computer graphics tübingen

Where's Waldo: Matching People in Images of Crowds

Georg Volk, Stefan Hettich

August 26, 2015

Praktikum Massively Parallel Computing SS2015
Wilhelm Schickard Institute of Computer Science

Where's Waldo: Matching People in Images of Crowds

Georg Volk, Stefan Hettich

Abstract

This paper describes the Cuda implementation of an algorithm to match people in images of crowds, based on the work of Garg et al. [RGS11]. The basic problem is explained and the used methods from the original paper are presented. Then the derived parallel GPU implementation is discussed. Throughout the paper the most complex and relevant kernels are explained as well as some modifications to improve the performance of the parallel solution. The result of this work is an extremely fast solution for finding people in images of crowds in comparison to a naive CPU implementation.

1 Introduction

This work handles the problem of matching people in different images of crowed places and events. That means images which were taken at the same place (weddings, public places, etc) and at almost the same time (only a few hours later or earlier). In a set of photos from the same place and almost the same time, there are several persons who appear twice or more often. The challenge is to find all appearances of one specific person in all images. The main problem is similar to the popular children's book "Where's Waldo".

But the searching for Waldo in a set of natural photos is even more difficult, not for an human but for a computer program. The program only sees a set of images with some colored pixels in it, at the beginning there are no more information. An human is able to detect other information from the image like buildings or other prominent persons. So it also necessary to make these contextual information available to the program. These information extend the Where's Waldo problem from a simple searching problem to a more complex problem.

Computers and especially programs are better in detecting geometric structures but really bad in detecting nonrigid object like persons. So factors like buildings in the image scene and the contextual information mentioned above like GPS tags, time stamps and things like that should make it easier to find the nonrigid people.

An other problem is that people don't always stand or sit still, so even if it would be possible to detect the area where the person was in image A it's not sure that the same person is still in this area in image B, which for example was taken an hour later. To make to problem tractable it's necessary to make the assumption that people don't move that much and it's alright the search in the found area for the person.

So this problem has nothing to do with face-detection, we try to find people in a crowd of hundred people, where face-detection is not able to achieve good results. We try to work with some contextual information but in the end it's also necessary to do some pixel to pixel comparisons. If we think of an image from a modern camera it's not only about a few pixels there is a huge amount of pixels to compare.

Comparing pixels is not a difficult problem but a problem that has to be solved a hundred or a thousand times. So the problem can be solved in a (massively) parallel way. This is the reason why CUDA (Compute Unified Device Architecture)[1] an architecture from NVIDIA is perfectly suitable to solve the problem above.

CUDA allows the user to use the GPU for computations. The GPU is a dedicated super-threaded, massively data parallel co-processor which is optimal for easy-to-solve problem which has to be computed several times. CUDA makes it possible to launch batches of threads on the GPU, all these threads use the same code to solve the same problem but with different data. In general SIMD (single instruction multiple data) problems are ideal for the GPU. CUDA also ships driver for loading the written computation programs into GPU, and allows explicit GPU memory management. (c.f. [Len15])

[1] `http://www.nvidia.de/object/cuda-parallel-computing-de.html`

2 Related Work

This work is based on the paper "Where's Waldo: Matching People in Images of Crowds" from Garg et al. [RGS11]. There the basic idea of matching people in different images is explained. We extended this approach and adapted it so it can be implemented on the GPU to drastically shorten the computing time of the described algorithms.

To identify a person Grag et al. have chosen section in an image, where the person they want to find is located. This section is then used as input for a pixel based logistic regression classifier. As classifier they used a one similar to the one described in [DRZ07]. Afterward the classification is weighted with a gaussian blur of the areas which were specified to mark the different body parts of Waldo. So they specified a weight so that the pixel inside for example the area of a persons torso is weighted more than a pixel on the outer side of its torso which may interact with the environment.

To improve their results they also took the time the different images were made into consideration. So they were able to gain better results as people tend to stay at the same location in a short time interval. Furthermore they also used the hypothesis that persons that appear in groups also appear in the same group in an other image.

In this paper we focus on the appearance model and classification of image sections showing Waldo. Time intervals as well as groups of people are not considered as we spend most of our effort in making the presented approach more efficient with a massively parallel implementation on the GPU.

3 Naive solution approach

At first a set of images to analyze is necessary. It wasn't possible to get the original dataset mentioned in the paper from Garg et al. [RGS11]. But a set of many images from the same place and taken at almost the same time is ideal for the problem. So images from a public place where many people are at the same time was chosen. In our case a set of images from Trafalgar Square in London was taken, like the one used by Garg et al.

The images which should be compared must be loaded into the program and one of these images will be the training image. Waldo (the person we want to find in all the other pictures) will be marked. Ideally the top and the bottom point of the person will be marked to specify the area where the person is in the image. Further the torso of the person will be marked with three different areas, these areas will mark body, legs and head of the person. (see Figure 1 (b)).

The data we got from marking our Waldo will be saved to a file, so that it is available in later usages of the program. The image section of the Waldo will be saved in a separate image file. Also the marked areas will be saved to an image file with the same size of the image section, the three areas will get the colors red, green and blue. This image will be smoothed with an Gaussian blur to reduce inaccuracies coming from bad user input. (see 1 (c)).

Figure 1: User Input and Appearance Model [RGS11].

At this point all reference data we need from the marked Waldo is there. Now it's necessary to build a 3D scene of all images so that they stand in a relation to each other. The main challenge is to find the camera position of each image as exact as possible. With the help of this information it is possible to reference a point from image A to image B. This makes it possible to find the image section of the marked Waldo in the image we want to search for Waldo.

Now we are able compare two image sections, the one containing Waldo (image A) and the one where Waldo should be (image B). With the help of an appearance model and a pixel based logistic regression classifier it should we are now capable to score a possible match. The appearance model generates a 9D feature for each pixel in both image sections. The features are the values red, green, blue (r, g, b) and their combinations (rr, gg, bb, rg, rb, gb). The score of a match is the sum of all positive classified pixels with the parts of the marked areas are scored higher than the pixels outside this areas.

At the end we decide on the basis of a percentage we get from the classifier, if the Waldo is in the

image section if image B or not. For more information about the appearance model and the logistic regression classifier please see 4.4 and 4.5, where the implementation and the concept of these methods is described in detail.

4 Description of the Solution

To make the program easy to use, we decided to implement a graphical user interface where the user of the program is able to load all the images he wants to compare, mark Waldo and start the searching and matching in all the other images. He is also able to save his work an load it for later use.

4.1 Marking Waldo

It's necessary to load all the images to compare into the graphical user interface of the program. The user selects one of the images and marks the person he want to find in the other images.

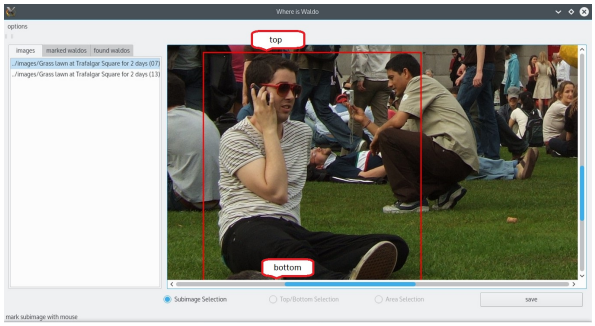

Figure 2: Graphical User Interface; Marking Waldo.

The user has to select the concrete image section Waldo is in and in the next step marks top and bottom point of the person (see figure 2). After that the user marks three areas in the image section. The first area (red) is the most important section and usually the body of the person, the second area (green) are the legs of the person and the last and least important one (blue) is the users head (see 3). We decided this order, because the shirt color (body) is usually the most important one to identify the person. In general the head might be more important than the legs, but we recognized that the face of the person often has a very different color in an other picture because of different shadows in the face. The legs however have a more consistent color in all images and offer a larger area of same coloring resulting in a better feature for the logistic regression classifier.

Figure 3: Graphical User Interface; Marking Waldo Areas.

After the user has marked Waldo and the corresponding areas he is able to save the marked training data for later usage. We decided to save the data to a simple json file. To save data in C++ to json can be complicated, so rapidJSON[2] was our choice for a json parser and generator. RapidJSON makes it possible to simple load json data from a file to an object, which makes the data access very easy. To save data we must at first initialize the object and set the data to save to the object and finally save to object to a file.

4.2 Gaussian blur

When all user input data is saved either to image file or json file it is necessary to smooth the image with the marked areas (body, legs and head) as seen in figure 4. This is important to increase the quality of the input data, because inexact marked areas could cause inaccuracies in detecting Waldo in other pictures. The resulting smoothed image is used as the weight for the later classified pixel, when doing the match of two image sections where the searched Waldo possibly is located.

The formula for Gaussian blur $G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$ is used to calculate a 9x9 dimensional matrix with the values $\sigma = 2$. The matrix is computed on the CPU because this step has to be done only once. The Gaussian blur matrix is put over every point and the new color value of the point is

[2] https://github.com/miloyip/rapidjson

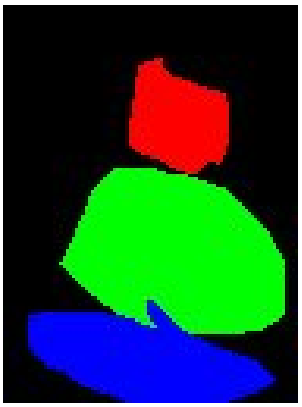

Figure 4: Image of marked areas before Gaussian blur.

calculated by including the values of the neighboring points. That's the reason the final pictures gets smoothed. To smooth the picture like mentioned in the paper from Garg ([RGS11]) we decided to run the Gaussian blur 50 times on one image to get a result like in figure 5.

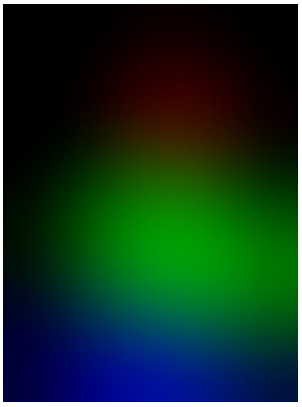

Figure 5: Image of marked areas after Gaussian blur.

The calculation of the Gaussian matrix is done on CPU like mentioned above, but the calculation for each pixel of the image is done on GPU, the details will be described in section 5.1 below.

4.3 3D Localization

When all the work on the reference image is done it comes to comparing the other images with the reference images with the marked Waldo. Like described in the paper from Garg -[RGS11], it is very important to find a position in the image where Waldo might be. Comparing pixels in the whole image would be to much overhead and also very error-prone, because similar color pattern can appear in other parts of the picture and cause mismatches.

So it is necessary to find an image section where Waldo might present and after that only search in that section. That means there are a lot less pixels to compare and also a smaller chance for a mismatch.

All the images from the set are taken at the same place and nearly the same time, so it is possible to create a 3D scene of all pictures, by detecting architectural buildings in each of the images and calculating the possible point of the camera which has taken the picture for each image. There are several popular implementations which solve this problem, so we decided to use one of them and didn't try to implement this on our own. We've chosen the tool VisualFM[3]. VisualFM allows us to automatically generate the needed 3D scene with the specified set of pictures. It creates an output nvm file with all needed information like focal length and other camera parameters. Usually VisualFM is not able to place all the images in the scene, because some images have to less information needed for a proper 3D reconstruction. In our image set of initially 39 images, VisualFM was able to set 17 of them in a very exact 3D scene.

The VisualFM step must be performed before using our program. In our program we just read the output nvm file with the help of a camera script from Prof.Dr. Hendrik Lensch and his team from the University of Tuebingen. With the help of this script we are able to project a point from image A to its corresponding point in the constructed 3d scene on image B.

4.4 Appearance Model

To be able to identify a person correctly in two images, Garg et al. in [RGS11] used an appearance model for the specified person which shall be found in other images. This appearance model then builds the foundation for a logistic regression classifier which is able to compare two images and decide whether Waldo is correctly identified in the corresponding image section or not.

[3] `http://ccwu.me/vsfm/`

Building a reliable Appearance model is really difficult as the poses of persons vary on different images therefore Garg et al. used an appearance model earlier described in [FH05]. This appearance model we also used for our approach. The appearance model consists of two coordinates Pi_{head} and Pi_{ground} and three areas as seen in figure 1. These two coordinates specify where the person touches the ground and the top of the persons head. The coordinates are used to calculate the corresponding area respecting the 3D relationship between two images[RGS11].

For each area which was marked by the user a separate pixel based logistic regression classifier is trained. A normal picture normally only has three values per pixel, which are red, green and blue varying from 0 to 255. So this classifier would have only a three dimensional feature. Hence also all quadratic combinations of the r,g and b values (rr,rg,...) are used to build up a 9D feature for classification[RGS11]. If a Pixel is inside the specified part the corresponding feature of this pixel is labeled positive ($y_i = 1$) otherwise the feature is labeled as negative sample ($y_i = 0$). So the classifier is trained to detect the areas of a person.

The scoring of a corresponding image section is done with the gaussian blur image. Each pixel classification is weighted with the gauss value of the same pixel in the gaussian blur image. As the gaussian blur image and the testing image are scaled to the same size these weights can easily be calculated. Because three different areas are specified, this results in three classifiers. To get the overall score of a corresponding image section all three classifiers have to be considered. Therefore each classifier has a different weight. The weights for the two most important areas are both 0.4 and for the least important area 0.2. These values were found by several test runs as the torso and the legs were most times equally good to identify a person, because they offered a high amount of true positive classifications. The least important part which in our approach always was the head has only a weight of 0.2 because it had really high false positive rates.

To get the final confidence value of Waldo on an image section the confidence values of each classifier have to be multiplied with the corresponding weights α_i and summed up (see formula 1). The confidence value are simply the probabilities that Waldo is found. If the probability is above the specified threshold of 0.5 the algorithm expects that Waldo is found and marks the image and the area were Waldo is found.

$$pWaldo(feat) = \sum_{i=0}^{numAreas} classifier_i(feat) \cdot \alpha_i \tag{1}$$

4.5 Logistic Regression Classifier

With logistic regression the influence of multiple independent variables on a dependent variable are modeled. Therefore a weight vector β has to be defined which best reflects the dependence of multiple independent variables to the dependent one[BS15].

For logistic regression typically we have our independent variables, which in our example are our samples X. The samples are the single pixels of the image section. x_i then represents the i-th pixel of our image section, where the samples are stored in a 1D-Array, beginning with the pixel at the top left. The dependent variable is Y which in our case is the label of each pixel. So y_i defines whether pixel i is inside or outside an area. With a weight vector β now the dependence between X and Y shall be modeled. Finding the best vector β represents the training problem of our logistic regression classifier[Den98, RGS11].

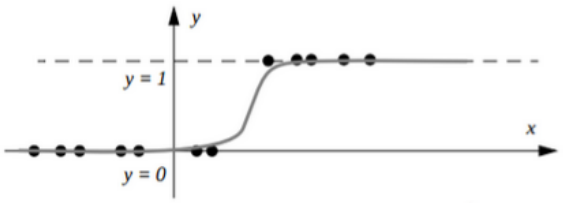

Figure 6: Logistic regression function used for the classifier[Den98].

In figure 6 a typical logistic regression function is plotted. The dotted points represent the different samples. Based on their position on the logistic regression function it can be stated if the sample belongs to class $y = 0$ or to class $y = 1$. The formula to find this function is the following as also stated in [Den98]:

$$\pi_i = \frac{1}{1 + exp(-(1, x_i^T)\beta)} \tag{2}$$

This formula evaluates the value of the logistic regression function with weight β for sample x_i. So according to the value of π and a defined threshold we are now able to evaluate if a sample belongs to class $y = 0$ or $y = 1$.

But still we have not covered a efficient way how to calculate a good β. This will be solved in the following. The basic idea is that we choose an initial beta and then update it iteratively until it reached a maximum and does not change anymore.

Listing 1: Pseudo code for finding a good beta.

```
choose initial beta
while gradient changes:
 for i in Pixels:
  calculate pi
   for k in FEATURE_LEN:
    gradient[k] =
        pixel_i *(label - pi)
 for k in FEATURE_LEN:
  beta[k] +=
     LEARN_CONST * gradient[k]
```

Code listing 1 illustrates the basic algorithm to find a good β. This algorithm was taken from [BS15] and represents a gradient descent approach. As first step an initial beta is chosen. Then we iterate over all pixels and calculate the value π_i for each sample with the current β. Then we can extract the gradient which simply is the value of each feature value multiplied with the difference of the label of this feature and the predicted label π_i. After we have iterated over all pixels and have defined our gradient respecting all pixels we can update our β. This is done by multiplying the gradient with a previous defined learn constant and finally adding it to the beta. The learn constant is necessary because it guarantees that the optimal beta can be found[BS15]. On the one hand if we chose a learn constant which is too big it may happen that the optimal value for β can not be found. On the other hand if the learn constant is too small the algorithm takes much longer to converge. In our implementation a learn constant of value 0.001 was chosen. This procedure is repeated until the gradient does not differ more than a defined threshold from the gradient of the previous run. So finally an optimal beta is found.

However in our application this approach was not completely realizable. The problem was that the gradient did not converge to the defined threshold because the calculated probabilities and therefore the labels in each run differed very much. This may have happened because we have a really low dimensional feature on the one side and on the other side we have a really high number of training samples for the logistic regression classifier. So we decided to iterate over all pixels for updating our β a specified number of epochs. For each iteration we remember the calculated β and the achieved percent correct. In the end we chose the best β. After multiple test runs we found that 1000 epochs were enough to find a good value for β. So within 1000 epochs we were able to find a β to correctly classify 80% of the pixel on the training image.

5 GPU Implementation

For the GPU implementation two main parts of the previous image matching were chosen for parallelization. On the one hand the gaussian blurring and on the other hand the training and prediction of the logistic regression classifier were implemented on the GPU. These two parts were chosen because they offer the greatest benefit from parallelization. The implementation details will now be covered.

5.1 Gaussian blur

Like mentioned in section 4.2, some parts of the Gaussian blur algorithm are running on the CPU. That means that all the needed data like input image array, output image array and Gaussian matrix must be copied to GPU. After that the kernel for Gaussian blur can be executed for every point of the initial image.

The implemented variant simple looks at the 3 neighboring pixels in each direction (to get a 9x9 dimensional matrix) and multiplies their values with the value of the matrix at this position and finally adds up all values to get the new value for the edited pixel. The new value is written back to the result array at the pixel position.

The Gaussian Blur wasn't the main approach of our work and it is only used when marking the Waldo in the reference image, so we decided to keep this part as simple as possible. We didn't put to much work on optimization of the GPU kernel. Although there were many points to optimize, like calculating the 9x9 dimensional matrix on the GPU or keep the input pixel values of the old picture on the GPU. But when working with a normal image

size and a normal pixel count the kernel is finishing in a very short time, so there wouldn't be to much time effort. As we decided to do the Gaussian blur 50 times it took some time, because the program is uploading the data to GPU and after the kernel performance downloading it again to CPU in each iteration. So most of the time GPU and CPU are working on up- and downloading data.

5.2 Logistic Regression Classifier

The logistic regression classifier consists of two parts. In the training step the values of β are calculated with a training image section. In the prediction step the labels of every pixel are calculated for the given image section based on a previous calculated β.

5.2.1 Training

In the naive approach for finding an optimal β, described in 4.5 we iterated over all pixels of the training image section one after the other and updated our gradient. After that the β is updated with the calculated gradient. This procedure is repeated n times, where n is specified by the number of epochs, which in our case is 1000.

For parallelizing this algorithm we first need to copy the image section to the GPU memory. On the GPU the image is represented by a float array. Each pixel is represented by its corresponding 9D feature. After the image is present on the GPU instead of iterating over all features separately we wrote two kernel functions such that the features can be processed in parallel. The first kernel calculates the gradients for all features in parallel and then a reduce kernel is called for reducing all gradients calculated for all features to one gradient to finally update the value of β.

To calculate the the gradients for the features in parallel the problem has to be split up. Therefore the number of features is divided by number of allowed threads per block, which we set to 512. This results in a number of blocks. So the feature array is split up in slices with size $512 \cdot 9 = 4608$, because each pixel has a 9D feature vector. So each CUDA block gets the indices of the feature slice it has to work on. Within each block a CUDA thread calculates the gradient for one feature. So he iterates of the 9D features, predicts the class corresponding to the current β and finally writes the calculated gradient into GPU memory. To speed up the computation the current β values are loaded into shared memory such that they have only to be read once. This reduces the memory accesses as all threads within one block now only have to read β once. Unfortunately this optimization is not possible for the feature values, as they only have to be read once.

After one iteration we now end up with a number of gradients which is equal to the pixels within an image section. These gradients are in a continuous area of memory on the GPU. The one thing left we have to do is summing all gradients up. An efficient way of calculating sums on the GPU with CUDA is using reduction. With reduction each CUDA thread calculates the sum of two values. So in each iteration the values which have to be summed up is halved. So instead of needing n iterations only $log(n)$ iteration steps are needed. For detailed information on reduction see [Len15].

We have very much values, which have to be summed up, as we take an image section with a width of 576 and a height of 594 there are 342144 pixels. Each pixel generates a 9D gradient which finally results in $342144 \cdot 9 = 3079296$ values to sum up. Because of this enormous size three reduction steps are necessary. Because in two reduction steps it is not in all cases possible to sum up all values as the maximum number of blocks is limited. In each reduction step the reduce kernel has to be called nine times, because each attribute of the 9D gradient has to be summed up. Therefore the kernel gets the gradient index so he knows on which of the nine values of the gradient he is working and the correct index can be calculated. In the first reduction step the values are reduced to the number of blocks. The number of blocks is calculated similar as before by dividing the number of pixels or features by the number of threads per block. So after the first reduction each CUDA block has summed up its values. The resulting values are written to a different block of GPU memory because the resulting temporary β values of the training kernel are needed in the next iteration epoch. This different block of GPU memory also is written continuous, so there are no gaps and the next reduce iteration can take place. In the second iteration we reduce the number of values to sum up beneath the number of threads per block. So in the last reduce iteration all values fit in one block and we end up with the

summed up gradient.

Now this whole procedure is repeated as in the iterative approach number of epoch times. But as we use in each step the calculated betas from before no memory transaction between CPU and GPU is necessary. Only in the end after all epochs have been iterated the final beta has to be copied back to the host.

5.2.2 Prediction

For prediction we have to evaluate each 9D feature with the previous calculated weight β. First all features have to be copied to the GPU. To parallelize the computation efficiently the problem, respectively the features, is once more split up in slices with size of 512 features. So each CUDA block processes 512 9D features. Within each block a thread computes the probability that feature x_i belongs to class 1. To enhance the speedup, once more the β values are loaded into shared memory that each block only need to read them once out of the slow global GPU memory. Once more only the β values can be loaded into the shared memory, because one feature value is only read once by its corresponding CUDA thread. Hence this memory access can not be improved with shared memory. After each CUDA thread has evaluated the formula 2 it is checked whether the probability is above 0.5 or not. Then the predicted label for the corresponding feature is written to global GPU memory.

After all CUDA threads have finished their computation the result array with the feature labels on global GPU memory space is filled completely. This array has the length equal to the number of input features. So finally this array of integer labels, has to copied back to the CPU and the prediction step for the classifier on the GPU is finished.

6 Pitfalls

In this section we will highlight the main pitfalls that occurred during the implementation. There where three major ones which will now be explained and the solution to resolve this problems is illustrated.

6.1 Normalizing Features

The first big problem was that the feature data led to very bad classification results. No matter how the algorithm was adapted to calculate a good weight β the results were equally bad. To gain better results in classification with logistic regression [BS15] highly recommends to normalize the features before training the classifier. Normalizing is done with the following formula as proposed by [BS15]:

$$normalized\_feature_{i,j} = \frac{feature_{i,j} - mean_j}{max_j - min_j} \tag{3}$$

The indice i stands for the feature or respectively a single pixel, indice j stands for one of the nine feature attributes of a pixel. With the normalized features the previously presented algorithm now works properly and computes consistent and stable results.

6.2 Gradient descent

An other problem we had to face was that the proposed gradient descent algorithm from [BS15] was not applicable to our specific problem. The gradient descent method represents a fast and good way to calculate the weights β. But with our test data the gradient descent method did not converge. The problem was that the fluctuation of percentage for classification in each round of training were very strong. So the resulting gradient was not getting below the defined threshold and the algorithm did not converge. This problem might come from the really low dimensional features and the very big amount of training data. So to solve this problem one could either produce more dimensional features or find an other way to calculate the β vector.

We decided to use a fixed amount of training iterations, we call them epochs, instead of reducing the gradient with each iteration. In each epoch we remember the calculated β and the percent correct which is achieved. In the end we chose that β which produces the highest percent correct. After several training sessions we found that within 1000 epochs a good value for β could be found. Our approach resolves the problem that the gradient descent method did not converge.

6.3 Reduction

The last pitfall is represented by the reduction algorithm as explained in [Len15]. Here two steps of reduction were used to build the final sum. But for the presented problem two reduction steps are not enough. It can occur that the feature array is too long and so in the first step the values can not be

reduced as much to fit all in one block. But it is necessary that in the last step of reduction all values are present in one block such that the final sum can be calculated.

7 Evaluation

In figure 7 three image sections are illustrated. Image a) in figure 7 shows the section with which our program was trained. Image b) and c) show reference images on which the classification algorithm is tested. Image b) represents a positive classification example for the marked Waldo in a) and Image c) is a negative classification example.

Figure 7: Image section a) is the training image, b) is a positively classified Waldo and c) is an example of a negative classification.

On this test data our classification algorithm performed quite good. On the same image we get an probability of 80% that the image section contains Waldo. The positive classification sample b) resulted in an 55% confidence that Waldo is present. So here Waldo was actually found. On the negative sample c) our algorithm computed a confidence of 34% that Waldo is present. So the output of our program was that here Waldo was not found. Also in this example our algorithm was able to correctly classify if Waldo was present or not.

Also in our test data set containing the images of Trafalgar square our program was able to find the marked Waldo from picture a) in figure 7. So we have shown that our implementation is not only capable of comparing two already prepared image sections, but also to project the coordinates of the image where Waldo is marked correctly on the other images, such that a comparison results in a positive classification.

As we have shown that our approach is capable of finding a marked person in a set of reference images, we will now compare the performance of the GPU version to the CPU version of the classification algorithm.

The performance tests where made on a PC with 16gb RAM, an Intel Xeon E3-1230 as CPU and a GTX 600 as GPU. As reference image a image section with width 576 and height 594 was taken. So this results in 342144 pixels which have to be processed for training the logistic regression classifier.

	Training (1000 epochs)	prediction
CPU	166.6s	0.058s
GPU	16.8s	0.013s
GPU speedup	9.92	4.46

Table 1: Runtime comparison of GPU and CPU implementation.

In table 1 the runtime of the CPU and GPU version for training and prediction is illustrated. The test runs were repeated 20 times and the average of all test runs was taken to compensate the influence of normal system workloads on the test system. As can be seen the GPU version offers a speedup of 10 in the training of the classifier and nearly a speedup of 5 in the prediction step. Due to the GPU implementation a remarkable speedup is gained and it is possible to process a much bigger data set of images in the same time.

8 Future Work

In future research it may be possible to improve the classification algorithm such that shorter computing times can be achieved as well as the probability of classification increases.

One way of improvement would be to gather better test data, this could be achieved by having higher dimensional features such that the logistic regression classifier is able to fit the training data much better. Possible feature attributes could be for example the brightness of a pixel. An other way of improving the classifier would be to take an other classification method than the proposed logistic regression classifier from [RGS11]. Instead of the logistic regression classifier for example convolutional neural networks could be used as these are currently state of the art in image processing as stated by [AKH12]. Convolutional neural networks would probably result in a better classification. Instead of improving the features or changing

the classification algorithm also other parameters like the time the photo was taken can be considered. This approach was already proposed by Garg et al.[RGS11]. So the information that a person moves only very little if two photos are made at nearly the same time can be used to improve the search area for this person.

But not only the performance of the classification can be improved. As we focused only on the main parts classification and Gaussian blur for parallelization other parts may also benefit from a GPU implementation. So due to parallelization of other parts this approach would propably gain an additional speedup. Furthermore the implementation of the Gaussian blur is not yet optimal as our main focus was the logistic regression classifier, so here also improvements especially in reducing the used memory transactions can be made.

9 Conclusion

In this paper we have shown that the approach of Garg et al. [RGS11] can be efficiently implemented on the GPU with the framework CUDA. Due to the GPU implementation a runtime speedup up to the factor of 10 can be achieved. So it is now possible to process much bigger datasets of images for people matching in the same time. But the presented solution is not only faster but also reliable and performed very good on the Trafalgar square dataset. The algorithm was able to find a marked person in the set of different images, of which some images contained the person and others not.

Additionally further possible improvements for the presented solution were shown. So this paper builds the basis for efficient search of people in images of crowds on the GPU.

References

[AKH12] I. Sutskever A. Krizhevsky and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, 2012.

[BS15] F. Botschen and A. Schilling. Machine learning. lecture notes, 2015.

[Den98] K. Deng. Omega: On-line memory-based general purpose system classifier. pages 59–76, 1998.

[DRZ07] D. A. Forsyth D. Ramanan and A. Zisserman. Tracking people by learning their appearance. In *IEEE Trans. on Pattern Analysis and Machine Intelligence*, pages 65–81, 2007.

[FH05] P. F. Felzenszwalb and D. P. Huttenlocher. Pictorial structures for object recognition. In *Int. J. of Computer Vision*, pages 55–79, 2005.

[Len15] H. Lensch. Massively parallel computing. lecture notes, 2015.

[RGS11] S. M. Seitz R. Garg, D. Ramanan and N. Snavely. Where's waldo: Matching people in images of crowds. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 1793–1800, 2011.