Fast Human Detection with Cascaded Ensembles on the GPU

Berkin Bilgic (Berthold K.P. Horn and Ichiro Masaki) 2010 January 11
Background

- There have been over 200,000 pedestrian fatalities in the past 30 years in the USA.
Background

- There have been over 200,000 pedestrian fatalities in the past 30 years in the USA.
- Driving assistance systems may reduce this number, if pedestrians can be detected accurately and in real time.
Background

- There have been over 200,000 pedestrian fatalities in the past 30 years in the USA.
- Driving assistance systems may reduce this number, if pedestrians can be detected accurately and in real time.
- The Dalal-Trigss algorithm exhibits high accuracy (rate of false positives at a given detection rate).
Background

- There have been over 200,000 pedestrian fatalities in the past 30 years in the USA.
- Driving assistance systems may reduce this number, if pedestrians can be detected accurately and in real time.
- The Dalal-Trigss algorithm exhibits high accuracy (rate of false positives at a given detection rate).
- But Dalal-Triggs cannot run in real time — it takes about a second to process even a $320 \times 240$ image.
Background

- There have been over 200,000 pedestrian fatalities in the past 30 years in the USA.
- Driving assistance systems may reduce this number, if pedestrians can be detected accurately and in real time.
- The Dalal-Trigss algorithm exhibits high accuracy (rate of false positives at a given detection rate).
- But Dalal-Triggs cannot run in real time — it takes about a second to process even a $320 \times 240$ image.
- The algorithm can work much faster if it is formulated as a cascade of rejectors.
Background

- There have been over 200,000 pedestrian fatalities in the past 30 years in the USA.
- Driving assistance systems may reduce this number, if pedestrians can be detected accurately and in real time.
- The Dalal-Trigss algorithm exhibits high accuracy (rate of false positives at a given detection rate).
- But Dalal-Triggs cannot run in real time — it takes about a second to process even a $320 \times 240$ image.
- The algorithm can work much faster if it is formulated as a cascade of rejectors.
- The algorithm can work much faster if parts of it are ported to a GPU.
The Dalal-Triggs Algorithm

Step 1
- Compute gradient magnitude and orientations
- Discretize orientations into bins

Step 2
- Vote into small cells according to each pixel’s orientation bin
- Locally normalize blocks that consist of these cells

Step 3
- Concatenate the block histograms into one big window descriptor
- Classify each window with a linear SVM
The Cascade of Rejectors

- Dalal-Triggs algorithm shifts the detection window over the image, making the same amount of computations for each window.

- We can rapidly reject windows that do not resemble a person, and focus on the harder ones with a cascaded classifier:

  ![Diagram](image)

  - A window needs to pass through all stages to be classified as a positive. If it is rejected at any stage, then that window is discarded.

- Each stage in the cascade is an ensemble of simple features (base learners):

  \[ s(x) = \text{sign}\left\{ \sum_t \alpha_t f_t(x) - T \right\} \]

- And the features \( f_t \) are chosen out of a feature pool of all histogram blocks defined in the detection window.
Cascaded Detector on CPU

1. Acquire image
2. Downscale image
3. Compute gradient magnitude and orientations
4. Compute integral histograms
5. Obtain block histograms
6. Evaluate cascade stages
7. Non-maxima suppression
8. Display results
Evaluating the Features

- The block histogram features $f_t$ are formed by concatenating and normalizing $2 \times 2$ cell histograms.

- Each pixel’s gradient orientation is divided into 9 bins. The cell histograms are the sums of gradient magnitudes belonging to these bins:

- By using 9 integral images to represent the gradient magnitudes in each bin, cell histograms can be efficiently computed.

- This is because the sum of pixels within a rectangular region can be obtained with four array accesses to the integral image $I_{\text{int}}$:

$$
\sum_{a<i\leq b, c<j\leq d} I(i, j) = I_{\text{int}}(b, d) + I_{\text{int}}(a, c) - I_{\text{int}}(a, d) - I_{\text{int}}(b, c)
$$

$$
I_{\text{int}}(x, y) = \sum_{i \leq x, j \leq y} I(i, j)
$$
Training the Cascade

- Each new stage in the cascade is trained with the false positives of the current cascade classifier. Hence, stages get more complex.
- We need to determine which features should be used together at each stage.
- The AdaBoost algorithm selects the features that compliment each other to achieve maximum false positive rate $f_{max}$ and minimum detection rate $d_{min}$ for the stage ensembles.

Our cascade reaches $(f_{max} = 0.99)^{23} \approx 80\%$ detection rate at $(d_{min} = 0.65)^{23} \approx 5 \cdot 10^{-5}$ false positives per window with 23 stages.
Algorithm: Training the cascade with AdaBoost

User selects values for $f_{max}$, the maximum acceptable false positive rate per stage, $d_{min}$, the minimum acceptable detection rate per stage and $F_{target}$, target overall false positive rate.

$Pos$: set of positive samples (INRIA training positives)
$Neg$: set of negative samples (sampled from INRIA training full-size negatives)

**Initialization:** $i = 0, D_i = 1.0, F_i = 1.0$

**while** $F_i > F_{target}$

$i = i + 1, f_i = 1.0$

**while** $f_i > f_{max}$

- Train 125 randomly sampled linear SVMs using $Pos$ and $Neg$
- Add the best SVM into the ensemble with the appropriate vote determined by AdaBoost
- Update weights of the examples in AdaBoost manner
- Evaluate $Pos$ and $Neg$ with the current ensemble
- Decrease the threshold $T_i$ until $d_{min}$ holds
- Compute $f_i$ under this threshold

$F_{i+1} = F_i \times f_i$

$D_{i+1} = D_i \times d_{min}$

Empty set $Neg$

**if** $F_i > F_{target}$

- Evaluate the current cascaded detector on the set of full-size negatives and add any false positives into $Neg$, subsample if necessary.
Cascade Detector on the CPU

- Our detector works about 15 times faster than the Dalal-Triggs algorithm and has similar accuracy:

<table>
<thead>
<tr>
<th>Detectors</th>
<th>Dense scan 320x240 image</th>
<th>Dalal-Triggs</th>
<th>7 sec</th>
<th>Our cascade</th>
<th>475 ms</th>
</tr>
</thead>
</table>

- But dense scanning a 1280×960 image even with this speed up takes 5.5 seconds.
Improving Performance with CUDA

- NVIDIA’s CUDA programming model enables the Graphics Processing Units (GPU) to execute thousands of threads in parallel and make more than 1 TFLOPS/s operations.
- We utilize this model to accelerate,
  - integral image computation and
  - cascade stage evaluation
Exploiting Parallelisms with GPU

- We compute the integral image in three steps:
  i. Scan image rows with the Parallel Prefix Sum algorithm [3]
  ii. Transpose the image
  iii. Scan the rows of the transposed image

- Cascade stages are evaluated with two levels of parallelism:
  i. Classify detection windows in parallel
  ii. Compute the features in each stage in parallel
Cascaded Detector on CPU

1. Acquire image
2. Downscale image
3. Compute gradient magnitude and orientations
4. Compute integral histograms
5. Obtain block histograms
6. Evaluate cascade stages
7. Non-maxima suppression
8. Display results
CPU

Acquire image, convert to grayscale

Downscale image

Compute gradient images

Compute row sums

Transpose, compute column sums

Evaluate cascade stages

GPU

Non-maxima suppression, display results

CPU
Performance Gain

• Using GPU, we observe a speed up by a factor of 13 relative our CPU implementation:

<table>
<thead>
<tr>
<th>Detectors</th>
<th>Dense scan 1280×960 image</th>
<th>Sparse scan 1280×960 image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our GPU</td>
<td>422 ms</td>
<td>131 ms</td>
</tr>
<tr>
<td>Our CPU</td>
<td>5.5 sec</td>
<td>1.7 sec</td>
</tr>
</tbody>
</table>

• Real time processing of a 1.2 Megapixel image is possible at 8 FPS.
Algorithm: Sequential integral image formulation

\( I \): input image with size \( w \times h \)
\( I_{int} \): integral image with size \( w \times h \)
Array elements are accessed in row major order.

\[
\begin{align*}
\text{for } x = 0 \text{ to } w-1 \text{ do} \\
& I_{int}[x] \leftarrow 0 \\
\text{for } y = 1 \text{ to } h-1 \text{ do} \\
& I_{int}[y \cdot w] \leftarrow 0 \\
& s \leftarrow 0 \\
& \text{for } x = 0 \text{ to } w-1 \text{ do} \\
& \quad s \leftarrow s + I[x + (y-1) \cdot w] \\
& \quad I_{int}[x + y \cdot w + 1] \leftarrow s + I_{int}[x + (y-1) \cdot w + 1]
\end{align*}
\]
CUDA Code: Scan kernel for the GPU

__global__ void scan(float *input, float *output, int n)
{
    extern __shared__ float temp[];
    int tdx = threadIdx.x; int offset = 1;

    temp[2*tdx] = input[2*tdx];
    temp[2*tdx+1] = input[2*tdx+1];

    for(int d = n>>1; d > 0; d >>= 1)
    {
        __syncthreads();
        if(tdx < d)
        {
            int ai = offset*(2*tdx+1)-1;
            int bi = offset*(2*tdx+2)-1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
    }

    if(tdx == 0) temp[n - 1] = 0;

    for(int d = 1; d < n; d *= 2)
    {
        offset >>= 1; __syncthreads();
        if(tdx < d)
        {
            int ai = offset*(2*tdx+1)-1;
            int bi = offset*(2*tdx+2)-1;

            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }
}

__syncthreads();
output[2*tdx] = temp[2*tdx];
output[2*tdx+1] = temp[2*tdx+1];
Input array

Scan blocks

Store sums to aux. array $I_{sum}$

Scan $I_{sum}$

Add $I_{sum}[i]$ to all elements in $(i+1)^{st}$ block
CUDA Code: Transpose kernel for the GPU

```c
__global__ void transpose(float *input, float *output, int width, int height)
{
    __shared__ float temp[BLOCK_DIM][BLOCK_DIM+1];

    int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;

    if ((xIndex < width) && (yIndex < height))
    {
        int id_in = yIndex * width + xIndex;
        temp[threadIdx.y][threadIdx.x] = input[id_in];
    }

    __syncthreads();

    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;

    if ((xIndex < height) && (yIndex < width))
    {
        int id_out = yIndex * height + xIndex;
        output[id_out] = temp[threadIdx.x][threadIdx.y];
    }
}
```
Float4 and Float Integral Image Times

- **Our Float4**
- **Our Float**
- **4x Our Float**

Integral Image Time (ms)

Image Size (pixels)

$\times 10^6$