EECS 468
Project Review
Study of Loop Perforation on GPUs

Alex Broad

Akhil Guliani

Aims and Objectives

- We propose to find the best method to apply loop perforation on a GPU like device with minimal loss in computational accuracy with reduction in both time taken to compute the results and reduction of power consumption.

What is loop perforation ?

- Loop perforation transforms loops to execute a subset of their iterations.
- The goal is to reduce the amount of computational work
  - amount of time
  - resources such as power

Ways of doing loop Perforation

- Criticality Testing :
  - Finding which methods produce unacceptable results
  - Throwing out a warp == not good
- Perforation Space Exploration
  - Finding all the various possible results
  - Throwing out threads selectively
    - Increment only one dimensions
    - Increment both dimensions
    - Skip certain loops based on divisibility by factor

Benefits of Approximate computing

- Approximate computing is a new research direction that improves efficiency by carefully relaxing correctness constraints
- Quicker results
- Lower utilization of Resources

Our Case

We Study effects of loop perforation on

- Filter like algorithm such as Image Blurring
- Feature classification algo’s like : Local Binary Patterns
- Feature vector matching example Histogram Comparison

And compare the results with standard algorithms available on the CPU and GPU

Method

- Implement the algorithms on a CPU and a GPU
- Optimize the performance on the GPU
- Find avenues for loop perforation and try various levels of perforation
- Obtain the various performance evaluation criteria
- Conclude Results

Image Blurring

- We use Box Blur Algorithm
  - Spatial domain linear filter
  - Each pixel equal to average of its neighbors
  - low-pass ("blurring") filter
- Easily implementable on the GPU
- Ideal Example to study Perforation

Image Blurring – CPU

```
    bool perf = false;
    int perf_rate = 5;

    // We don't do any bluring around the boarder
    for (int x = one_filter_dim; x < src.rows-one_filter_dim; x+=i) // should d
    {
      for (int y = one_filter_dim; y < src.cols-one_filter_dim; y+=j)
      {
        if (perf && ((y % perf_rate) && (x % perf_rate)) == 0))
        {
          continue;
        }
        r_sum = 0.0; g_sum = 0.0; b_sum = 0.0;
        for (int f_x = x-one_filter_dim; f_x <= x+one_filter_dim; f_x++)
        {
          for (int f_y = y-one_filter_dim; f_y <= y+one_filter_dim; f_y++)
          {
            r_c = src.at<cv::Vec3b>(f_x,f_y)[0];
            g_c = src.at<cv::Vec3b>(f_x,f_y)[1];
            b_c = src.at<cv::Vec3b>(f_x,f_y)[2];
            r_sum += r_c; g_sum += g_c; b_sum += b_c;
          }
        }
        dst.at<cv::Vec3b>(x,y)[0] = r_sum/(filter_dims*filter_dims);
        dst.at<cv::Vec3b>(x,y)[1] = g_sum/(filter_dims*filter_dims);
        dst.at<cv::Vec3b>(x,y)[2] = b_sum/(filter_dims*filter_dims);
      }
    }
```

Image Blurring – GPU

```
__global__
void blur(unsigned char* input_image, unsigned char* output_image, int width, int height) {

    const unsigned int offset = blockIdx.x*blockDim.x + threadIdx.x;
    int x = offset % width;
    int y = (offset-x)/width;
    int fsize = 5; // Filter size
    if(offset < width*height) {

        float output_red = 0;
        float output_green = 0;
        float output_blue = 0;
        int hits = 0;
        for(int ox = -fsize; ox < fsize+1; ++ox) {
            for(int oy = -fsize; oy < fsize+1; ++oy) {
                if((x+ox) > -1 && (x+ox) < width && (y+oy) > -1 && (y+oy) < height) {
                    const int currentoffset = (offset+ox+oy*width)*3;
                    output_red += input_image[currentoffset];
                    output_green += input_image[currentoffset+1];
                    output_blue += input_image[currentoffset+2];
                    hits++;
                }
            }
        }
        output_image[offset*3] = output_red/hits;
        output_image[offset*3+1] = output_green/hits;
        output_image[offset*3+2] = output_blue/hits;
        }
}
```

Image Blurring – A comparison

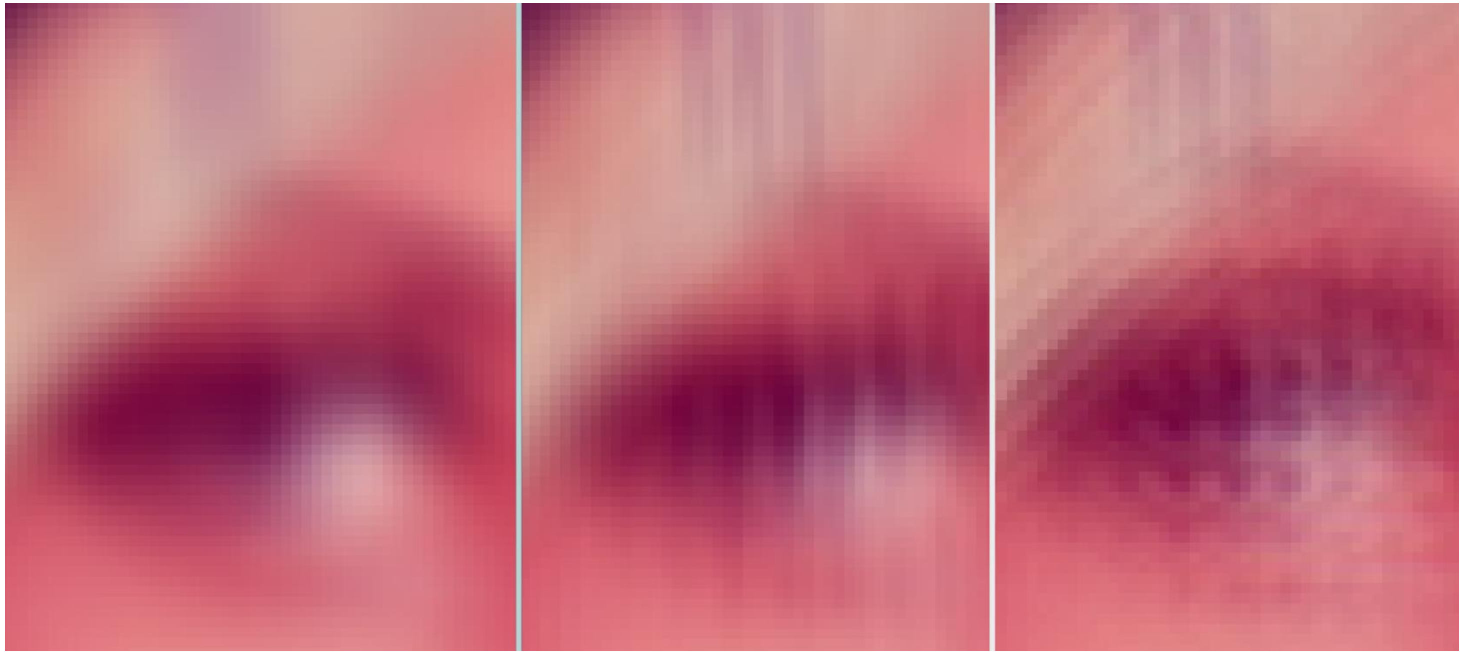
LOOP PERFORATION

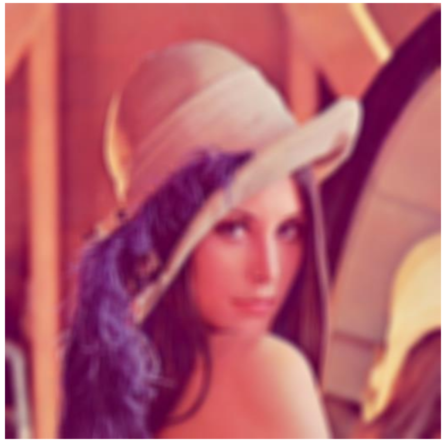
BLURING

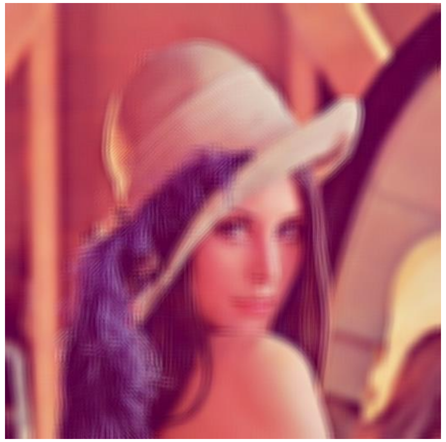
CPU
No Perf - Clock Time (for 50 iterations) = 40.019
loop+=2 - Clock Time (for 50 iterations) = 10.259
loop+=5 - Clock Time (for 50 iterations) = 1.611

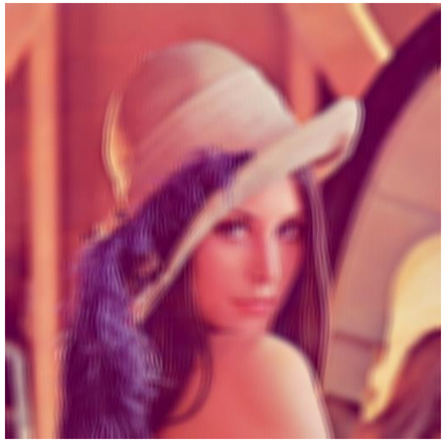
GPU -
No Perf - Clock Time (for 50 iterations) = 0.263085
loop+=2 - Clock Time (for 50 iterations) = 0.155025
loop+=5 - Clock Time (for 50 iterations) = 0.121848

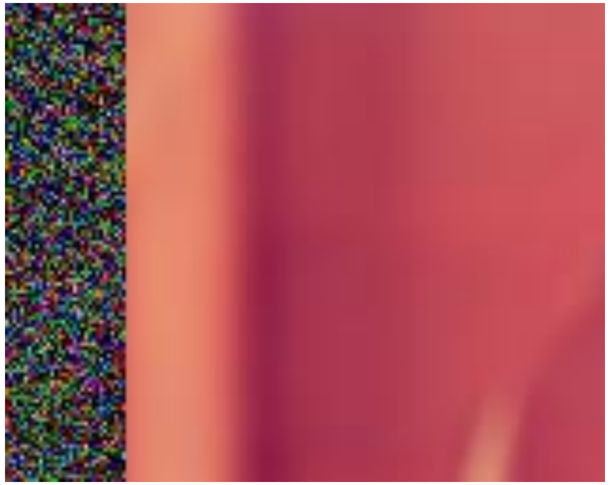
Different Types of Loop Perforation

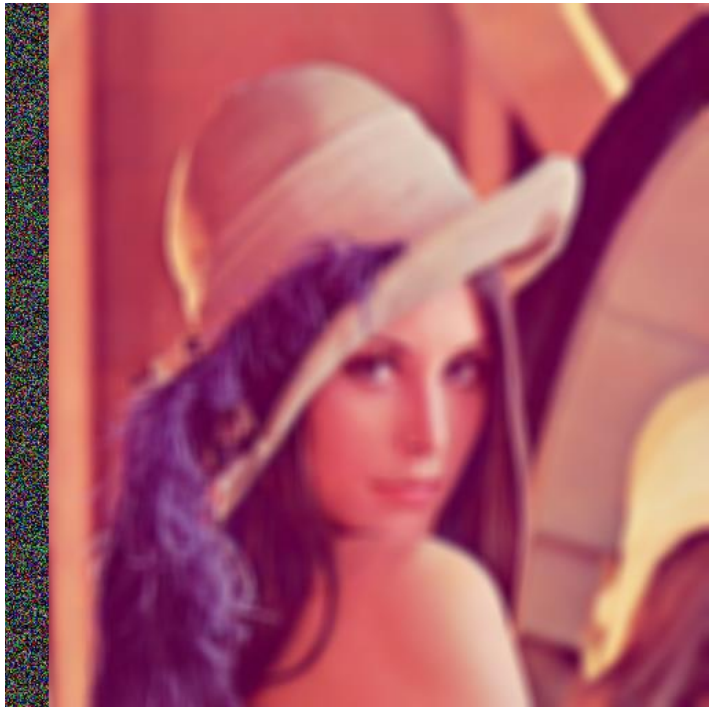




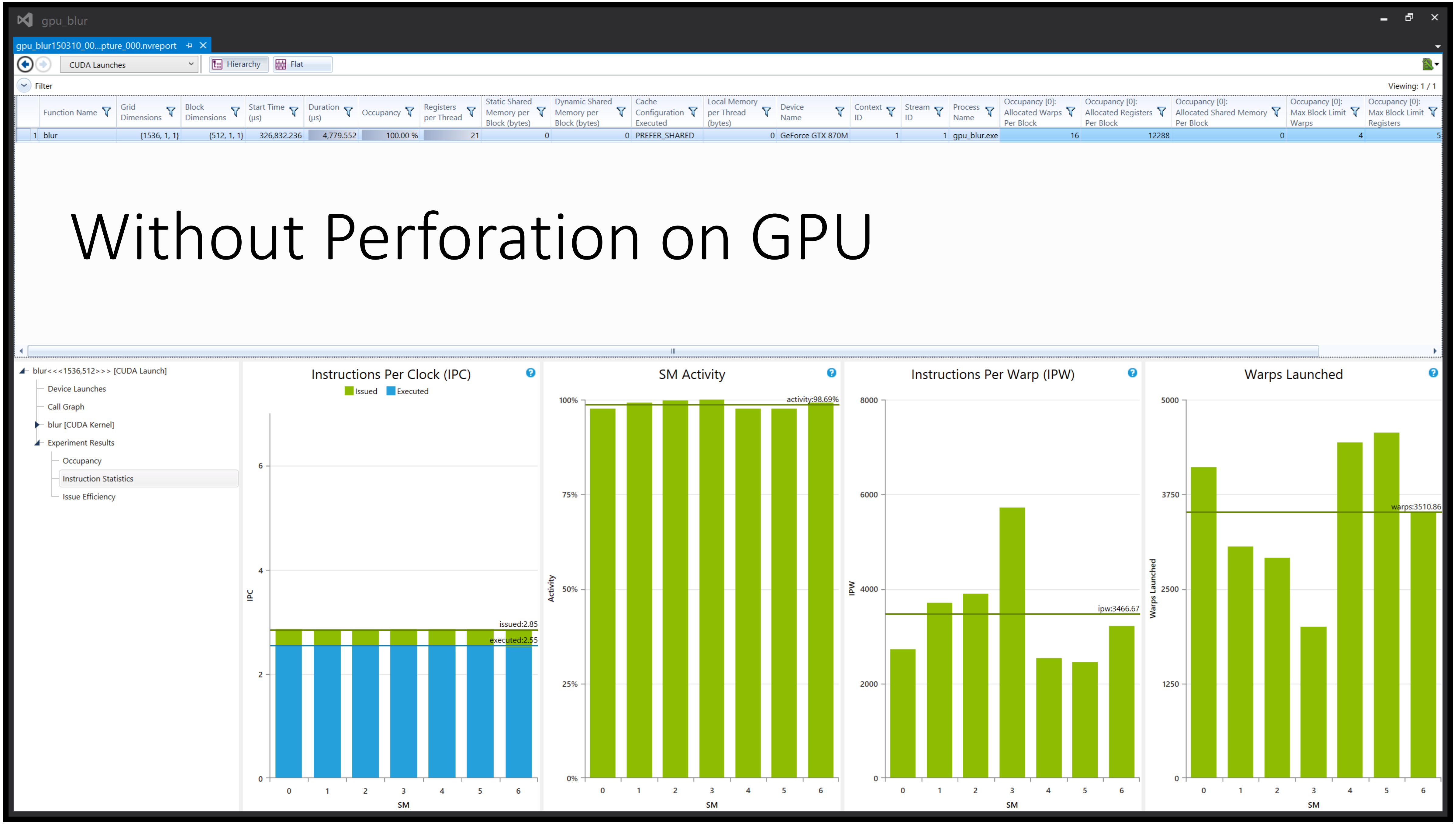








Without Perforation on GPU



With Perforation on GPU

Local Binary Patterns

- The Algorithm

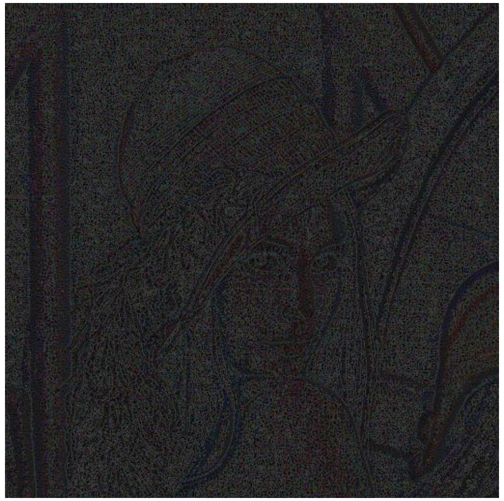
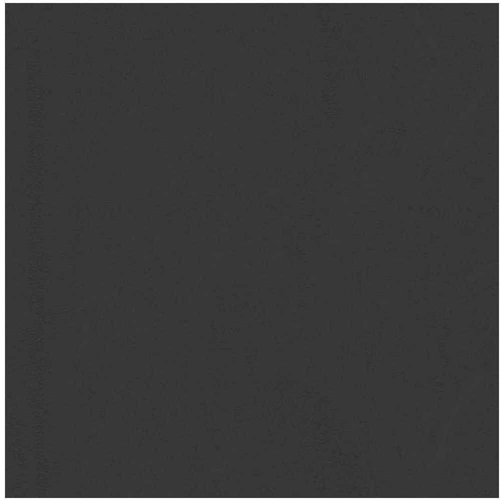
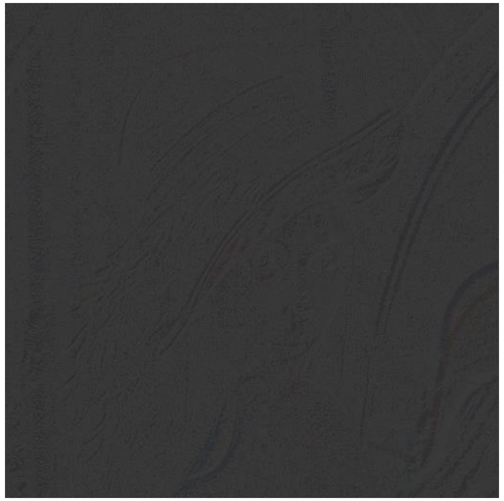
Local Binary Patterns – CPU

Local Binary Patterns – GPU

Local Binary Patterns – A comparison

CPU (Change Loop in LBP.cpp file - line 319)

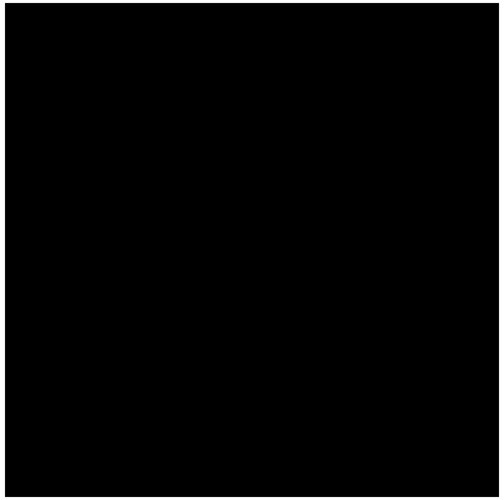
- No Perf - Example took 0.166892s
- loop+=2 - Example took 0.134939s
- loop+=3 - Example took 0.111867s

GPU -

- No Perf - Example took 0.063818s
- loop+=2 - Example took 0.104177s
- loop+=3 - Example took 0.0965s





Conclusions And Discussion

- Loop Perforation studied with Image blurring and LBP algorithms
- Used Various Methods for Loop Perforation
  - Throwing out Warps Not a Good Idea
  - Doing it on the CPU produces patterned results
- GPU Implementations Generally faster
  - For Blurring we got 150x Speed up (without perforation)
  - We got 200x Speed up (with Perforation +2)
  - We got 66x Speed up (with Perforation +5)
- Even Faster with Perforation
  - 1.7x Speed up (with Perforation +2)
  - 2.1x Speed up (with Perforation +5)

Future Work

- Try and add Power Consumption Statistics for the current results
- Try Perforation on a Feature Vector matching Algorithm
- Using perforated LBPs and Perforated Feature vector matching For face detection

Thank You

Resources Used

Bad Result CPU