Compute unit in objects_detector_p->compute();

In function
void ObjectsDetectionApplication::one_step(const cv::Mat& m_detecting_image)
We assume the objects_detector_p has been init

GpuIntegralChannelsDetector.cpp

```
void GpuIntegralChannelsDetector::compute()
{
    detections.clear();
    num_gpu_detections = 0; // no need to clean the buffer

    // some debugging variables
    static bool first_call = true;

    assert(integral_channels_computer_p);
    //assert(gpu_detection_variant_cascade_per_scale.getBuffer() != NULL);

    // for each range search
    for(size_t search_range_index = 0; search_range_index < search_ranges_data.size(); search_range_index +=1)
    {
Page 6  compute_detections_at_specific_scale_v1(search_range_index, first_call);
    } // end of "for each search range"

    collect_the_gpu_detections();

    log.info() << "number of raw (before non maximal suppression) detections on this frame == "
               << detections.size() << std::endl;

    // windows size adjustment should be done before non-maximal suppression

    (*model_window_to_object_window_converter_p)(detections);
Page 10
    compute_non_maximal_suppresion();

    first_call = false;

    return;
}
```

Search range has been initialized, so what's the search range mean?
\src\objects_detection\DetectorSearchRange.hpp

IntegralChannelsDetector.cpp

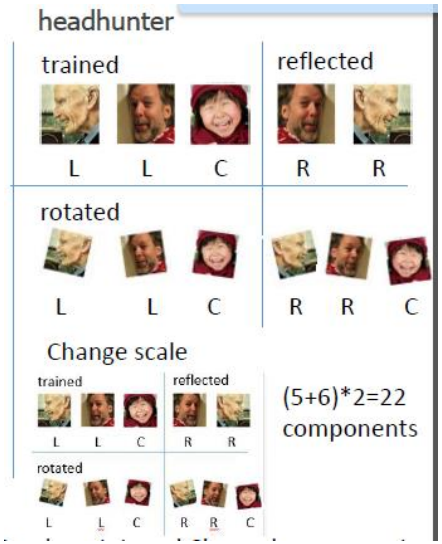
```
void IntegralChannelsDetector::compute()
{
    detections.clear();

    // some debugging variables
    static bool first_call = true;

    assert(integral_channels_computer_p);
    assert(search_ranges_data.size() == detection_cascade_per_scale.size());

    // for each range search
    for(size_t search_range_index=0; search_range_index < search_ranges_data.size(); search_range_index +=1)
    {
      compute_detections_at_specific_scale(search_range_index,first_call);

    } // end of "for each search range"

    process_raw_detections();

    first_call = false;

    return;
}

void IntegralChannelsDetector::process_raw_detections()
{

    const size_t num_raw_detections = detections.size();

    // windows size adjustment should be done before non-maximal suppression

    (*model_window_to_object_window_converter_p)(detections);

    log_info() << "number of detections (before non maximal suppression)  on this frame == "
               << num_raw_detections << " (raw) / " <<  detections.size() << " (after filtering)" << std::endl;

    compute_non_maximal_suppresion();

    return;
}
```

First, we focus on the key step, compute.
In compute both cpu/gpu version call a function like this.

```
compute_detections_at_specific_scale(search_range_index, first_call);
```

so what's the search range mean?

In next page, I crop the output of the example program using trained model headhunter with the config file eccv2014_face_detection_pascal.config.ini



22 unique Integral Channel components
With 2 scales， 11 components(11 semantic categories) each scale.

1 in 2

11

In each scale to be searched, choose a scales near it and generate a search range for each component

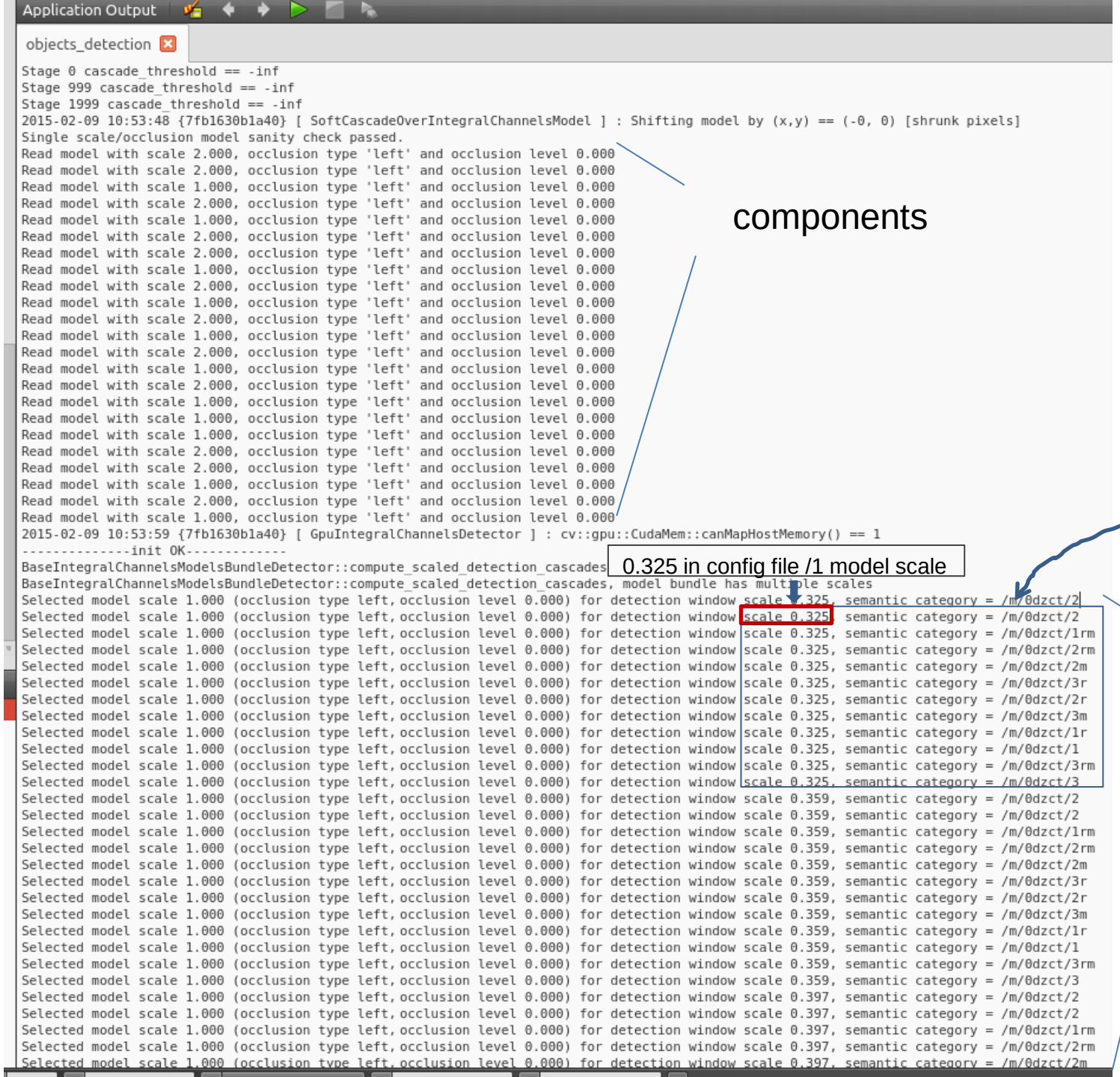
One search range bind to one certain detector component and one scale of the input image

30

```
x_stride = 0.001
y_stride = 0.001

non_maximal_suppression_method = greedy
minimal_overlap_threshold=0.3

min_scale = 0.325
max_scale = 6

num_scales = 30
```

The input image will be resize to 30 scales

- In this example we will have 30*11=330 search range.



components

0.325 in config file /1 model scale

11 components each scale

Search range

Consider one search range in last slide.

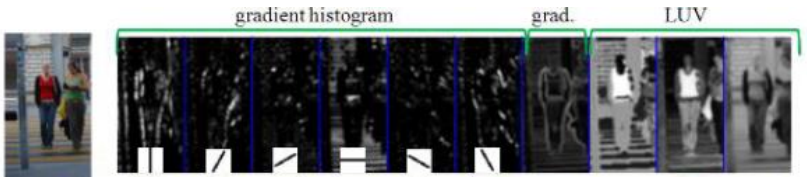
GpuIntegralChannelsDetector.cpp

```
void GpuIntegralChannelsDetector::compute_detections_at_specific_scale_v1(
        const size_t search_range_index,
        const bool first_call)
{
    doppia::objects_detection::gpu_integral_channels_t &integral_channels =
            resize_input_and_compute_integral_channels(search_range_index, first_call);

    const ScaleData &scale_data = extra_data_per_scale[search_range_index];

    // compute the scores --
    invoke_v1_integral_channels_detector visitor(
                integral_channels,
                search_range_index,
                scale_data,
                score_threshold,
                gpu_detections, num_gpu_detections);

    // compute the detections, and keep the results on the gpu memory
    boost::apply_visitor(visitor, gpu_detection_variant_cascade_per_scale);

    // ( the detections will be colected after iterating over all the scales )

#if defined(BOOTSTRAPPING_LIB)
    throw std::runtime_error("GpuIntegralChannelsDetector::compute_detections_at_specific_scale_v1 "
                             "should not be used inside bootstrapping_lib, use v0 instead");
#endif
    return;
}
```

- Resize the input image to the scale.
- Computer the integral feature of the image at that scale

(to speed up we can compute only once in each scale but not done here)

- Search faces in the search range above.



10 channels =
6 hog features
1 gradient
3 LUV (color)

This is the 10 channels
and integral channels are the integral of these channels.

\src\objects_detection\gpu\integral_channels_detector.cu

```
671 /// this method directly adds elements into the gpu_detections vector
672 template<typename GpuDetectionCascadePerScaleType>
673 void integral_channels_detector_impl(gpu_integral_channels_t &integral_channels,
674                                      const size_t search_range_index,
675                                      const doppia::ScaleData &scale_data,
676                                      GpuDetectionCascadePerScaleType &detection_cascade_per_scale,
677                                      const float score_threshold,
678                                      gpu_detections_t& gpu_detections,
679                                      size_t &num_detections)
680 {
                          ⋮
724     const int
725             width = search_range.max_x - search_range.min_x,
726             height = search_range.max_y - search_range.min_y;
727
728     if((width <= 0) or (height <= 0))
729     { // nothing to be done
730         // num_detections is left unchanged
731         return;
732     }
733
734     dim3 grid_dimensions(div_up(width, block_dimensions.x),
735                          div_up(height, block_dimensions.y));
736
737     // prepare variables for kernel call --
738     bind_integral_channels_texture(integral_channels);
739     move_num_detections_from_cpu_to_gpu(num_detections);
740
741     integral_channels_detector_kernel
742             <CascadeStageType>
743             <<<grid_dimensions, block_dimensions>>>
744                                              (scale_datum,
745                                               integral_channels,
746                                               search_range_index,
747                                               detection_cascade_per_scale,
748                                               score_threshold,
749                                               gpu_detections);
750
```

- Search faces in the search range
- This function compute the each possible windows parallel.
- The detector (the cascade) Compute the score of the windows and deicide if it is a face with the score_threshold defined in config file.

```
20  score_threshold = 0.05 # nice for visualization
21  #score_threshold = 0 # default threshold
```

\src\objects_detection\gpu\integral_channels_detector.cu

```
/// this kernel is called for each position where we which to detect objects
/// we assume that the border effects where already checked when computing the DetectorSearchRange
/// thus we do not do any checks here.
/// This kernel is a mirror of the CPU method compute_cascade_stage_on_row(...) inside IntegralChannelsDetector.cpp
/// @see IntegralChannelsDetector
template <typename DetectionCascadeStageType>
__global__
void integral_channels_detector_kernel(
        const gpu_scale_datum_t scale_datum,
        const gpu_integral_channels_t::KernelConstData integral_channels,
        const size_t scale_index,
        const typename Cuda::DeviceMemory<DetectionCascadeStageType, 2>::KernelConstData detection_cascade_per_scale,
        const float score_threshold,
        gpu_detections_t::KernelData gpu_detections)
{
                                :
                                :

    // retrieve current score value
    float detection_score = 0;

    const size_t
            cascade_length = detection_cascade_per_scale.size[0],
            scale_offset = scale_index * detection_cascade_per_scale.stride[0];

    for(size_t stage_index = 0; stage_index < cascade_length; stage_index += 1)
    {
        const size_t index = scale_offset + stage_index;

        // we copy the cascade stage from global memory to thread memory
        // (when using a reference code runs at ~4.35 Hz, with copy it runs at ~4.55 Hz)
        const DetectionCascadeStageType stage = detection_cascade_per_scale.data[index];

        update_detection_score(x, y, stage, integral_channels, detection_score);

        if(detection_score < stage.cascade_threshold)
        {
            // this is not an object of the class we are looking for
            // do an early stop of this pixel
            detection_score = -1E5; // since re-ordered classifiers may have a "very high threshold
            break;
        }

    } // end of "for each stage"

    // >= to be consistent with Markus's code
    if(detection_score >= score_threshold)
    {
        // we got a detection
        add_detection(gpu_detections, x, y, scale_index, detection_score);
    }

    return;
}
```

- The final score is the sum of all stages in a cascade

```
stages {
  feature_type: Level2DecisionTree
  weight: 0.829010546207
  cascade_threshold: -inf
  nodes {
    id: 0
    parent_id: 0
    parent_value: true
    decision_stump {
      feature {
        channel_index: 3
        box {
          min_corner {
            x: 18
            y: 25
          }
          max_corner {
            x: 22
            y: 29
          }
        }
      }
      feature_threshold: 49.0
      larger_than_threshold: false
    }
  }
```

Never break, sum up all the stages in this case

```
  nodes {
    id: 1
    parent_id: 0
    parent_value: true
    decision_stump {
      feature {
        channel_index: 7
        box {
          min_corner {
            x: 0
            y: 29
          }
          max_corner {
            x: 1
            y: 30
          }
        }
      }
      feature_threshold: 1.0
      larger_than_threshold: false
    }
  }
```

```
  nodes {
    id: 2
    parent_id: 0
    parent_value: false
    decision_stump {
      feature {
        channel_index: 3
        box {
          min_corner {
            x: 5
            y: 22
          }
          max_corner {
            x: 13
            y: 30
          }
        }
      }
      feature_threshold: 91.0
      larger_than_threshold: false
    }
  }
```

\src\objects_detection\gpu\integral_channels_detector.cu

In root stump, score compared with threshold return true or false.

In leaf stump, Score compared with threshold return +weight or -weight

```
256     const float level1_feature_value =
257             get_feature_value<feature_t, use_2d_texture>(
258                 weak_classifier.level1_node.feature, x, y, integral_channels);
284         if(evaluate_decision_stump(weak_classifier.level1_node, level1_feature_value))
285         {
286             const float level2_true_feature_value =
287                     get_feature_value<feature_t, use_2d_texture>(
288                         weak_classifier.level2_true_node.feature, x, y, integral_channels);
289
290             current_score += evaluate_decision_stump(weak_classifier.level2_true_node, level2_true_feature_value);
291         }
292         else
293         {
294             const float level2_false_feature_value =
295                     get_feature_value<feature_t, use_2d_texture>(
296                         weak_classifier.level2_false_node.feature, x, y, integral_channels);
297
298             current_score += evaluate_decision_stump(weak_classifier.level2_false_node, level2_false_feature_value);
299         }
```

```
stages {
  feature_type: Level2DecisionTree
  weight: 0.829010546207
  cascade_threshold: -inf
```

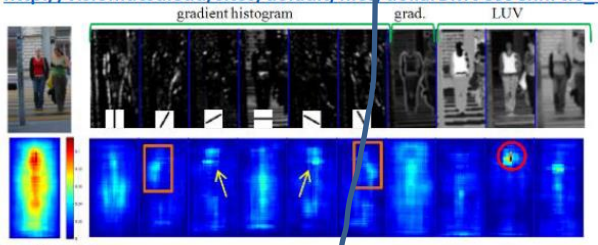
```
nodes {
  id: 0
  parent_id: 0
  parent_value: true
  decision_stump {
    feature {
      channel_index: 3
      box {
        min_corner {
          x: 18
          y: 25
        }
        max_corner {
          x: 22
          y: 29
        }
      }
    }
    feature_threshold: 49.0
    larger_than_threshold: false
  }
}
```

```
nodes {
  id: 1
  parent_id: 0
  parent_value: true
  decision_stump {
    feature {
      channel_index: 7
      box {
        min_corner {
          x: 0
          y: 29
        }
        max_corner {
          x: 1
          y: 30
        }
      }
    }
    feature_threshold: 1.0
    larger_than_threshold: false
  }
}
```

```
nodes {
  id: 2
  parent_id: 0
  parent_value: false
  decision_stump {
    feature {
      channel_index: 3
      box {
        min_corner {
          x: 5
          y: 22
        }
        max_corner {
          x: 13
          y: 30
        }
      }
    }
    feature_threshold: 91.0
    larger_than_threshold: false
  }
}
```

feature

Dollár, Piotr, et al. "Integral Channel Features." *BMVC*. Vol. 2. No. 3. 2009.
http://vision.ucsd.edu/sites/default/files/dollarBMVC09ChnFtrs_0.pdf

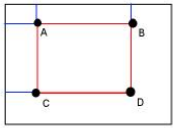


10 channels =
6 hog features
1 gradient
3 LUV (color)

```
feature {
  channel_index: 4
  box {
    min_corner {
      x: 17
      y: 26
    }
    max_corner {
      x: 21
      y: 30
    }
  }
}
```

Score = sum of the value in channel n within the box.

1. For one input picture, calculate integral image on each channels. Which can be reused when calculate the feature.
2. Calculate score in one feature box.
3. Compare to the threshold, output true of false.



Sum = D - B - C + A
Summed area table
integral image

\objects_detection\non_maximal_suppression\GreedyNonMaximalSuppression.cpp

```
275 void GreedyNonMaximalSuppression::compute()
276 {
277     candidate_detections.sort(has_higher_score);
278     maximal_detections.clear();
279     maximal_detections.reserve(64); // we do not expect more than 64 pedestrians per scene
280
281     candidate_detections_t::iterator detections_it = candidate_detections.begin();
282     for(; detections_it != candidate_detections.end(); ++detections_it)
283     {
284         const detection_t &detection = *detections_it;
285
286         // this detection passed the test
287         maximal_detections.push_back(detection);
288
289         candidate_detections_t::iterator lower_score_detection_it = detections_it;
290         ++lower_score_detection_it; // = detections_it + 1
291         for(; lower_score_detection_it != candidate_detections.end(); )
292         {
293             const float overlap = compute_overlap_inlined(detection, *lower_score_detection_it, overlap_method);
294
295             if(overlap > minimal_overlap_threshold)
296             {
297                 // this detection seems to overlap too much, we should remove it
298                 lower_score_detection_it = candidate_detections.erase(lower_score_detection_it);
299             }
300             else
301             {
302                 // we keep this detection in the candidates list
303                 ++lower_score_detection_it;
304             }
305
306         } // end of "for each lower score detection"
307
308
309     } // end of "for each candidate detection"
310
311     return;
312 }
```

- Sort the candidate by score
- remove the overlap candidate with small score

Summary for the detection

using non-maximal suppression

A candidate face
A candidate face
A candidate face
A candidate face

Result faces

Detector with soft_cascade_model

```
stages {
  feature_type: Level2DecisionTree
  weight: 0.839436531067
  cascade threshold: -inf
```

```
stages {
  feature_type: Level2DecisionTree
  weight: 0.555842518806
  cascade threshold: -inf
```

```
stages {
  feature_type: Level2DecisionTree
  weight: 0.506709218025
  cascade_threshold: -inf
```

⋮

```
stages {
  feature_type: Level2DecisionTree
  weight: 0.538094162941
  cascade threshold: -inf
```

Sum(stages)>score_threshold?

A candidate face

window
Image size / stride

search range
N components * image scale