in different platforms (1: the spec is not available in the public dataset. 2: The C1060 GPU inside a workstation is to perform computing, but graphics tasks. Therefore the GPlygon/s is not applicable. 3: the number is estimated from benchmark results of PowerVR SGX530 and Tegra 2 GPUs)

5.1. GPU-implementation of Gabor face feature extraction

From the profiling results shown in Table 1, we know that computing Gabor wavelets is the most time consuming task in the face recognition pipeline. Although reducing the number of Gabor kernels can reduce the total execution time, it also causes the recognition rate decrease dramatically as is depicted in Figure 4(a). Therefore, it is highly desirable to offload this compute-intensive task to another platform resource, the mobile GPU.

Gabor wavelet can be implemented by convolution or the Fast Fourier Transform (FFT). The convolution method is suitable only for small-size kernels due to the memory limitation.
However, a small-size kernel is not realistic for object and pattern recognition [18]. Therefore, our GPU-based Gabor face feature extraction is based on FFT method: first transforms both face image and the Gabor kernel into the Fourier space, multiplies them together, and then inverse-transforms the result back to the space domain.

In our study, we examined an implementation of the Cooley-Tukey FFT algorithm, based on the approach presented in [17], using OpenGL ES 2.0 API and the shader language. The processing flow of the Cooley-Tukey method for an 8 samples FFT is depicted in Figure 6. A total of $\log_2 N$ stages are required to complete the computation for $N$ samples. Samples in each stage form pairs between two groups, as shown in the dotted box of Figure 6. The computation of this group is expressed as: $c = a + w^0 \cdot b$ and $c = a - w^0 \cdot b$. The coefficients of each sample (i.e. $\pm w^0$) is pre-computed and stored as a texture for the shader program to fetch. It should be noticed that the sign of the coefficient is also included in the texture in order to avoid the conditional computation (e.g. branches) within the shader program.

In the face feature extraction, a 128x128 complex FFT is performed on face image and kernels. The transformation of a 2D image is done by applying 1D FFT to rows and columns consecutively. Each stage requires a two channels texture of size 128x1 to store the pre-computed real part and the imaginary part coefficients. Another texture storing the fetch indices is required. Although it could be combined with the coefficient texture, such implementation is wasteful because indices require a lower resolution texture than that required for coefficients. Therefore, we allocate a texture with a lower resolution for fetch indices to reduce the memory bandwidth. The iterative processing is implemented by multiple rendering passes, with a floating point framebuffer object (FBO) (Chapter 12 in [16]) to store the intermediate rendering results.

6. Experimental results

Our experiments were performed on a Nvidia Tegra SoC/board [6] with the following specifications: a 1GHz dual-core ARM Cortex A9 CPU, 1GB of RAM, a Nvidia GeForce GPU, and 512MB of Flash memory. The operating system is Android 2.2. The Tegra board is powered by a 15V DC input. Based on this input voltage, a regulator converts the voltage into 3.3V, 5V, 1.8V and 1.05V respectively. It is difficult to precisely measure the power consumption because it requires isolating the traces on the board that power the Tegra chip and measuring the current values. Also, because the CPU and the GPU are integrated in a single chip, it is not possible to measure exactly the current drawn by each individual core in the chip. Therefore, we approximate the current jointly consumed by the CPU and the GPU as the current consumed by the whole board. The Tektronix TDS 2014B Oscilloscope is used in the measurement. The average idle current is about 0.25A which is used as the level of offset current.

6.1. Feature extraction on mobile CPU and GPU

We compared two implementations of feature extraction: utilizing Tegra’s CPU alone and offloading Gabor wavelets to Tegra’s GPU. The measured power consumption results are shown in Figure 7. The blue line is the power consumption of a CPU implementation, and the red line is the GPU’s result. Both CPU and GPU start running roughly at time 2.2 second. After the CPU starts running, it takes about 1 second for the CPU to initialize the GPU and to transfer data from the CPU to the GPU before the GPU runs at its full capacity.

The measurement results show that, to extract the Gabor face feature, the GPU and CPU spends 2.25 and 6 seconds respectively. The GPU consumes slightly more power (4.7W) than the CPU (4.5W) since CPU is not idle when the GPU is running and is standing by for the completion of GPU. The total energy consumptions of the CPU and GPU for Gabor face feature extraction are 27.0 and 10.3 Joules respectively.

![Figure 6: Processing flow of 8 samples 1D FFT.](image6)

![Figure 7: Power consumption of the feature extraction on Tegra’s CPU and GPU](image7)
The first row of Table 4 shows the results without incorporating any algorithmic adjustment (i.e. using all 40 kernels and 20 patches) and ran the program on Tegra’s CPU. This baseline setting results in a total computation time of 6.1 seconds for face feature extraction. The second and third rows are two exemplar results of tuning algorithm parameters described in Section 3. Using a subset of kernels (Row 2) reduces the execution time from 6.1 second to 4.1 seconds, but the recognition accuracy drops from 92.8% to 85.0%. Using a subset of patches (Row 3) reduces the computation time as well as the total required memory used in the subsequent stage of feature dimension reduction. Since this stage contributes only 11.8% of the total computation time for face recognition, the contribution in execution time and energy consumption of this scheme is somewhat limited.

Table 4 further shows the comparison of performance and energy efficiency when combining the algorithm tuning and the utilization of GPU. The energy efficiency of the GPU-based implementation came from the dramatic reduction in execution time. The bold box shows the case that has the best energy setting without sacrificing too much recognition accuracy. In comparison with the baseline results, the first row of Table 2, this implementation successfully reduces the computation time by 71% (from 6.1 seconds to 1.7 seconds) and saves 70% energy consumption (from 27.45J to 8.12J), at the cost of 1.3% drop in recognition rate (from 92.8% to 91.5%) for this face feature extraction task.

7. Conclusion

In this paper, we present a case study of mapping compute-intensive computer vision applications to a smartphone platform with the goal of maximizing per-energy user experience. Our driving application is a high-accuracy face recognition system which employs Gabor face feature representation. A baseline implementation of the application on a Tegra platform takes 8.5 seconds to recognize a person. In this case study, we explore two directions for both performance and energy optimization: (1) tuning the algorithmic parameters and characterizing the accuracy-runtime-energy trade-offs, and (2) better utilization of available computational resource, specifically the mobile GPU, to alleviate the computation burden and improve energy efficiency. The implementation based on the best algorithmic configuration and GPU implementation, achieves 71% reduction in computation time and 70% saving in energy consumption for the most time-consuming task, face feature extraction, in our driving application. For the whole face recognition system, it achieves 2.1x speedup while saves 50% of the total energy consumption.

Table 3: Comparison of tuning algorithmic parameters - running on Tegra’s CPU

<table>
<thead>
<tr>
<th>Algorithm setting</th>
<th>Exec. time (s)</th>
<th>Energy (J)</th>
<th>Recog. rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using all kernels and patches</td>
<td>6.1</td>
<td>27.4</td>
<td>92.8</td>
</tr>
<tr>
<td>Kernel selection: 3 scale x 8 ori</td>
<td>4.1</td>
<td>18.4</td>
<td>85.0</td>
</tr>
<tr>
<td>Patch selection: combination 5</td>
<td>5.6</td>
<td>25.2</td>
<td>91.5</td>
</tr>
</tbody>
</table>

Table 4: Comparison of combining algorithm tuning and utilizing Tegra’s GPU

<table>
<thead>
<tr>
<th>Algorithm setting</th>
<th>Core selection</th>
<th>Exec. time (s)</th>
<th>Energy (J)</th>
<th>Recog. rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using all kernels and patches</td>
<td>CPU Only</td>
<td>6.1</td>
<td>27.4</td>
<td>92.8</td>
</tr>
<tr>
<td></td>
<td>CPU+GPU</td>
<td>2.2</td>
<td>10.3</td>
<td>92.8</td>
</tr>
<tr>
<td>Kernel selection: 3 scale x 8 ori</td>
<td>CPU Only</td>
<td>4.1</td>
<td>18.4</td>
<td>85.0</td>
</tr>
<tr>
<td></td>
<td>CPU+GPU</td>
<td>1.7</td>
<td>8.02</td>
<td>85.0</td>
</tr>
<tr>
<td>Patch selection: combination 5</td>
<td>CPU Only</td>
<td>5.6</td>
<td>25.2</td>
<td>91.5</td>
</tr>
<tr>
<td></td>
<td>CPU+GPU</td>
<td>1.7</td>
<td>8.12</td>
<td>91.5</td>
</tr>
</tbody>
</table>

8. References