Maximizing Face Detection Performance

Paulius Micikevicius

Developer Technology Engineer, NVIDIA

GTC 2015

- **Outline**
  - Very brief review of cascaded-classifiers
  - Parallelization choices
  - Reducing the amount of work
  - Improving cache behavior
  - Note on feature format
- **The points made apply to any cascaded classifier**
  - Face detection is just one example

Quick Review

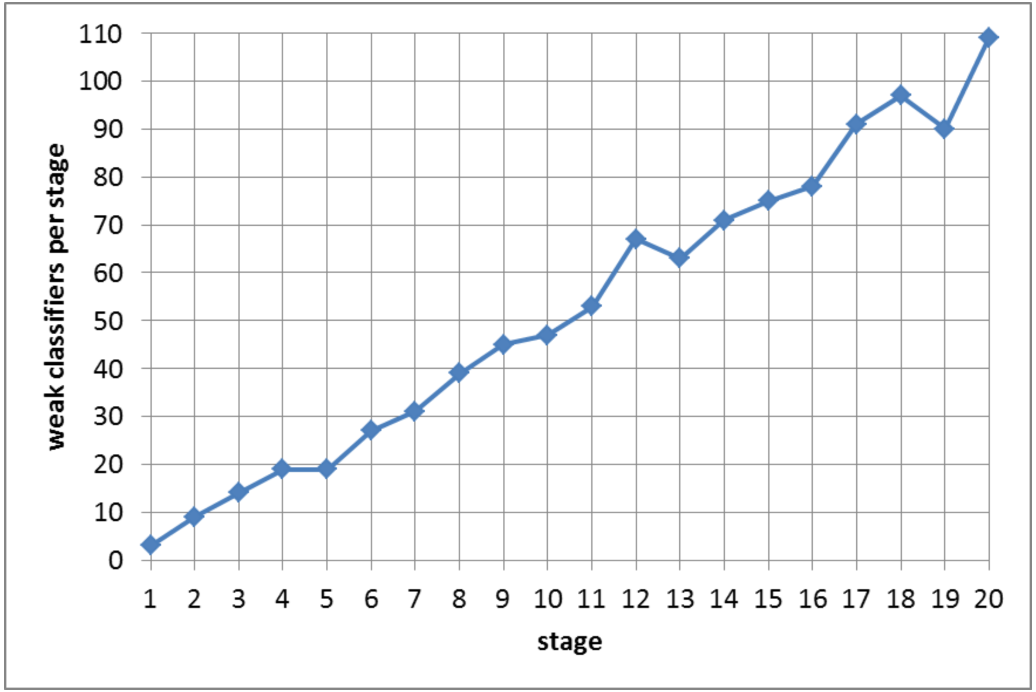
- **“Slide” a window around the image**
  - Use weak classifiers to detect object presence in each position
  - I’ll call a position a ***candidate***
    - Think of all the (x,y) positions that could be upper-left corners of a candidate window
    - Each candidate is independent of all others -> easy opportunity to parallelize
- **Cascade of weak-classifiers per candidate**
  - Some number of stages are cascaded
    - Decision to continue/abort is made after each stage
  - Each stage contains a number of weak-classifiers
    - Evaluate some feature on the window, add its result to the running-stage sum
- **Do this at multiple scales**
  - Classifiers are trained on small windows (~20x20 pixels)
  - To detect objects of different sizes, do one of:
    - Adjust the size of candidate windows (and scale features)
    - Adjust (scale) image to match training window-size
- **“Group” the candidates that passed the entire cascade**

Input Image

Candidates that Pass All Stages

Candidates After Grouping

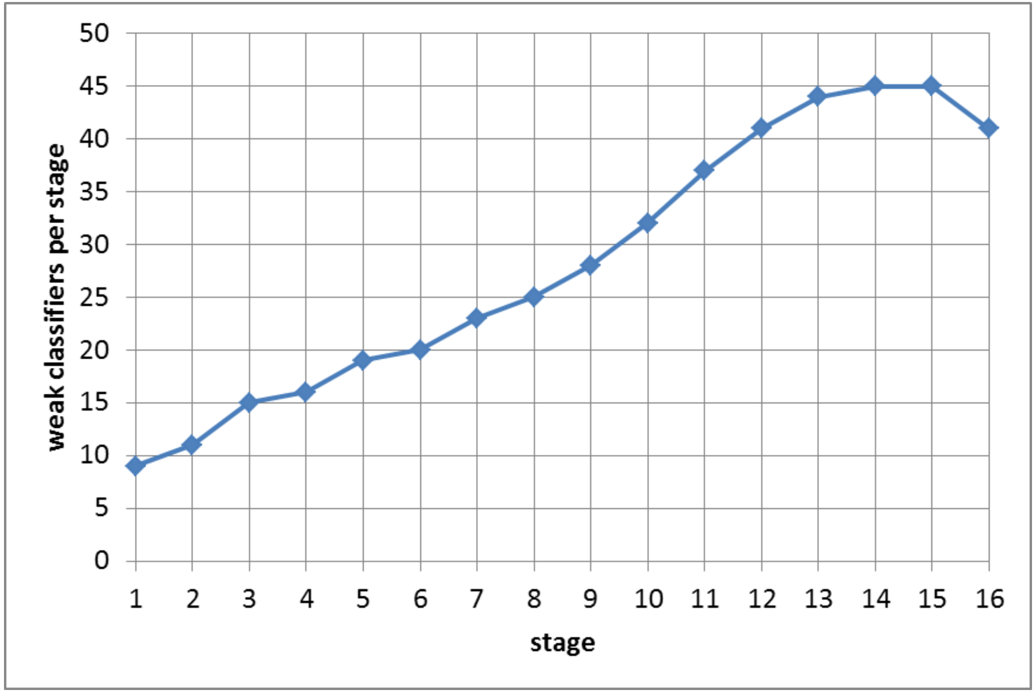
OpenCV haarcascade_frontalface_alt2.xml



- **20 stages**
- **1047 weak-classifiers**
  - 2094 Haar-like features
  - Each weak classifier is a 2-feature tree
- **4535 rectangles**
  - 1747 features contain 2 rects
  - 347 features have 3 rects

- **Idea is to reject more and more negatives with successive stages, passing through the positives**
- **Earlier stages are simpler for perf reasons**
  - Quickly reject negatives, reducing work for subsequent stages
  - False-positives are OK, false-negatives are not OK

MBLBP Classifier



- **16 stages**
- **451 features**
  - 4059 rects
  - 419 unique features

Parallelization

- **Ample opportunity for parallelization**
  - Scales are independent of each other
  - Each scale has a (large) number of candidates, all independent
- **A number of choices to be made:**
  - Number of threads per candidate window
    - One or multiple threads per candidate
  - Cascade stage processing
    - All stages in a single or multiple kernel launches
  - Scale processing
    - In sequence (single stream) or concurrent (multiple streams)

Parallelization

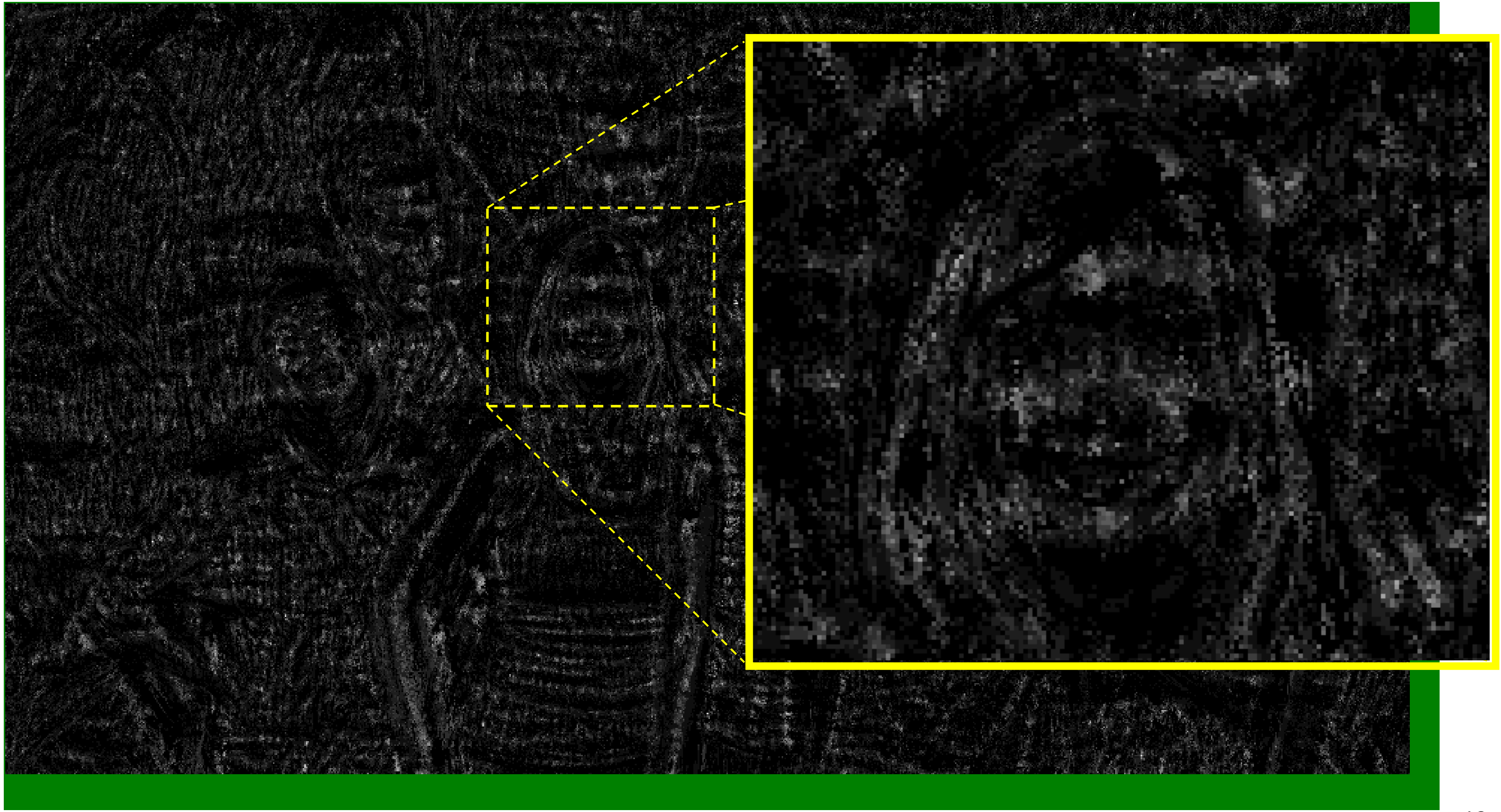
- **Ample opportunity for parallelization**
  - Scales are independent of each other
  - Each scale has a (large) number of candidates, all independent
- **A number of choices to be made:**
  - Numbe
    - One
  - Cascade
    - All st
  - Scale processing
    - In sequence (single stream) or concurrent (multiple streams)

The combination of choices can be overwhelming, so it helps to get some intuition for the algorithm operation

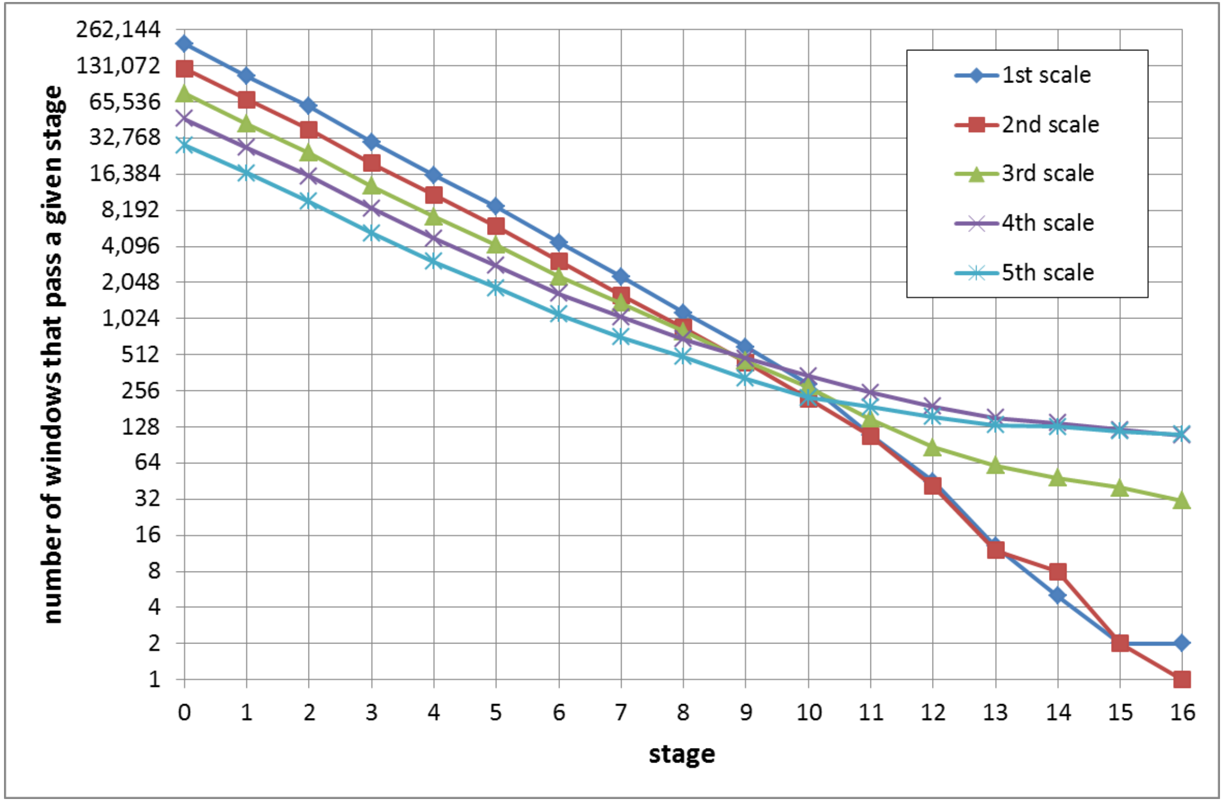
Input Image

Lighter = Candidate Passed More Stages

Lighter = Candidate Passed More Stages



Candidates Passing Stages



1920x1080 input image

5 scales:

- 50-200 pixel faces
- 1.25x scaling factor

Process each candidate

- Start with 478K candidates
- 254 pass all stages

Observations

- **Adjacent candidates can pass very different number of stages**
  - Different amount of work for adjacent candidates
- **The amount of candidates remaining decreases with the number of stages**
  - Often each stage rejects ~50% of candidates
    - Depends on training parameters, etc.

Parallelization Choices

Chosen Parallelization

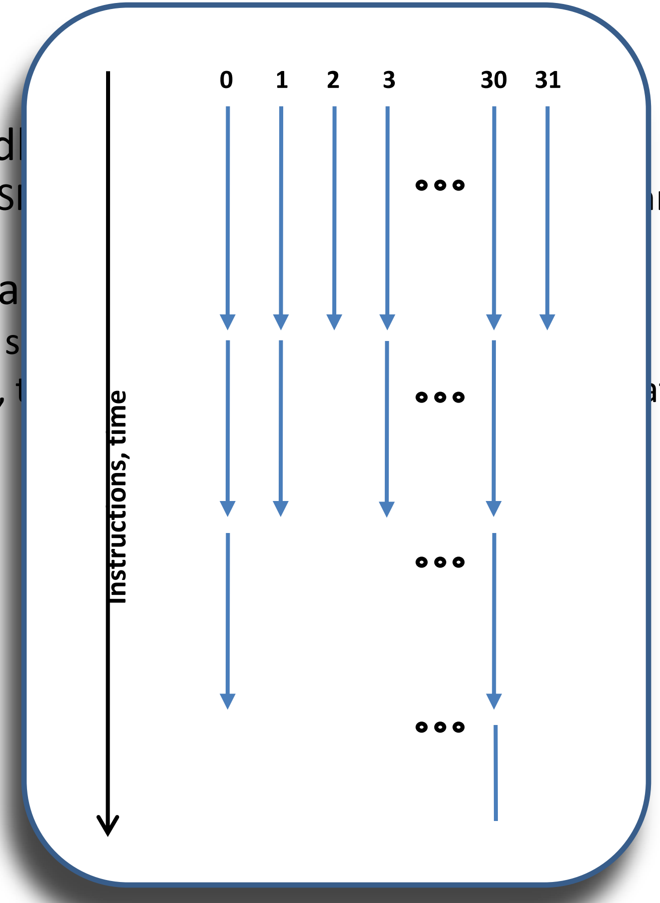
- **One thread per candidate**
  - A thread iterates through the stages, deciding whether to continue after each stage
    - Loop through the weak-classifiers for each stage
  - Simple port: kernel code nearly identical to CPU code
    - CPU-only code iterates through the candidates ("slides the window")
    - GPU code launches a thread for each candidate
      - GPU kernel code = CPU loop body
- **Two challenges:**
  - Different workloads per candidate (thus per thread)
  - Having enough work to saturate a GPU

Challenge: Different Workloads

- **GPU execution refresher:**
  - Threads are grouped into threadblocks
    - Resources (thread IDs, registers, SMEM) are released only when all the threads in a block terminate
  - Instructions are executed per warp (SIMT)
    - 32 consecutive threads issue the same instruction
    - Different code paths are allowed, threads get “masked out” during the path they don’t take

Challenge: Different Workloads

- **GPU execution refresher:**
  - Threads are grouped into threadl
    - Resources (thread IDs, registers, SI   reads in a block terminate
  - Instructions are executed per wa
    - 32 consecutive threads issue the s
    - Different code paths are allowed, t   ath they don't take



Challenge: Different Workloads

- **GPU execution refresher:**
  - Threads are grouped into threadblocks
    - Resources (thread IDs, registers, SMEM) are released only when all the threads in a block terminate
  - Instructions are executed per warp (SIMT)
    - 32 consecutive threads issue the same instruction
    - Different code paths are allowed, threads get “masked out” during the path they don’t take
- **What these mean to cascades:**
  - If at least one thread in a warp needs to evaluate a stage, all 32 threads go through evaluation instructions
    - Inactive threads waste math pipelines
  - If at least one thread in a threadblock needs to continue evaluating, the resources of all other threads in that block are not released
    - Prevent new threads from starting right away

Stage Processing

- **Threads decide whether to terminate after each stage**
- **Could process all stages with a single kernel launch**
  - Potentially wasting the math and resources
- **Could break stages into segments (work “compaction”)**
  - A sequence of kernel launches, one per segment
  - Maintain a work-queue
    - Launch only as many threads as there are candidates in the queue
    - At the end of each segment append the live candidates to the queue
      - Use atomics for updating the index
  - Work-queue maintenance adds some overhead
    - Read/write queues (writes are atomic)
    - Communicate queue size to CPU for subsequent launch

Stage Processing: Timing Results

- **20-stage classifier, TK1**
  - 1 segment: 127 ms (1-20 stages)
  - 2 segments: 93 ms (1-3, 4-20 stages)
  - 3 segments: 84 ms (1-3, 4-7, 8-20 stages)
- **16-stage classifier:**
  - 1 segment: 134 ms
  - 2 segments: 126 ms (1-2, 3-16 stages)
    - K40: 9.8 ms, 8.7 ms

Why I Didn't Choose SMEM Here

- **SMEM could be used to store the integral image tile needed by a threadblock, but:**
  - SMEM makes scaling features impractical
    - SMEM overhead becomes prohibitive, forcing us to scale images
  - SMEM precludes work-compaction:
    - A threadblock must cover a contiguous region to read all the inputs
- **Preliminary test with another classifier showed very small difference between using SMEM or just reading via texture cache**
  - And the texture code was still scaling image (could have been avoided)
  - Can use either texture functions, or __ldg() with "regular" pointers
- **Caution: the evidence isn't conclusive yet**
  - Classifiers that benefit little from compaction may benefit from SMEM

Why I Didn't Choose SMEM Here

- **SMEM coul**
  threadbloc
  - SMEM n
    - SMEI
  - SMEM p
    - A thr
- **Preliminar**
  difference

- Say the training window is 20x20 (can be bigger)
- Given a 32x32 threadblock we'd need:
  - Scale 1 (20x20 faces): (32+20)x(32+20) SMEM
  - Scale 5 (100x100 faces): (32+100)x(32+100) SMEM
    - Exceeds SMEM available for a single threadblock
    - Halo is ~16x bigger than the output tile
- So, we'd have to resize input image for each scale
  - Additional passes to memory, to both scale and compute integral images

  - And the texture code was still scaling image (could have been avoided)
  - Can use either texture functions, or __ldg() with "regular" pointers
- **Caution: the evidence isn't conclusive yet**
  - Classifiers that benefit little from compaction may benefit from SMEM

Why I Didn’t Choose SMEM Here

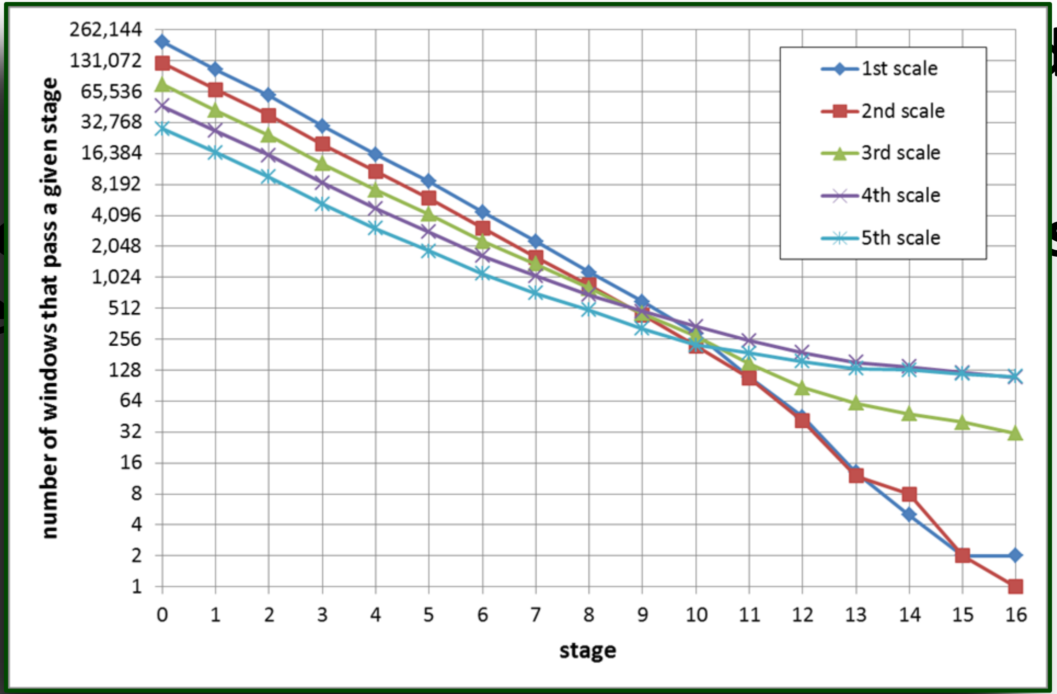
- **SMEM could be used to store the integral image tile needed by a threadblock, but:**
  - SMEM makes scaling features impractical
    - SMEM overhead becomes prohibitive, forcing us to scale images
  - SMEM precludes work-compaction:
    - A threadblock must cover a contiguous region to read all the inputs
- **Preliminary test with another classifier showed very small difference between using SMEM or just reading via texture cache**
  - And the texture code was still scaling image (could have been avoided)
  - Can use either texture functions, or __ldg() with “regular” pointers
- **Caution: the evidence isn’t conclusive yet**
  - Classifiers that benefit little from compaction may benefit from SMEM

Challenge: Enough Work to Saturate a GPU

- **We start out with 100s of thousands of candidates**
  - Plenty to saturate even the biggest GPUs
- **We are left with fewer and fewer candidates as stages reject them**
  - Even 1-SM GPUs (TK1) will start idling
  - Bigger GPUs will start idling sooner

Challenge: Enough Work to Saturate a GPU

- We start … didates
  - Plenty
- We are l … s as stages re
  - Even 1
  - Bigger



Challenge: Enough Work to Saturate a GPU

- **We start out with 100s of thousands of candidates**
  - Plenty to saturate even the biggest GPUs
- **We are left with fewer and fewer candidates as stages reject them**
  - Even 1-SM GPUs (TK1) will start idling
  - Bigger GPUs will start idling sooner
- **Two solutions:**
  - Process scales concurrently
  - Switch parallelization after some number of stages

Concurrent Scale Processing

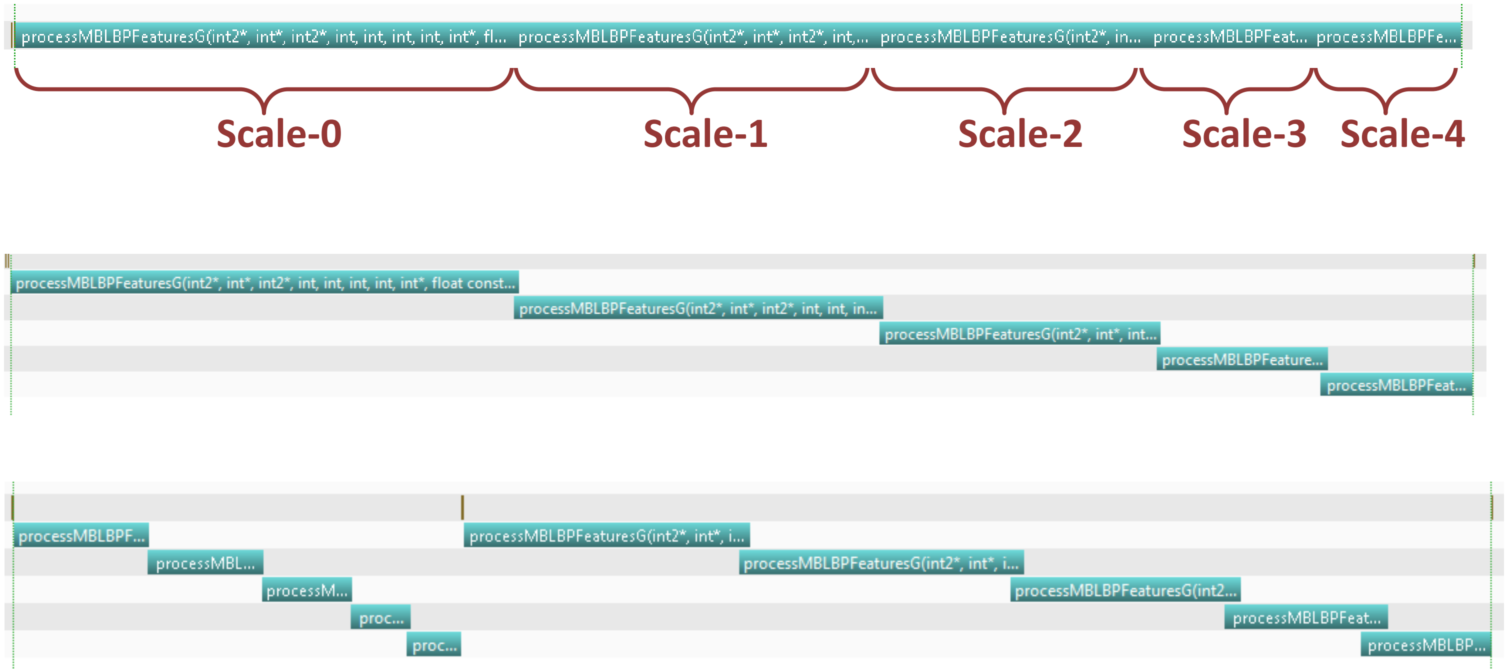
- **Issue kernels for different scales into different streams**
  - Scales are independent
  - Maintain a different work-queue for each scale
    - So that features can be properly scaled
- **Orthogonal to work-compaction:**
  - Loop through the segments
  - For each segment launch as many kernels as you have scales
- **GPU stream support in hw:**
  - TK1 supports 4 streams
  - Other GPUs (Kepler and more recent) support 32 streams
  - More streams can be used in sw, but will result in stream aliasing

TK1: 16-stage MBLBP Classifier

Sequential

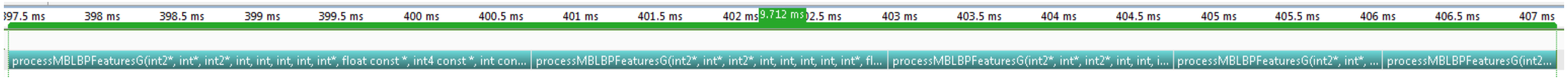
Concurrent

Concurrent 2 segments

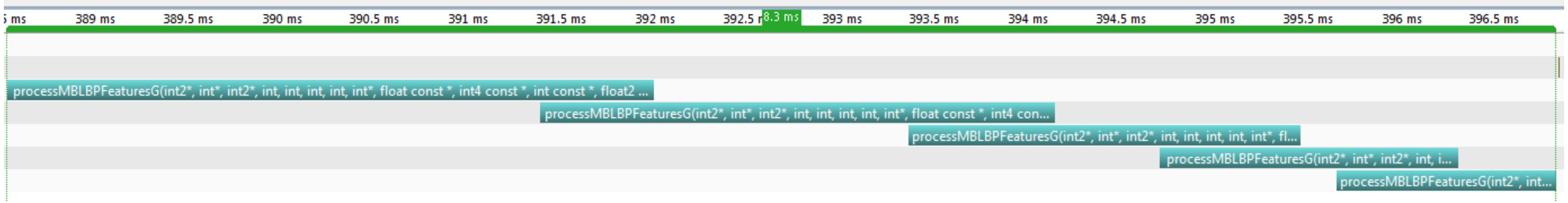


K40: 16-stage MBLBP Classifier

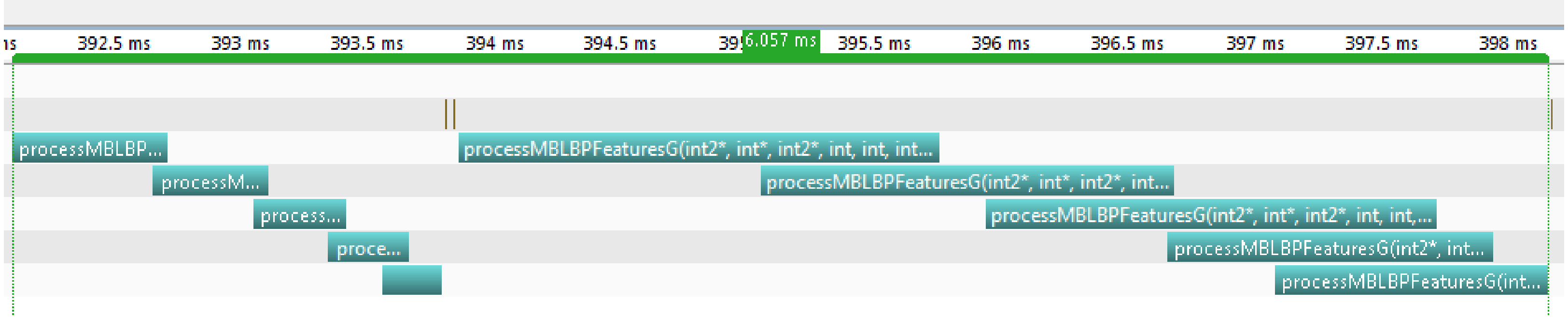
9.7 ms



8.3 ms

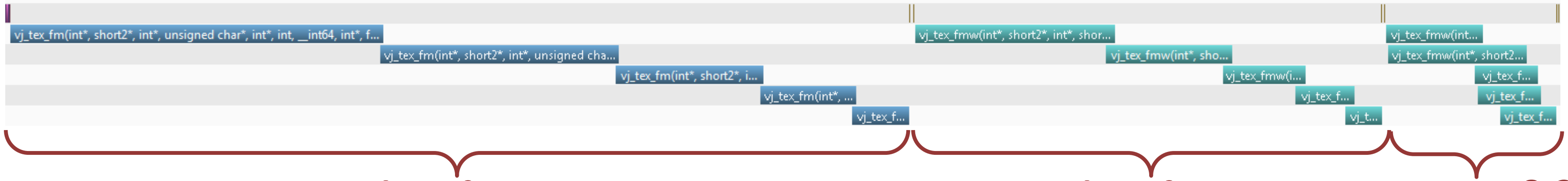


2-segments
6.1 ms



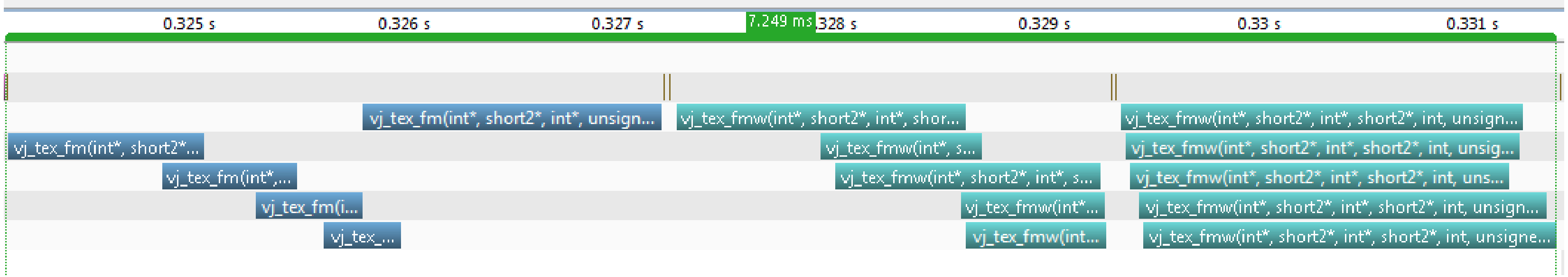
20-stage Haar-like

TK1: 78.9 ms



5 scales of stages 0-3 | 5 scales of stages 4-7 | stages 8-20

K40: 7.2 ms



Switching Parallelizations

- **One thread per candidate:**
  - Pro: candidates go through minimal stage count
  - Con: GPU becomes latency limited
    - After a number of stages there isn't enough work to hide latency
      - Very rough rule of thumb: fewer than 512 threads per SM
    - GPU becomes underutilized
- **Alternative parallelization:**
  - Use multiple threads per candidate, say a warp
    - A warp evaluates 32 features in parallel
    - Performs a reduction (or prefix sum) to compute stage sum
    - Power-of-2 up to a warp is nice because of the shfl/vote instructions
  - May do unnecessary work
    - A thread evaluates a feature it wouldn't have reached sequentially

Switching Parallelizations

- **Idea: change parallelization when only a few 100 candidates remain**
  - Prior to that continue to use 1 thread/candidate
    - Avoids inter-thread communication and unnecessary work
- **Preliminary work on a different classifier:**
  - A few 100 features
  - Speedup:
    - K40: 1.6-1.75x (depending on image)
    - TK1: 1.0x
  - Results suggest that:
    - Alternative parallelization helps when you have lots of stages with too few candidates to saturate the GPU
    - Confirmed when TK1 ran a classifier with even more stages

Work Reduction

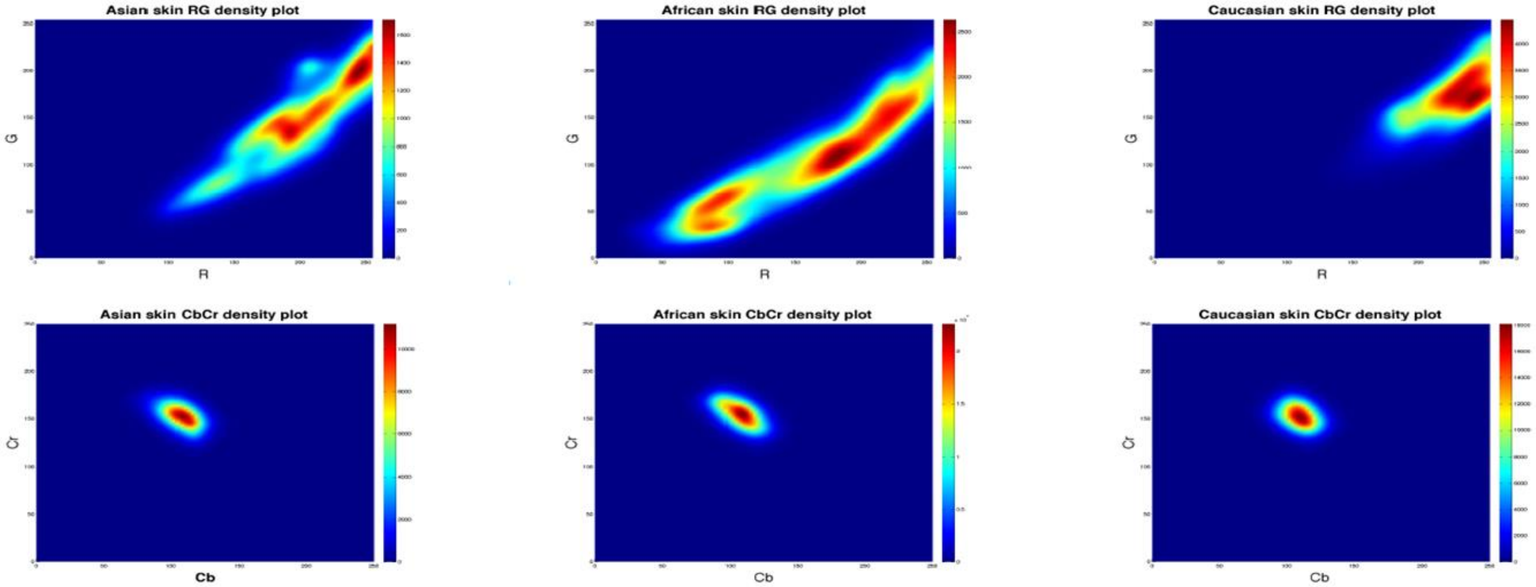
Reduce the Initial Number of Candidates

- **Less work -> less time**
  - Will reach the point of non-saturated GPU sooner
  - Makes concurrent scale processing even more useful
- **Two ways to reduce the initial candidate count:**
  - Use a mask to not consider some candidates
    - ROI, skin-tone, etc.
  - “Skip” candidates (stride > 1)
    - Post-process neighborhoods of rectangles that didn’t get grouped

Skin Tone Mask

- **Race invariant, simply needs a white-balanced camera**
- **Color density plots for asian, african, and caucasian skin**

from http://www.cs.rutgers.edu/~elgammal/pub/skin.pdf:



Candidate Mask

- **Mask pixel at (x,y) corresponds to upper-left corner of a candidate window**
  - Shown for scale-0 (58-pixel face)
  - A candidate window is masked out (black) if fewer than 50% of its pixels were not skin-toned
  - 76% of candidates were rejected at this scale

More Candidate Masks

Skin-tone Masking

- **A bit of extra work:**
  - Classify each input pixel as skin-toned or not
    - 5-10 math instructions in RGB or YUV
    - Can be done in the same kernel as RGB->luminance conversion
  - Compute integral image of pixel classes
  - Use the integral image to reject candidates when creating the initial work-queue for detection
- **Experimental data:**
  - TK1:
    - No mask, no streams, no segments: 134.5 ms
    - Mask, no streams, no segments: 34.9 ms (~4x speedup, as expected)
    - Mask, streams, no segments: 34.4 ms
    - Mask, streams, segments: 34.0 ms
  - K40:
    - No mask, no streams, no segments: 9.8 ms
    - Mask, no streams, no segments: 4.3 ms (~2.3x speedup -> less than 4x expected indicates GPU is idling)
    - Mask, streams, no segments: 2.8 ms
    - Mask, streams, segments: 2.1 ms

Improving Cache Behavior

Improving Cache Behavior

- **Till now the integral image was 1921x1081**
- **First scale (scale-0) is 2.44x:**
  - Training window is 24 pixels
  - Smallest face of interest is 50 pixels, scaling factor is 1.25x
  - Implies that a 787x443 integral image is sufficient
    - ~6x smaller than original size
- **Smaller image footprint can improve cache behavior**
  - In this case it's the read-only (aka "read-only") cache on the SM
  - Reduces requests to L2
    - Lower latency
    - Less bandwidth-pressure
    - higher L2 hit-rate -> less traffic to DRAM

Empirical Data

- **16-stage MBLBP classifier**
  - 2 segments, concurrent scale processing
- **TK1:**
  - Mask: 2.12x speedup (34 ms -> 16 ms)
  - No mask: 2.33x speedup (126 ms -> 54 ms)
- **K40:**
  - Mask: 1.27x speedup (2.1 ms -> 1.7 ms)
  - No mask: 1.56x speedup (7.5 ms -> 4.8 ms)

Benefits of Downscaling

- **Reduced requests to L2 by ~3x on both GPUs**
  - TK1 was being limited by L2 bandwidth:
    - Before downscaling: 40-93% of L2 theory
    - After downscaling: 28-74%
  - K40 was sensitive to L2 bandwidth:
    - Before downscaling: 12-70% of theory
    - After downscaling: 5-35%
    - K40 has 1.6x more L2 bandwidth/SM than TK1
      - Thus less sensitive to bandwidth for this application than TK1

Benefits of Downscaling

- **Reduced requests to L2 by ~3x on both GPUs**
  - TK1 was being limited by L2 bandwidth:
    - Before downscaling: 40-93% of L2 theory
    - After downscaling: 28-74%
  - K40 was sensitive to L2 bandwidth:
    - Before downscaling: 12-70% of theory
    - After downscaling: 5-35%
    - K40 has 1.6x more L2 bandwidth/SM than TK1
      - Thus less sensitive to bandwidth for this application than TK1
- **Improved L2 hit-rate (lowered traffic to DRAM)**
  - TK1: from 5-55% to 54-98%
  - K40: from 44-99% to 98-99%
    - K40 has 12x more L2 than TK1
      - Thus able to achieve a higher hit-rate than TK1, reducing traffic to DRAM

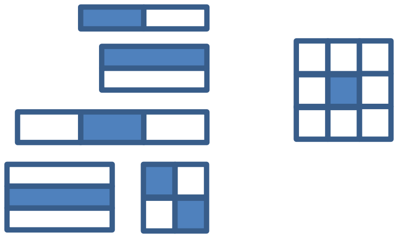
Quick Summary

- **We've examined several ways to improve performance**
  - Breaking stages into segments: up to **1.3x**
  - Concurrent processing of scales: **1.2 - 2x**
    - Can be higher, depending on classifier and GPU
  - Downscaling to the first scale first: **1.3 - 2.3x**
  - Masking (ROI): **~3x**
    - Depends on content and masking approach
- **All of the above use the same exact kernel code**
  - Adjust only image or launch parameters
  - Together improved cascade time:
    - TK1: from 134 ms to 16 ms (**8.4x** speedup)
    - K40: from 9.8 ms to 1.7 ms (**5.8x** speedup)
- **Switching parallelization after a number of stages**
  - Potential further speedup of **~1.5x**

Note on Feature Format

Feature Storage Format

- **Many features are rectangle based**
- **Two approaches to storing features in memory:**
  - Geometry:
    - coordinates/sizes within a window
  - Pointers:
    - Popular in OpenCV and other codes
    - Compute pointers to the vertices of window **0**
      - **Window-0**: the first window (top-left corner, for example)
    - Vertices for window ***k*** are addressed by adding offset ***k*** to these pointers
      - Pointer math per vertex: 64-bit multiply-add
        - » A dependent sequence of 2+ instructions on GPU

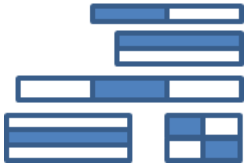


MB-LBP Features

- **Only one pattern: 3x3 tile of rectangles**
- **Pointers:**
  - Need 16 pointers: **128 B per feature**
  - 32 or more address instructions per window
- **Geometry:**
  - 4 values: (x,y) of top-left corner, width, height
  - **16 bytes per feature** when storing ints
    - could be as low as 4B when storing chars, but would require bit-extraction instructions
  - Address math: ~50 instructions

Haar-like Features

- **5 fundamental patterns (2 or 3 rectangles)**
- **Pointers:**
  - 6, 8, or 9 pointers: **48-72 bytes per feature**
  - 12-18 or more instructions per window
- **Geometry:**
  - Several choices:
    - Store each rectangle (2 or 3 per feature)
    - Store vertices (would need 5 categories)
  - When storing each rectangle
    - 4 values: (x,y) of top-left corner, width, height
    - All 4 values are relative to training window
      - Usually 20x20 to 32x32 in size
      - So, could store as few 4B (4 chars), 16 B if storing ints
        - » 4 chars would require bit-extraction instructions
    - 3x16B = **48 B per feature**
    - ~3x16 = 48 instructions per window



Pointers vs Geometry

- **When processing multiple scales:**
  - Geometry places no requirements when processing multiple scales
  - Pointers require one of:
    - Compute pointers for each scale
    - Scale image and compute integral for each scale
- **Pointers also require one of:**
  - Additional buffer for the integral image
    - Buffer to be reused by all images
  - Compute pointers for each input image

MBLBP Performance

- **Geometry was 3.5x faster than pointers**
  - Quick test: no segments, no streams, no mask
- **All other numbers in this presentation were measured with “geometry”**

Feature Multiples

- **Sometimes the same feature is used in several stages**
- **Two choices:**
  - Have multiple copies of the feature in memory
    - Simple array traversal
    - Consumes more memory
  - Add a level of indirection:
    - Each feature is stored exactly once
    - Maintain an array of indices
      - Map weak-classifiers to unique features
      - Approach implemented in OpenCV
- **Preference for performance: store multiple copies, avoid indirection**
  - Indirection adds 100s-1000s cycles of latency, adds to bandwidth pressure as well
    - Read the index from memory
    - Use the index to read feature from memory
  - Typically only a very small percentage of features are replicated
    - Negligible impact on memory consumed

Summary

- **Cascade performance for a 16-stage MBLBP classifier:**
  - TK1: 16.0 ms
  - K40: 1.6 ms
  - Can likely be improved further (these are without switched-parallelization)
- **We looked at:**
  - How to parallelize cascaded classifiers
  - How to reduce input to a cascade
  - How to maximize cache performance for cascades
  - How to store features in memory
  - Performance impact of the above:
    - Varies with classifier, detection parameters and GPU
    - Good choices can lead to O(10) speedup over the naïve approach