Fast Bilateral Filter GPU implementation

Multi-Core Architectures and Programming

Gerhard Mlady, Rafael Bernardelli

Hardware/Software Co-Design, University of Erlangen-Nuremberg

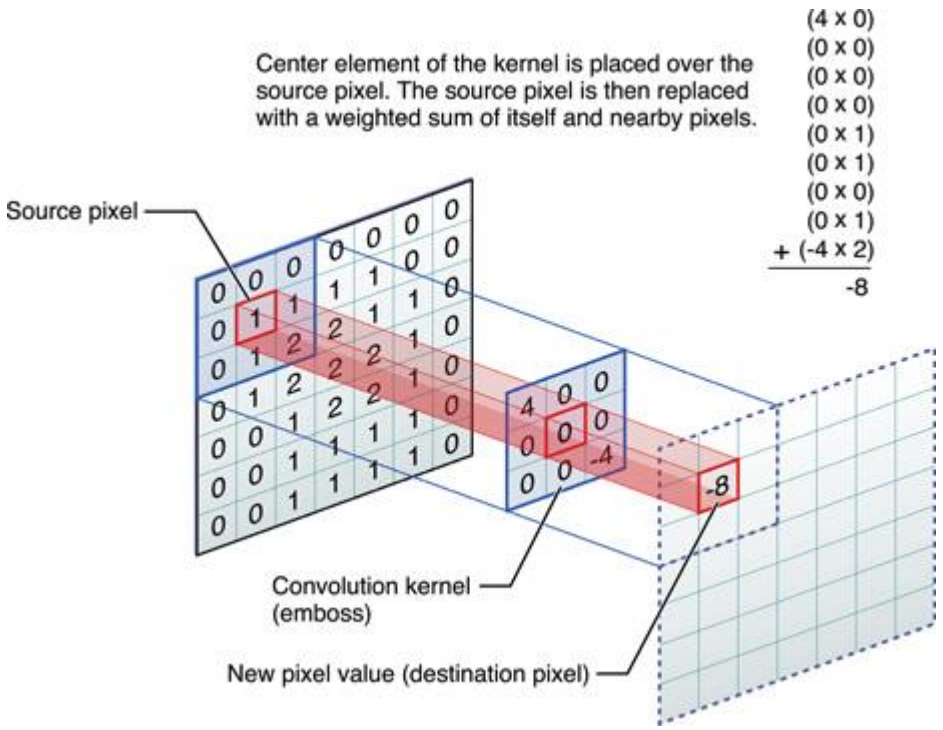
July 21, 2016

Overview

- **Fast Bilateral Filter**
- **Implementation**
- **Benchmark**

Introduction to Bilateral filter

First lets take a look on 2d convolution



$$f(x) = g(x) * k(x)$$

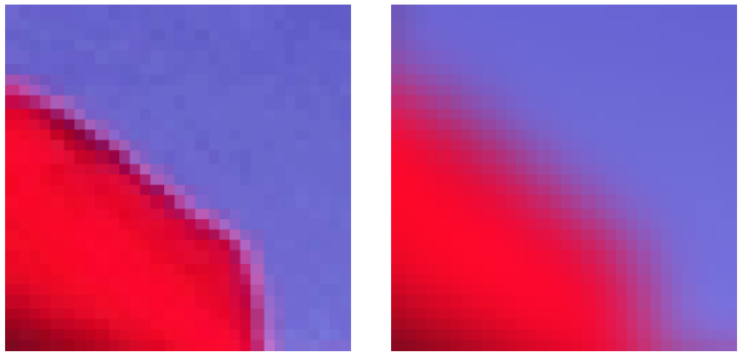
$$f(x) = \sum_{x' \in \mathcal{N}(n)} g(x')\, k(x - x')$$

$$f(x) = \mathcal{F}^{-1}\{\mathcal{F}\{g(x)\}\mathcal{F}\{k(x)\}\}$$

$$\frac{1}{273}$$

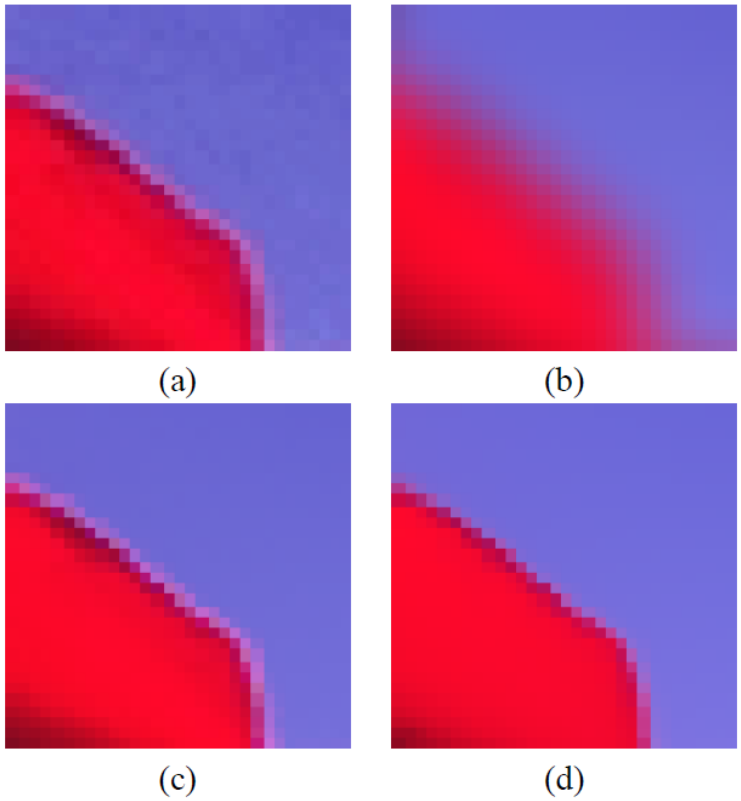
1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Gaussian kernel

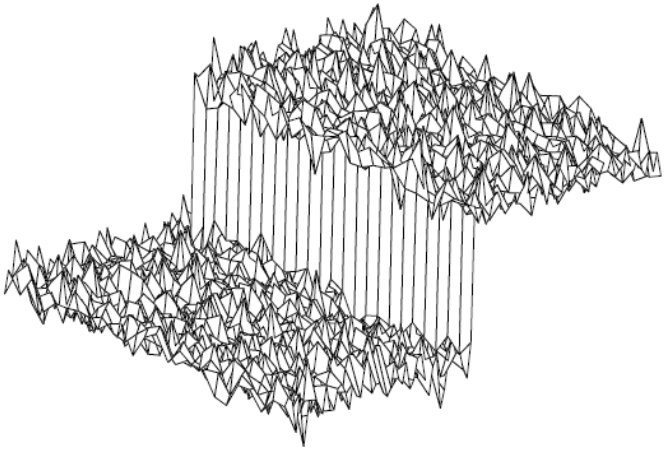


(a) (b)

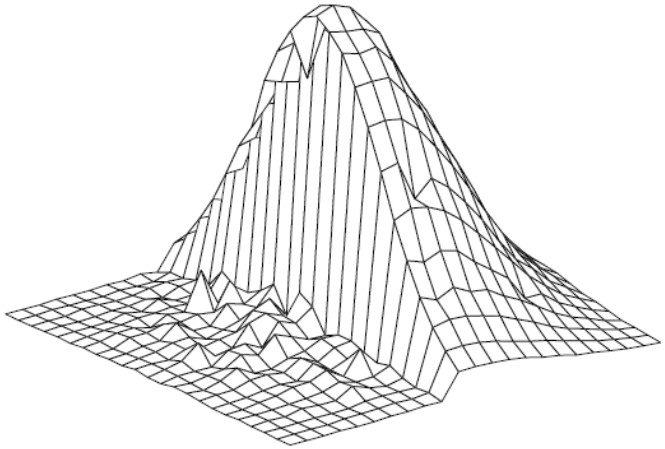
(a) Original image
(b) Gaussian filtered image



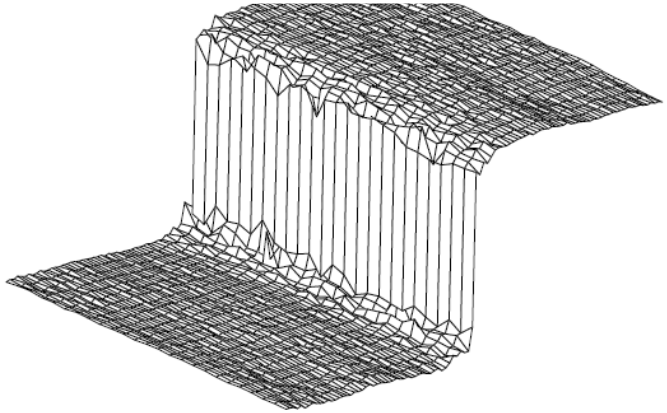
(a) Original image
(b) Gaussian filtered image
(c) Bilateral filtered image*
(d) Bilateral filtered image**



(a)



(b)



(c)

(a) Original image
(b) Bilateral kernel
(c) Result

Bilateral filter equations

$$f(\boldsymbol{x}) = k^{-1}(\boldsymbol{x}) \sum_{x' \in \mathcal{N}} g(\boldsymbol{x}')\, c(\boldsymbol{x}, \boldsymbol{x}')\, s(g(\boldsymbol{x}), g(\boldsymbol{x}'))$$

$f(\boldsymbol{x})$: Output image

$g(\boldsymbol{x})$: Input image

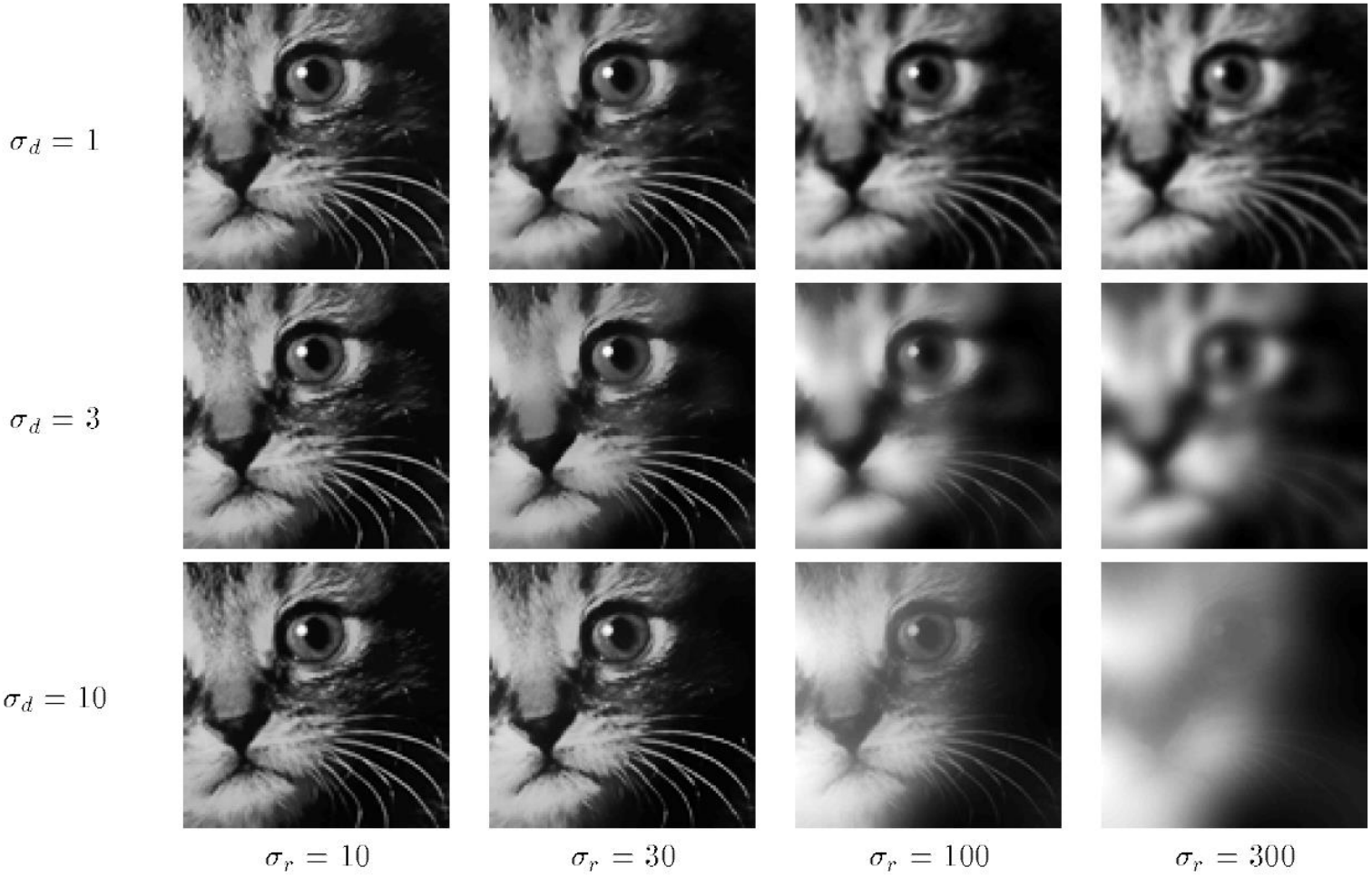
Bilateral filter equations

$$f(\boldsymbol{x}) = k^{-1}(\boldsymbol{x}) \sum_{x' \in \mathcal{N}} g(\boldsymbol{x'})\, c(\boldsymbol{x}, \boldsymbol{x'})\, s(g(\boldsymbol{x}), g(\boldsymbol{x'}))$$

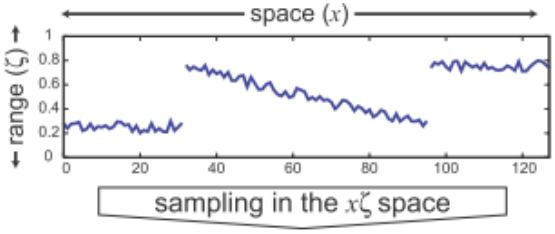
$$c(\boldsymbol{x}, \boldsymbol{x'}) = e^{-\frac{\|\boldsymbol{x}-\boldsymbol{x'}\|_2^2}{2\sigma_d^2}}$$ Spacial similarity function

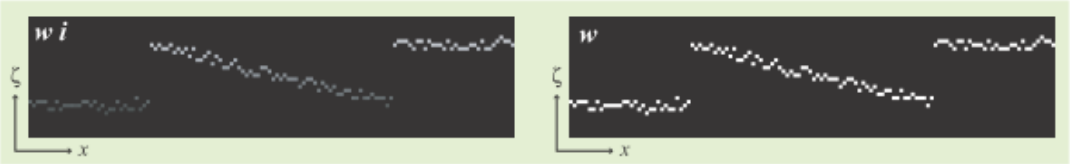
$$s(g(\boldsymbol{x}), g(\boldsymbol{x'})) = e^{-\frac{\left(g(\boldsymbol{x})-g(\boldsymbol{x'})\right)^2}{2\sigma_r^2}}$$ Range similarity function

$$k(\boldsymbol{x}) = \sum_{\boldsymbol{x'} \in \mathcal{N}} c(\boldsymbol{x}, \boldsymbol{x'})\, s(g(\boldsymbol{x}), g(\boldsymbol{x'}))$$ Normalizing factor

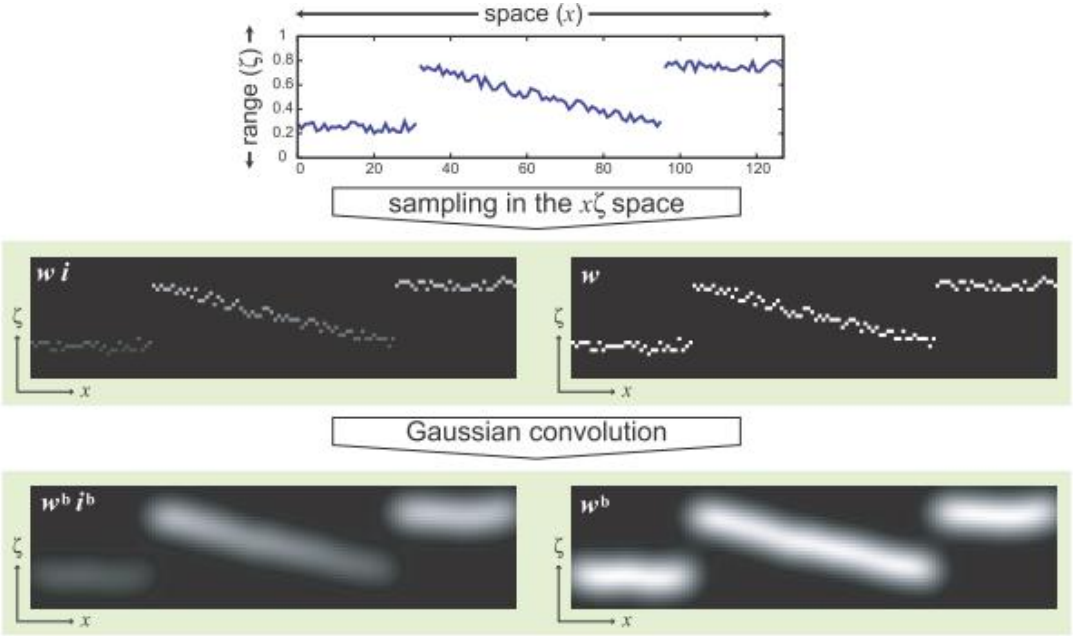

$\sigma_d = 1$
$\sigma_d = 3$
$\sigma_d = 10$
$\sigma_r = 10$
$\sigma_r = 30$
$\sigma_r = 100$
$\sigma_r = 300$

Fast Bilateral Filter

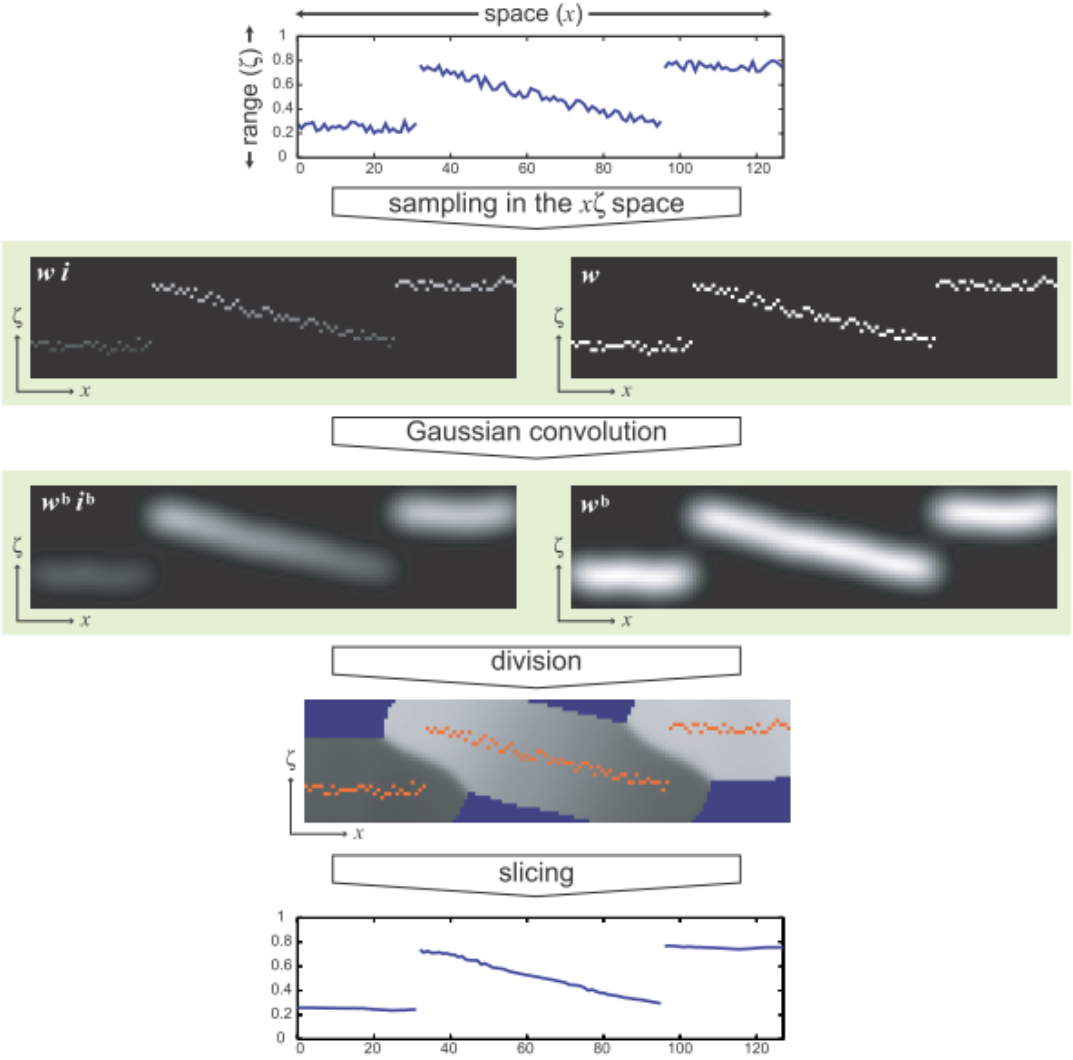




Fast Bilateral Filter



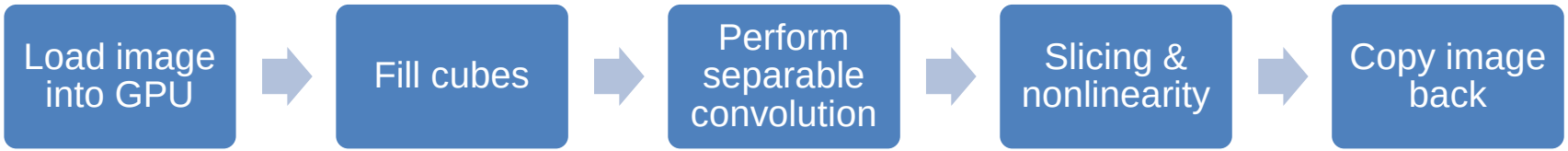
Fast Bilateral Filter



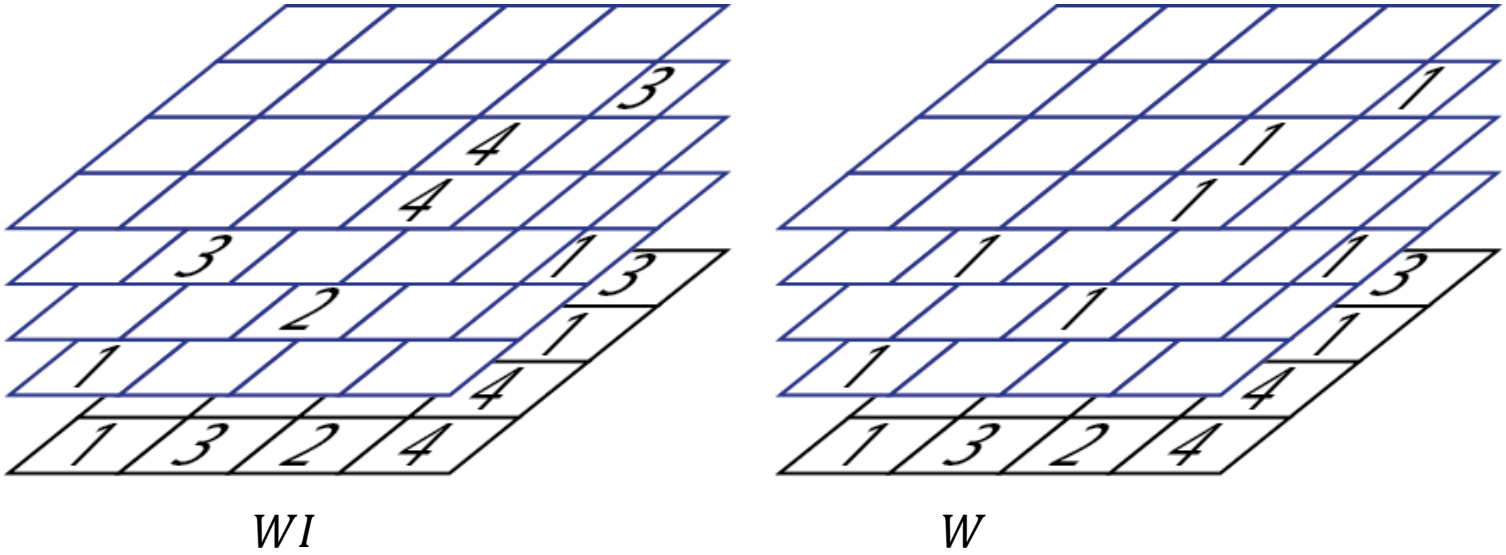
Overview

- Fast Bilateral Filter
- **Implementation**
- Benchmark

Implementation



Cube filling



WI W

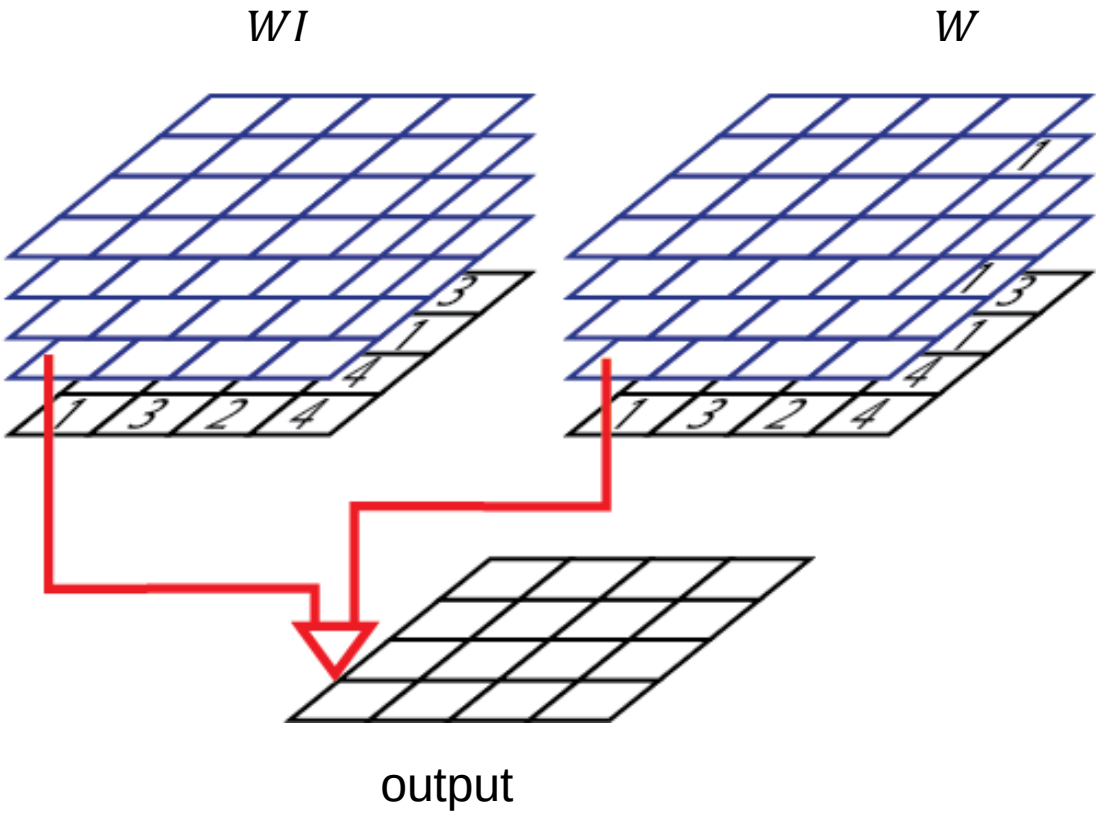
Perform separable convolution

$$I * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = I * \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

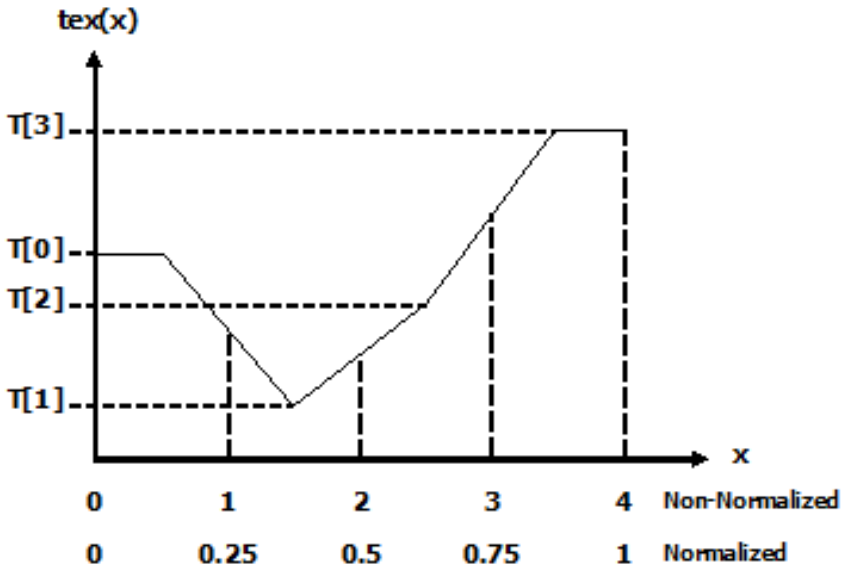
$O(NMk^2)$ $\qquad$ $O(2 * NMk)$

N, M : Image dimensions
k: convolution kernel length

Perform Slicing & Nonlinearity



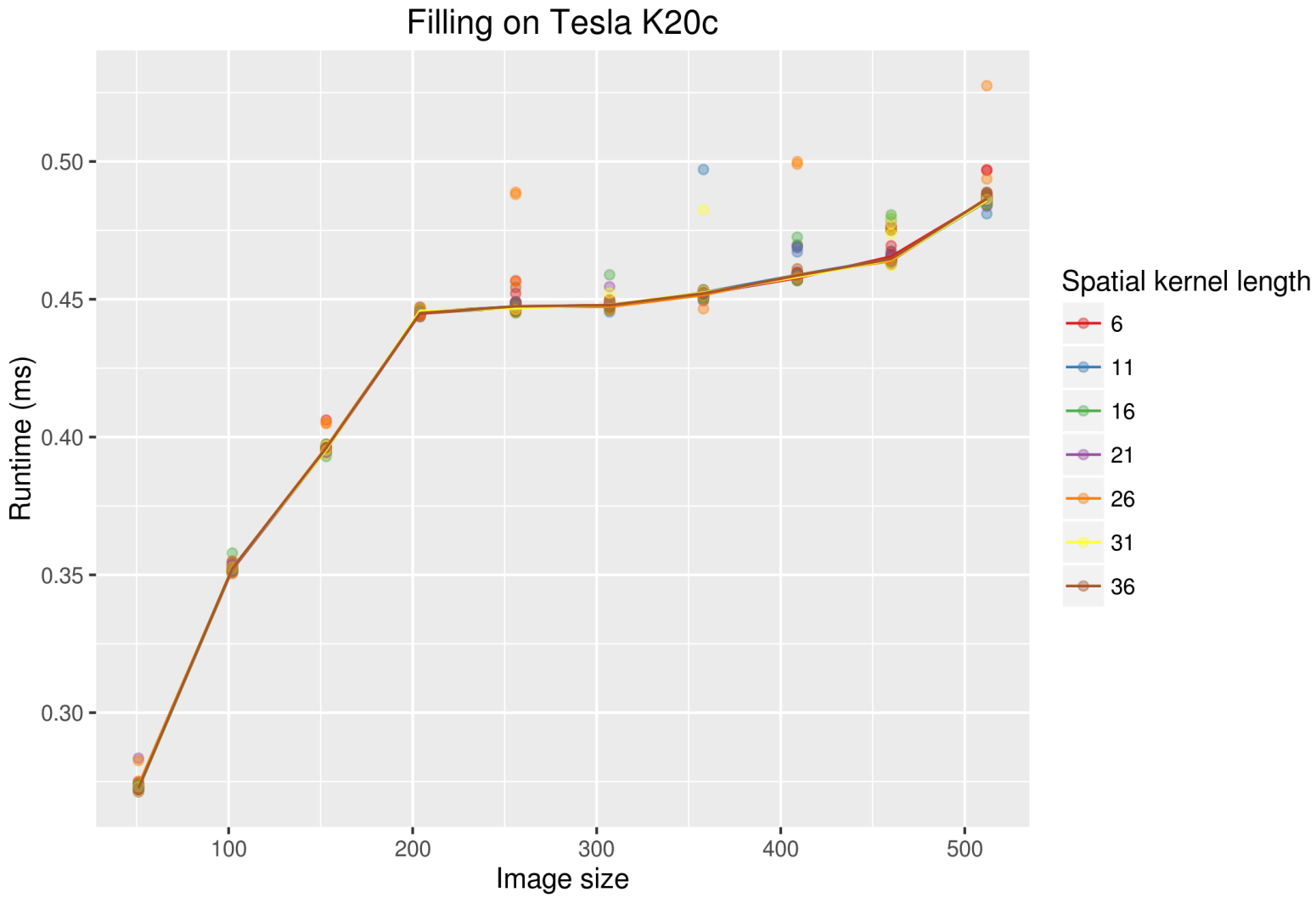
Texture fetching



Overview

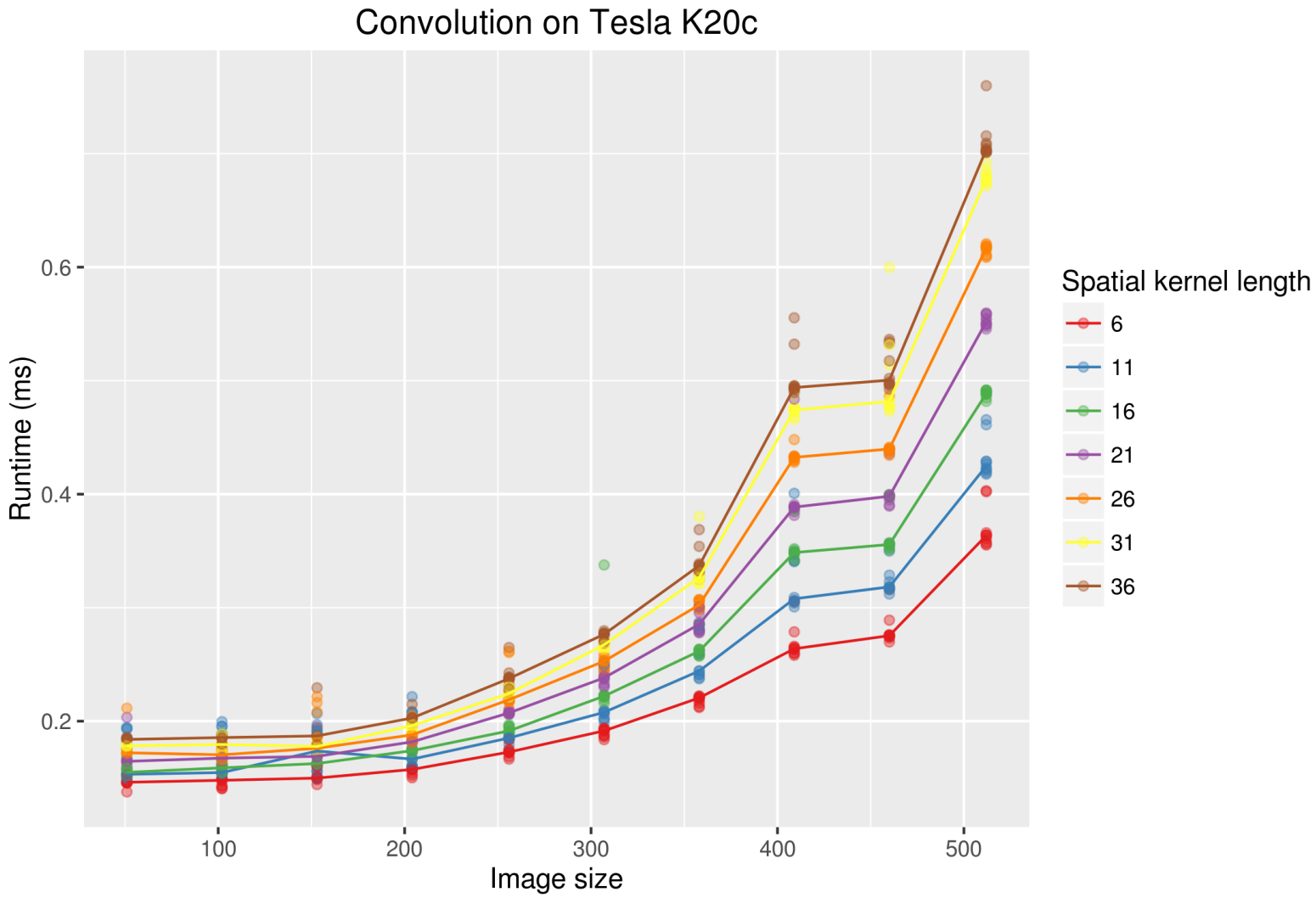
- Fast Bilateral Filter
- Implementation
- **Benchmark**

Benchmark



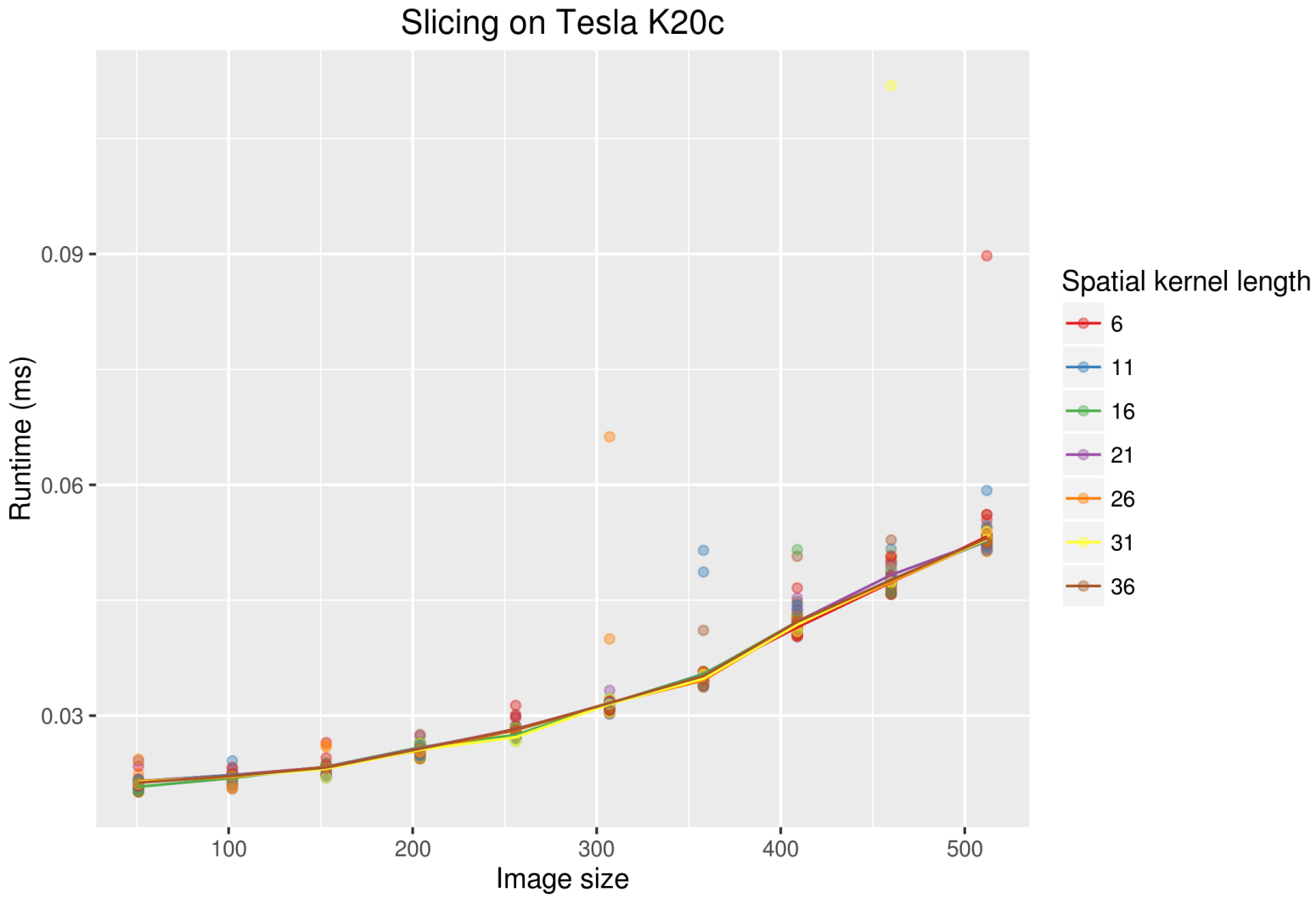
for intensity kernel length, intensity and spatial scalling 11

Benchmark



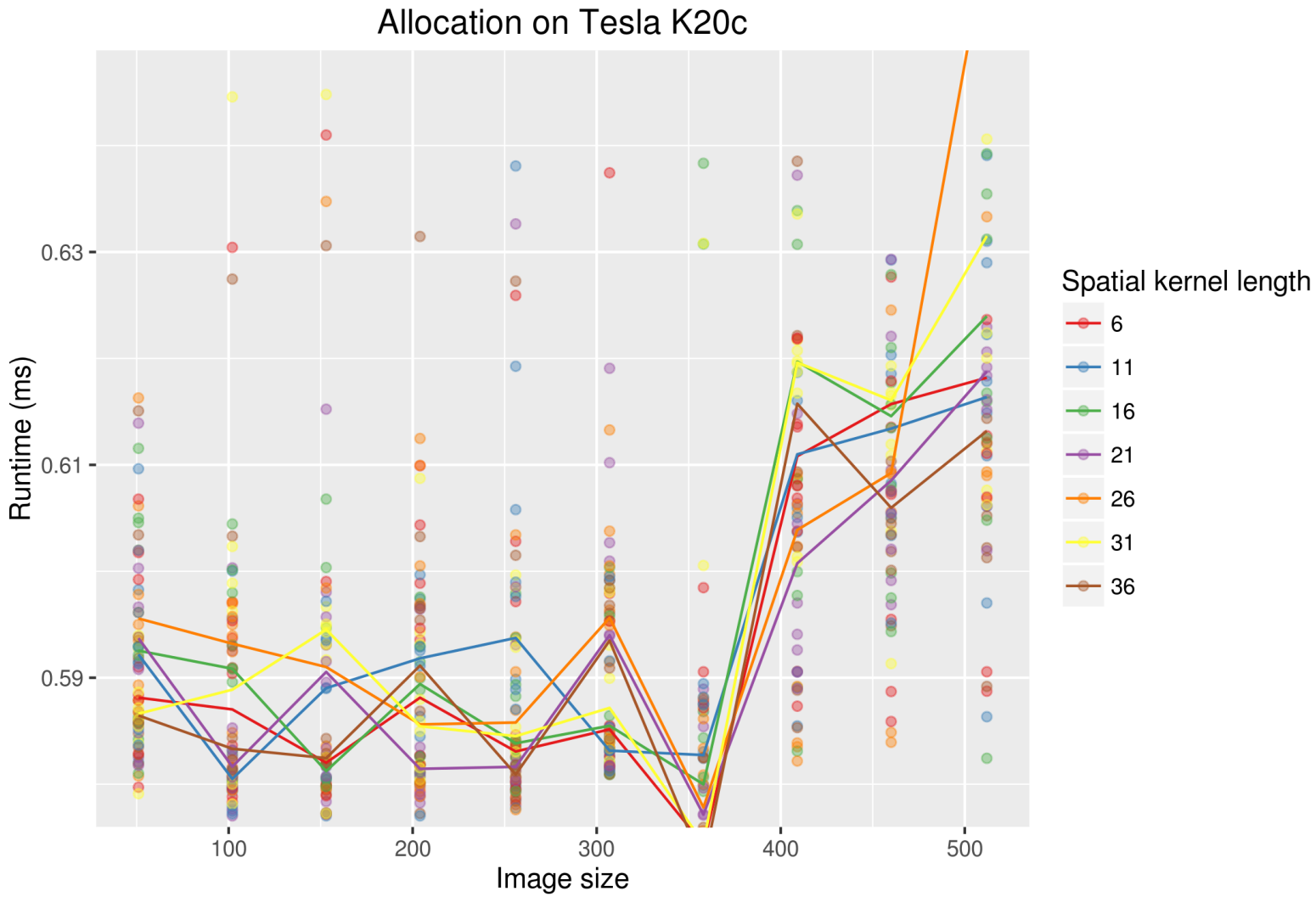
for intensity kernel length, intensity and spatial scalling 11

Benchmark



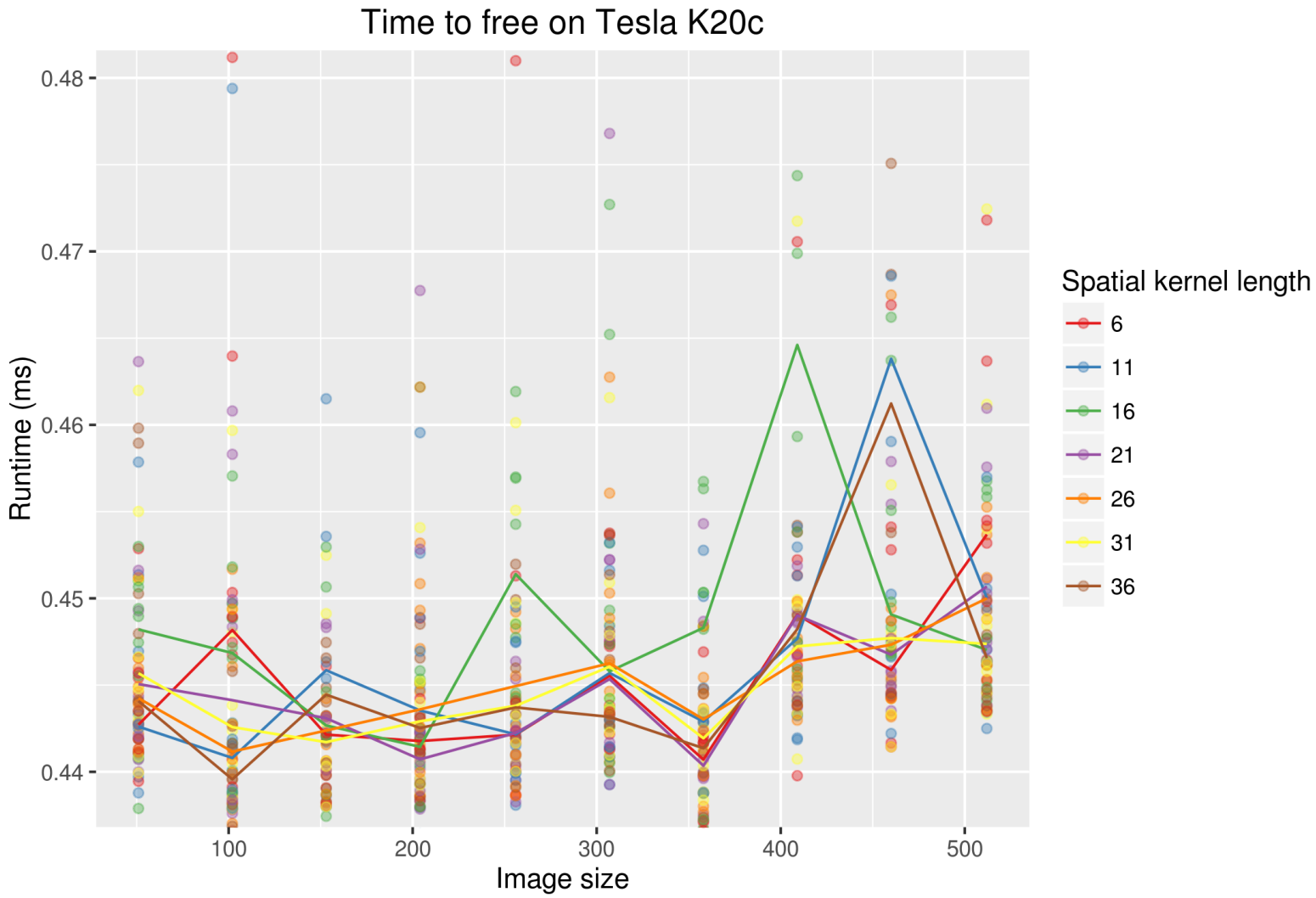
for intensity kernel length, intensity and spatial scalling 11

Benchmark



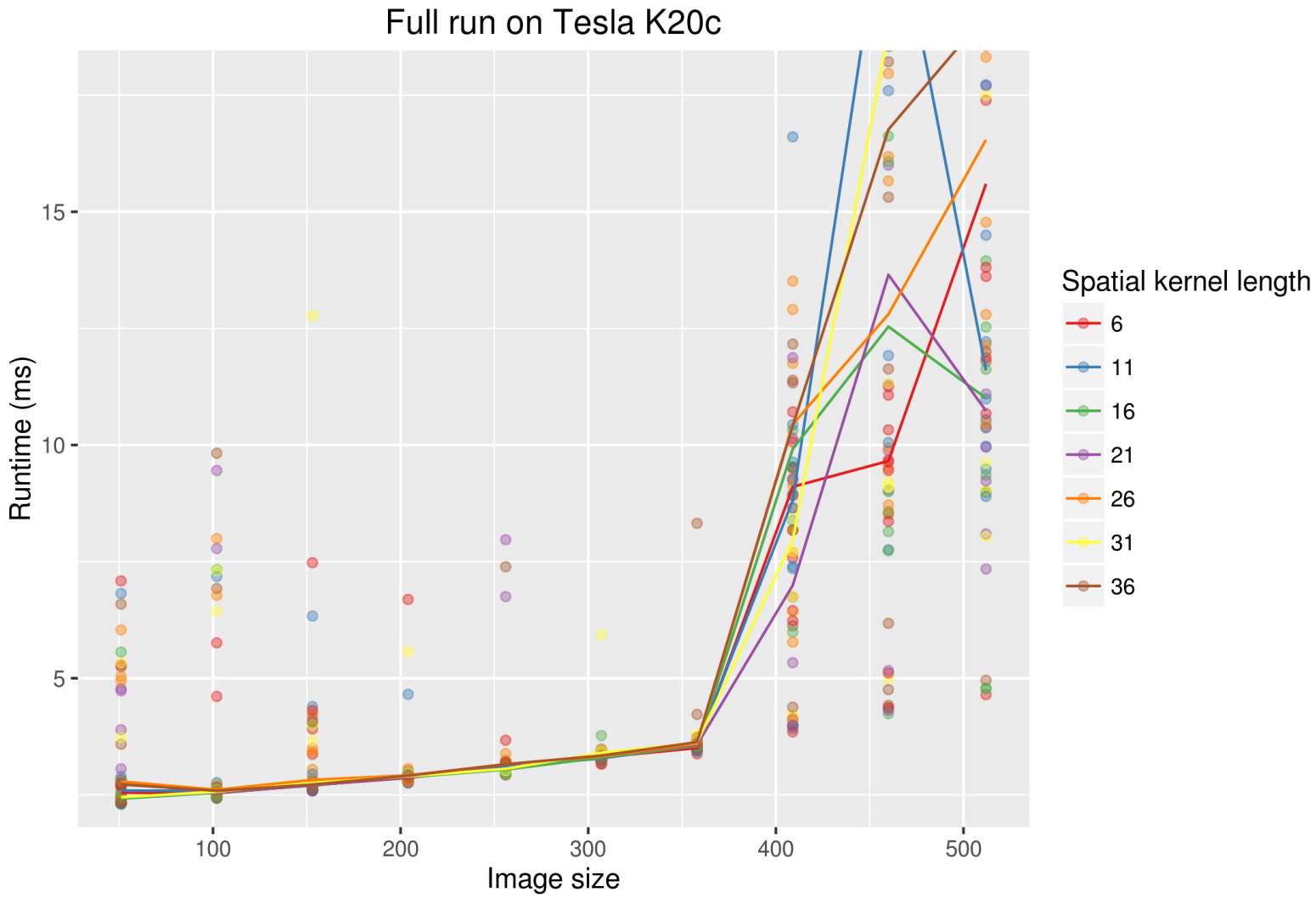
for intensity kernel length, intensity and spatial scalling 11

Benchmark



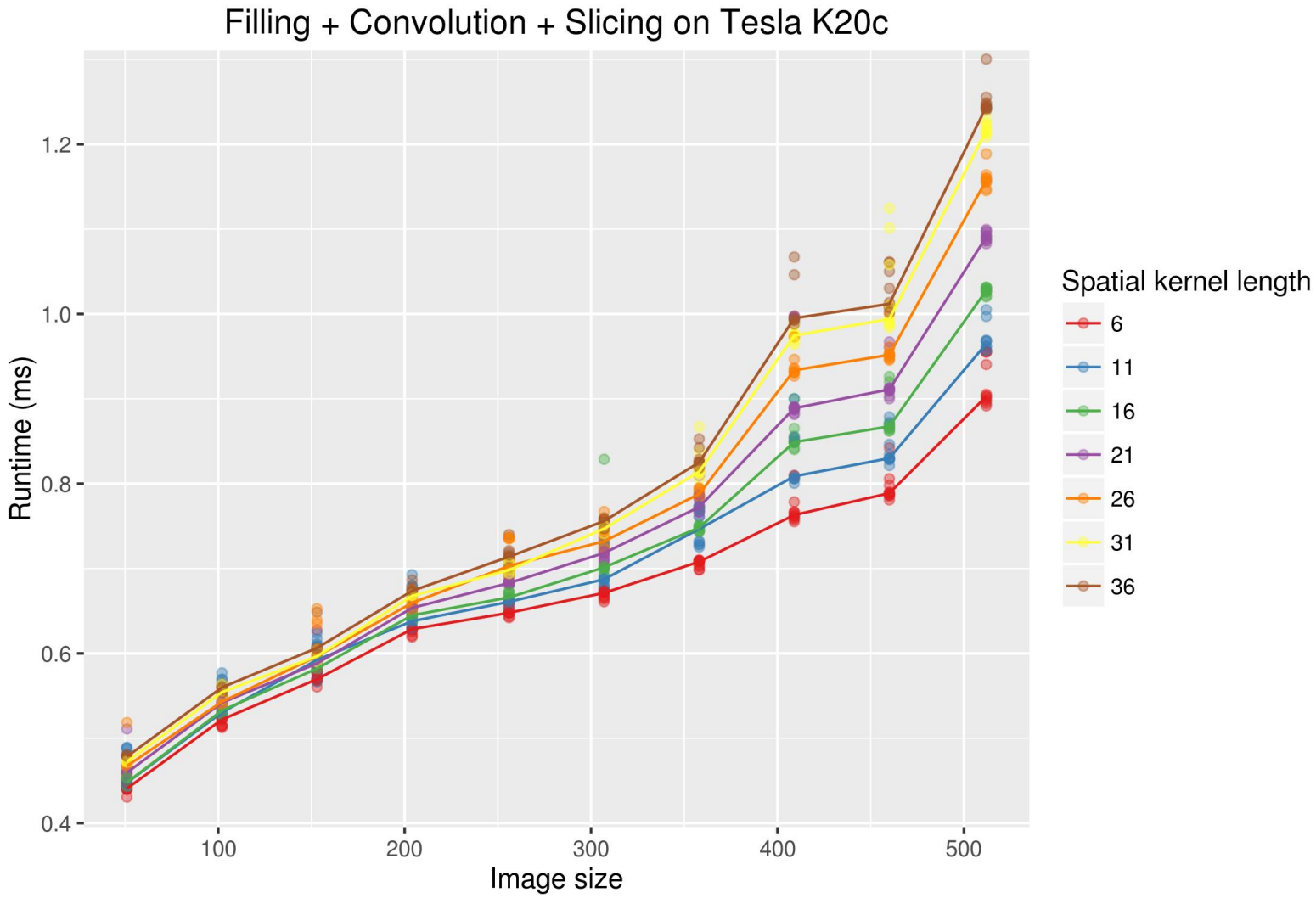
for intensity kernel length, intensity and spatial scalling 11

Benchmark



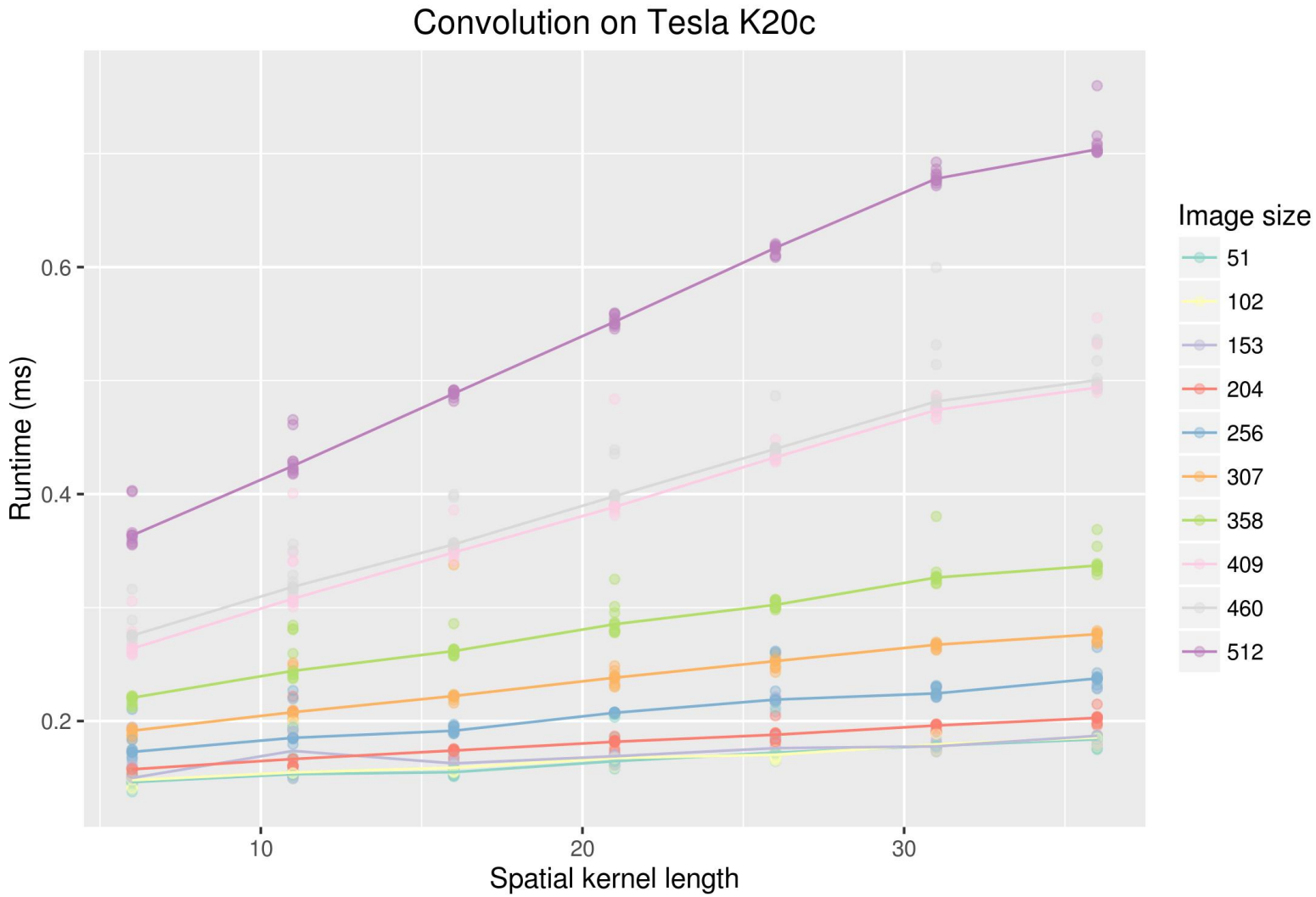
for intensity kernel length, intensity and spatial scalling 11

Benchmark



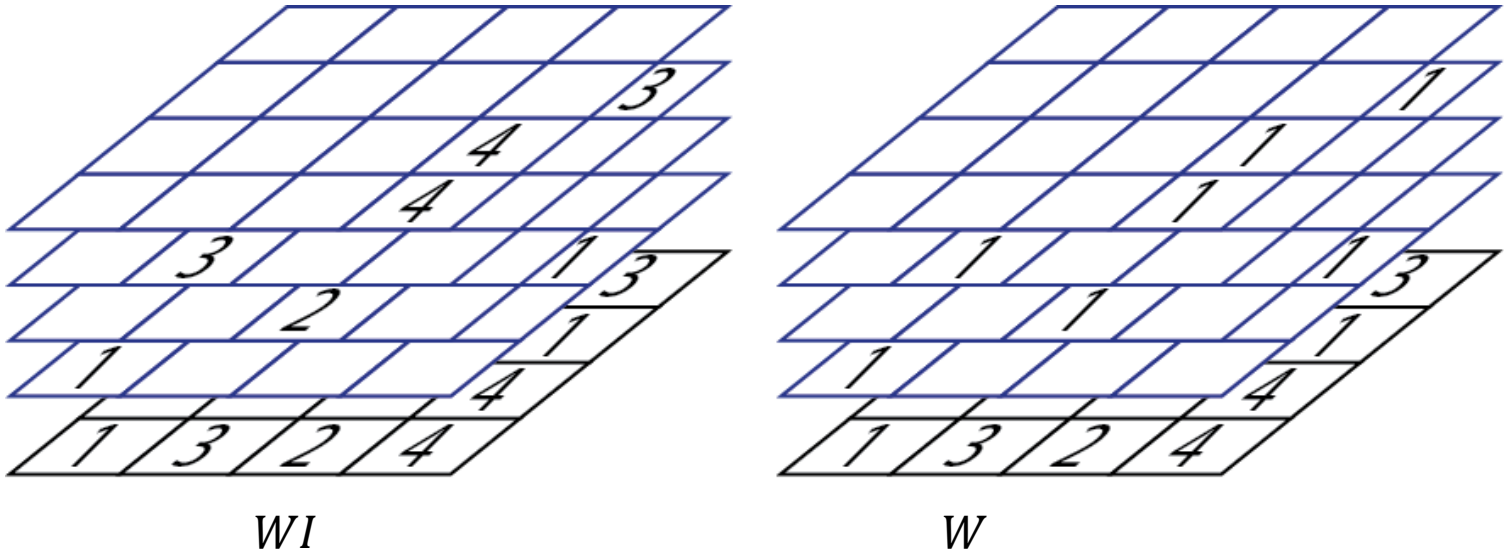
for intensity kernel length, intensity and spatial scalling 11

Benchmark



for intensity kernel length, intensity and spatial scalling 11

Benchmark – Best method for filling

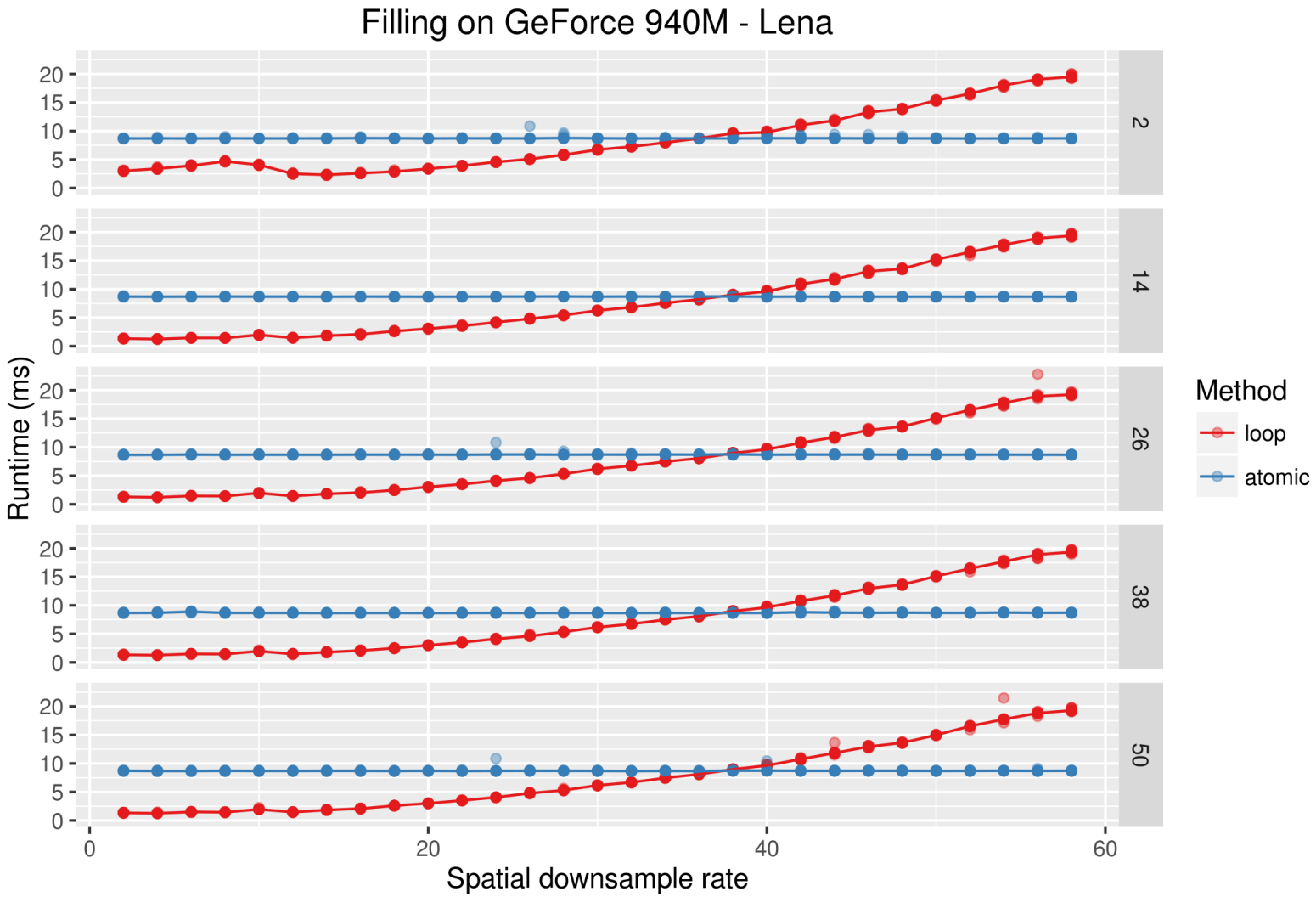


WI W

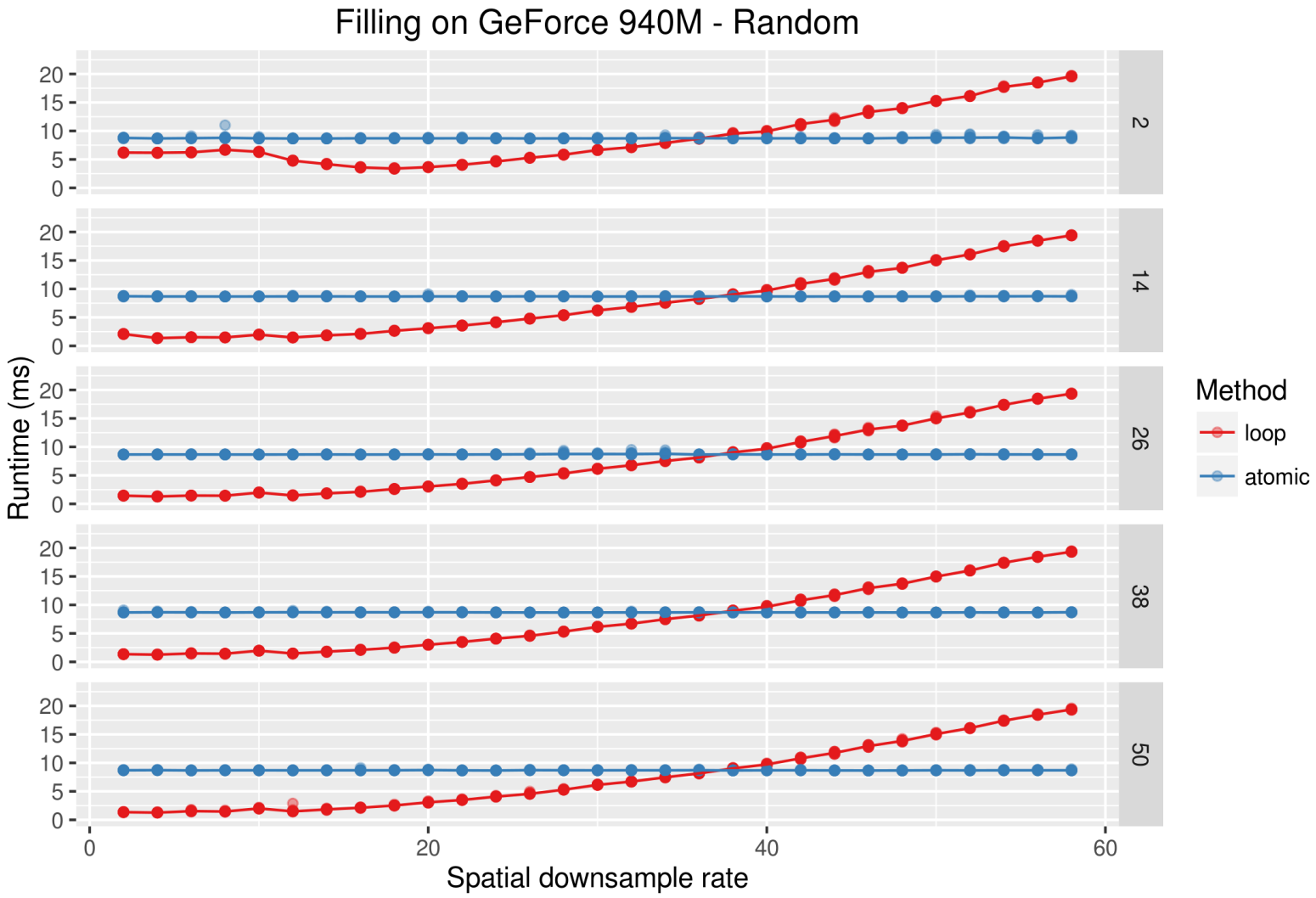
```
__global__ void cubefilling_loop(const float* image, float *dev_cube_wi, float *dev_cube_w, const dim3 image_size, int scale_xy, int scale_eps, dim3 dimensions_down)
{
        unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
        if (i < dimensions_down.x && j < dimensions_down.y) {

                size_t cube_idx_1 = i + dimensions_down.x*j;
#pragma unroll
                for (int ii = 0; ii < scale_xy; ii++)
                {
        #pragma unroll
                        for (int jj = 0; jj < scale_xy; jj++)
                        {
                                size_t i_idx = scale_xy*i + ii;
                                size_t j_idx = scale_xy*j + jj;
                                if (i_idx < image_size.x && j_idx < image_size.y)
                                {

                                        float k = image[i_idx + image_size.x*j_idx];
                                        size_t cube_idx_2 = cube_idx_1 +
dimensions_down.x*dimensions_down.y*floorf(k / scale_eps);

                                        dev_cube_wi[cube_idx_2] += k;
                                        dev_cube_w[cube_idx_2] += 1.0f;
                                }

                        }
                }
        }

}
```

```
__global__ void cubefilling_atomic(const float* image, float *dev_cube_wi, float *dev_cube_w, const dim3 image_size, int scale_xy, int scale_eps, dim3 dimensions_down)
{
        const size_t i = blockIdx.x * blockDim.x + threadIdx.x;
        const size_t j = blockIdx.y * blockDim.y + threadIdx.y;
        if (i < image_size.x && j < image_size.y) {
                const float k = image[i + image_size.x*j];
                const size_t cube_idx = (i / scale_xy) + dimensions_down.x*(j / scale_xy) + dimensions_down.x*dimensions_down.y*((int)k / scale_eps);


                atomicAdd(&dev_cube_wi[cube_idx], k);
                atomicAdd(&dev_cube_w[cube_idx], 1.0f);

        }


}
```

Benchmark – Best method for filling

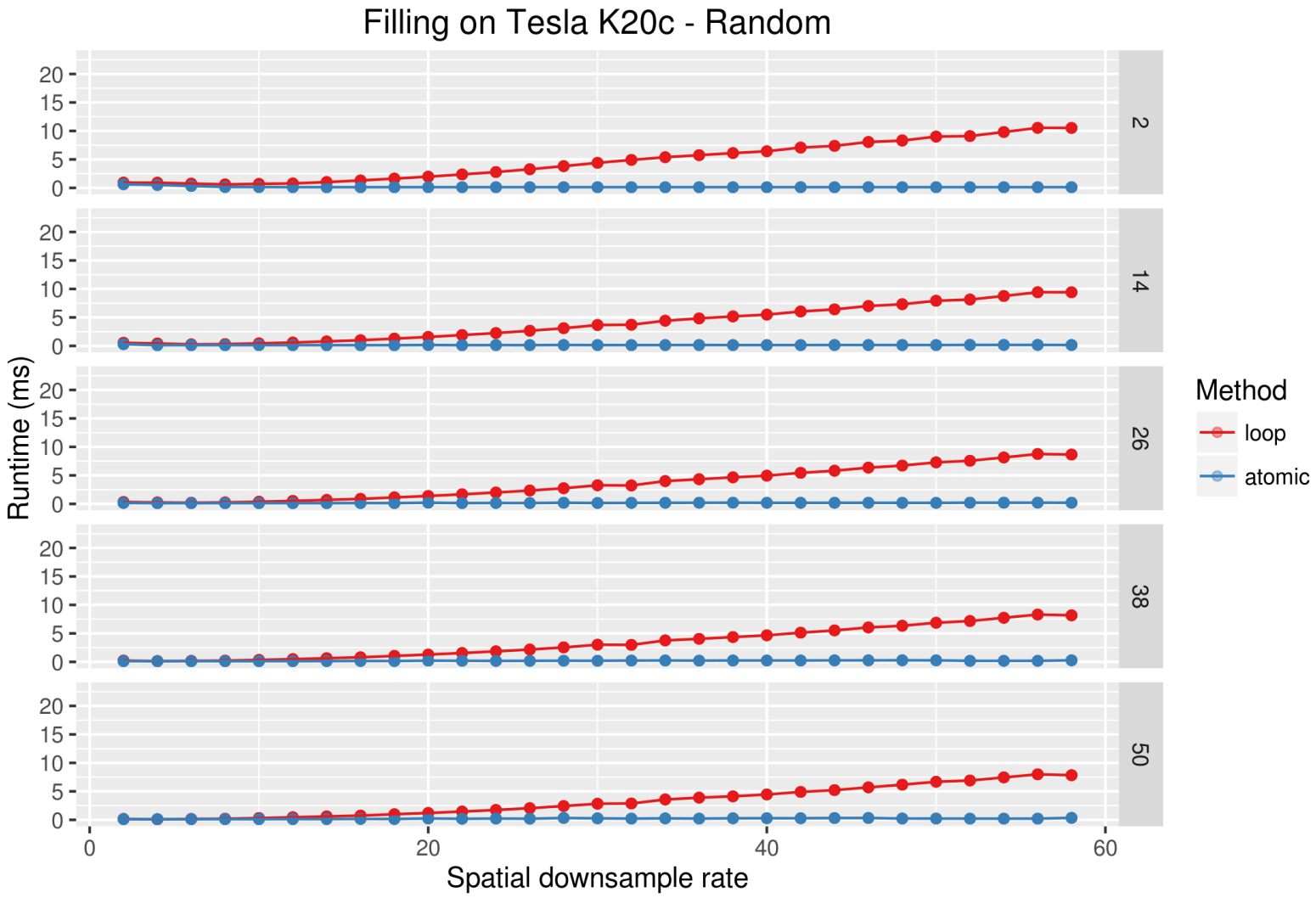


Benchmark – Best method for filling

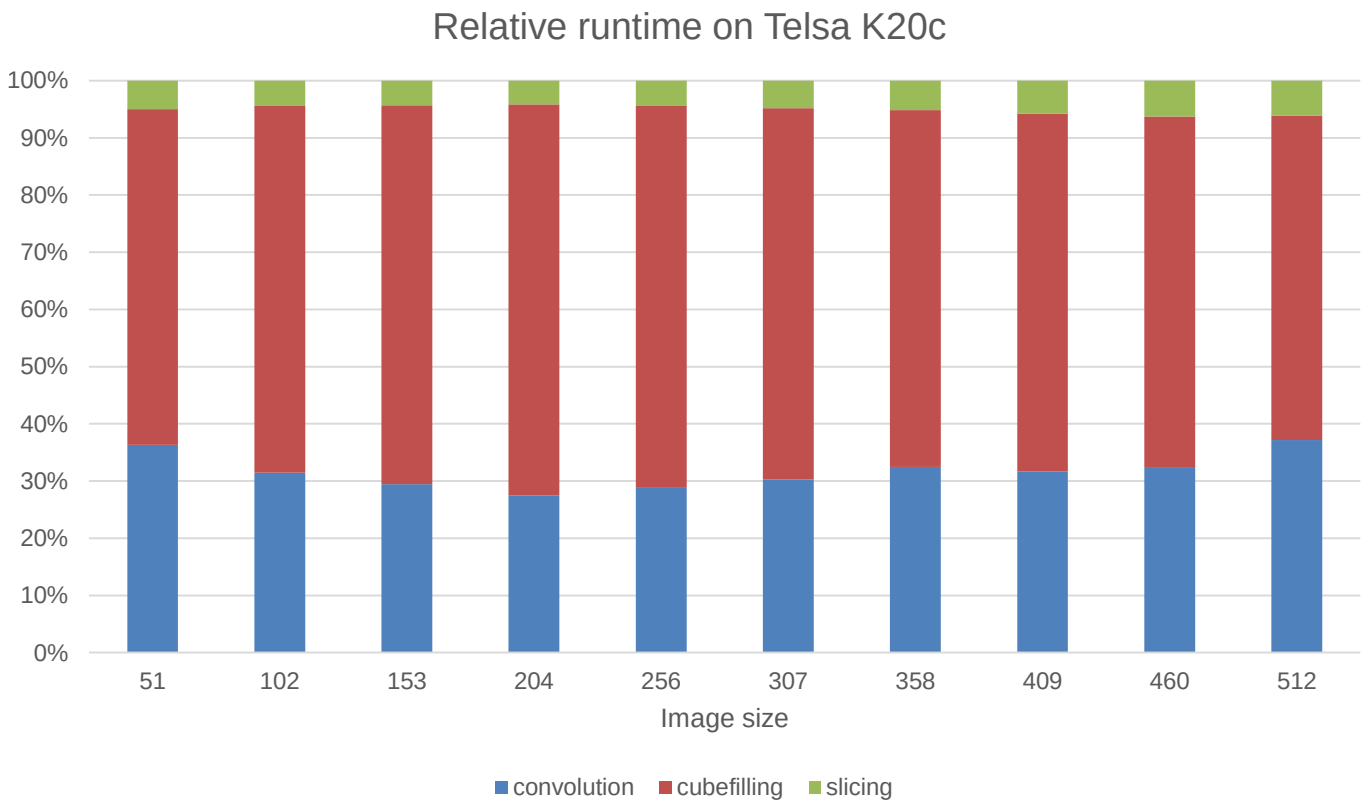


Benchmark – Best method for filling

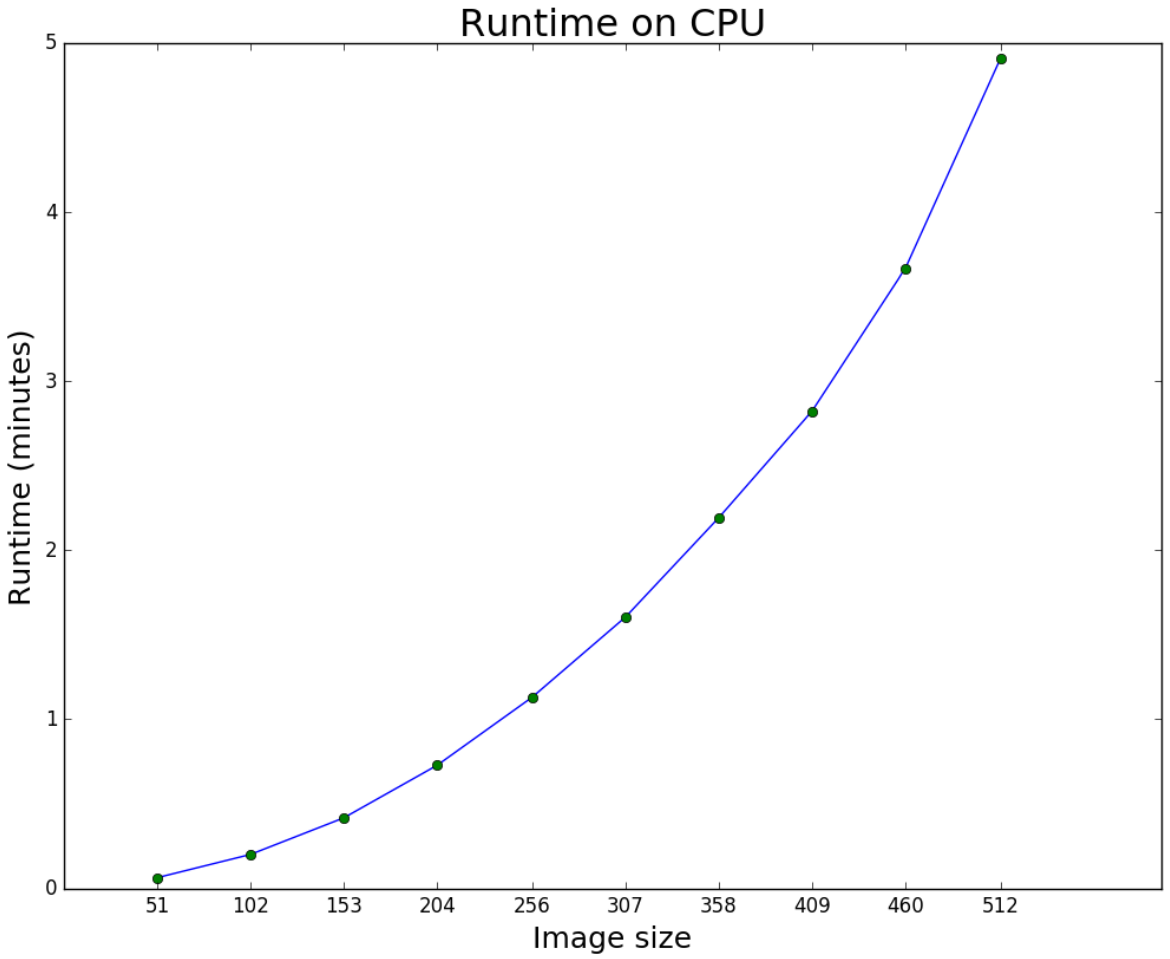
Benchmark – Best method for filling



Benchmark


Relative runtime on Telsa K20c
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
51
102
153
204
256
307
358
409
460
512
Image size
convolution
cubefilling
slicing

Benchmark - CPU



for intensity kernel length 21 and spatial k.l. 25

Issues on the Implementation

- On windows, a register key must be set to allow the GPU to run a kernel more than 2 seconds
  (HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers > TdrDelay=10)
- On (almost) every CUDA API call, retrieve `cudaStatus` and check for errors. It doesn't raise any exceptions.
- On linux, add some lines to **`.bashrc`**

```
export PKG_CONFIG_PATH=/scratch-local/usr/lib64/pkgconfig/
export PATH=$PATH:/opt/cuda/bin
export LD_LIBRARY_PATH=/scratch-local/usr/lib/:$(LD_LIBRARY_PATH)
```

If you are interested on the code and more plots…

- https://github.com/bernardelli/Bilateral-Filter-GPU

Thank You For Your Attention!

References

- https://developer.apple.com/library/prerelease/content/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html
- http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm
- C. Tomasi and R. Manduchi **Bilateral Filtering for gray and color images.** In *Proceedings of IEEE International Conference on Computer Vision*, pages 839-846
- http://people.csail.mit.edu/sparis/publi/2006/eccv/Paris_06_Fast_Approximation.pdf
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#linear-filtering__linear-filtering-of-1-d-texture-of-4-texels