# Resizing images using NVIDIA GPUs (OpenCV vs. PyTorch vs. TensorFlow)

**Contents**



## Introduction

When it comes to deep learning, it is (almost) somewhat standard that the pre-processing of images happens on host memory using CPUs and training happens on GPUs. Well, depending what we're going to do and what resources we have available there are a few reasons to do both on the GPU. A more critical thing is usually running a model in inference. It is very easy to fuck up a rather fast pipeline completely by introducing a performance bottleneck that may cut frames per seconds in half and therefore causes insufficient hardware utilization. The "image resize/down size component" that exists in many pipelines is often neglected when it comes to finding bottlenecks.

A proper pipeline deployed for any inference scenario should be benchmarked properly and adapted to a specific use case. There is no one size fits all solution. When using CUDA, it should be considered to put as much as possible on the GPU to minimize latencies and maximize throughput. This could be done using a form of OpenCV CUDA integration or on a lower level. When it comes to resizing an image for inference, we're basically having the following choices:

- resizing happens on a capture device using a FPGA
- resizing an image using a CPU (using an interpolation algorithm)
- resizing an image using memory views/pointers on host memory
- resizing an image using both options on a GPU

## Setup Notes

Luckily, OpenCV, PyTorch and TensorFlow provide interpolation algorithms for resizing so that we can compare them easily (using their respective Python APIs). Using randomly generated images of type `float32` of interval [0,1] of different sizes prevents caching of image data. The sizes are:

```
image_dims = [[800,600],\
              [1280,720],\
              [1920,1080],\
              [2560,1440],\
              [4096,2160],\
              [3840,2500],\
              [4096,3000],\
              [5456,3076],\
              [6464,4852],\
              [7680,4320],\
              [8192,5120]]
```

Software versions used are:

```
OpenCV: 4.5.5
PyTorch: 1.11.0
Tensorflow: 2.8.0
```

Let’s have a look at these rescale functions for each software library. OpenCV has to be compiled with full CUDA support to run these operations. The difference to rescaling it on the CPU is pretty small. The image has to be availabe as type `GpuMat` on the GPU and the rest is similar to what we are used to using the CPU. A big difference compared to the two deep learning frameworks is that OpenCV requires 2 or 3D inputs as images.

```
# opencv

import cv2

interpolation_methods = {}
interpolation_methods["bilinear"] = cv2.INTER_LINEAR
interpolation_methods["nearest"] = cv2.INTER_NEAREST
interpolation_methods["area"] = cv2.INTER_AREA
interpolation_methods["bicubic"] = cv2.INTER_CUBIC

...

img_out = cv2.cuda.resize(img_cu,
(608,608),interpolation=interpolation_methods[i])
```

When using PyTorch the situation is a bit different. Most functions seem to require a 4D tensor (batch,channels,height,width) and require floating point tensors as input data.

```
# pytorch

import torch.nn.functional as F

interpolation_methods = {}
interpolation_methods["bilinear"] = "bilinear"
interpolation_methods["nearest"] = "nearest"
interpolation_methods["area"] = "area"
interpolation_methods["bicubic"] = "bicubic"

...

img_out = F.interpolate(img,size=(608,608),mode=interpolation_methods[i])
```

The situation is similar for TensorFlow which provides `tf.image.resize`. However, the input tensor requires a different ordering (batch, height, width, channels) and 3D Tensors seem to be supported as well (height,width,channels).

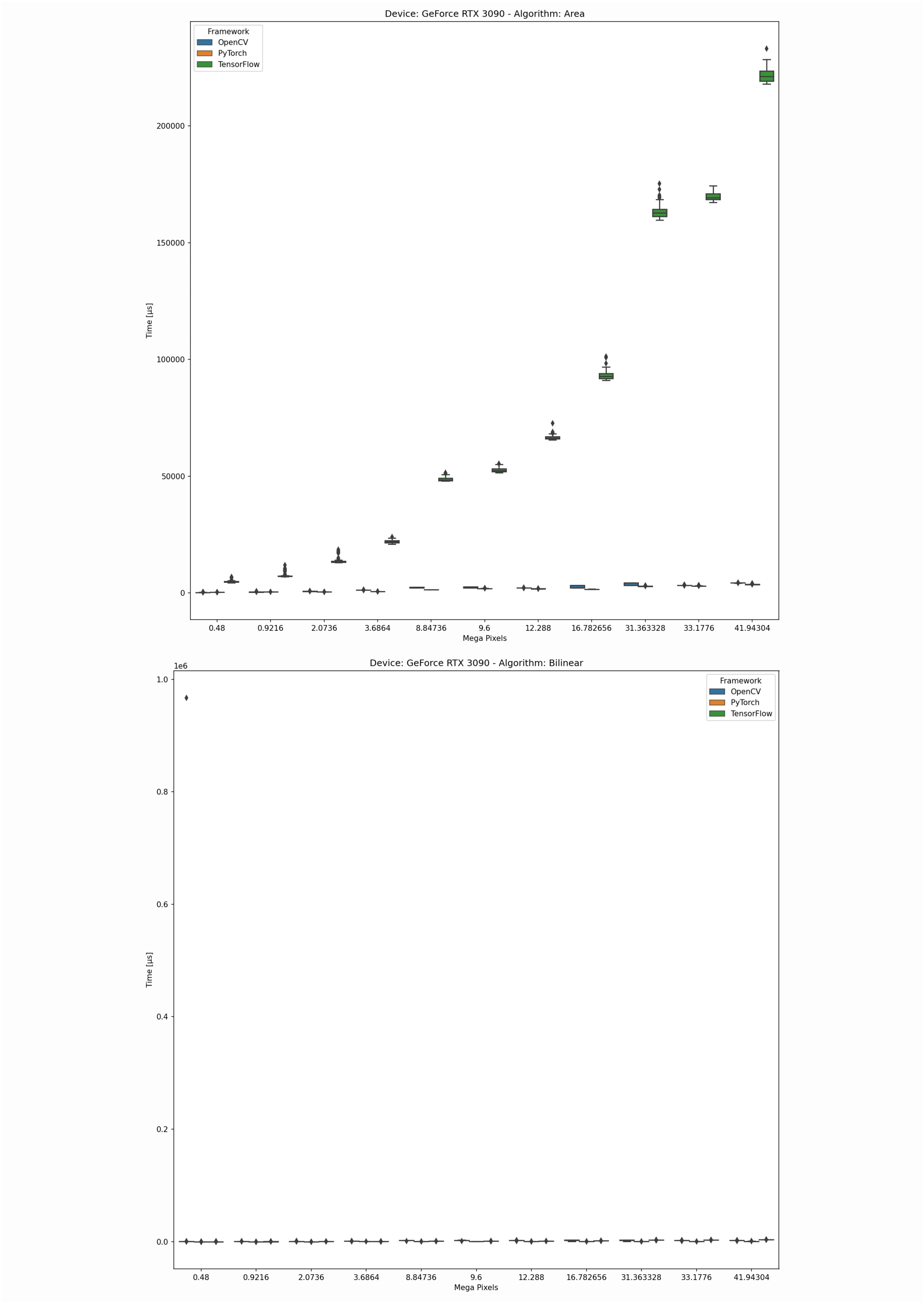
```
# tensorflow

import tensorflow as tf

interpolation_methods = {}
interpolation_methods["area"] = tf.image.ResizeMethod.AREA
interpolation_methods["bicubic"] = tf.image.ResizeMethod.BICUBIC
interpolation_methods["bilinear"] = tf.image.ResizeMethod.BILINEAR
interpolation_methods["nearest"] = tf.image.ResizeMethod.NEAREST_NEIGHBOR

...

img_out = tf.image.resize(img, size=(608,608), antialias=False,
method=interpolation_methods[i])
```

The results are based on measuring 100 iterations for each combination of framework, method and algorithm.

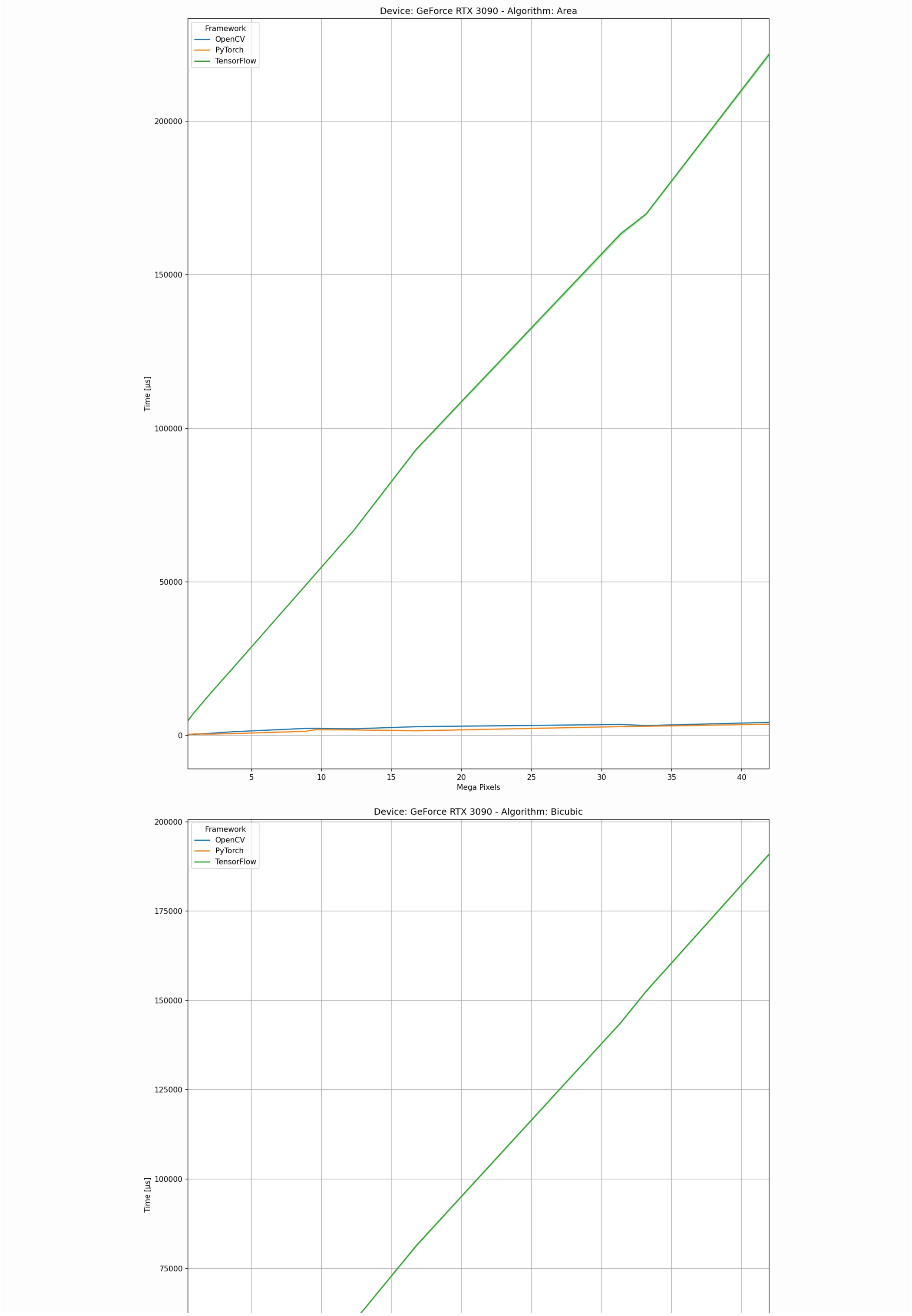
## GPU Results

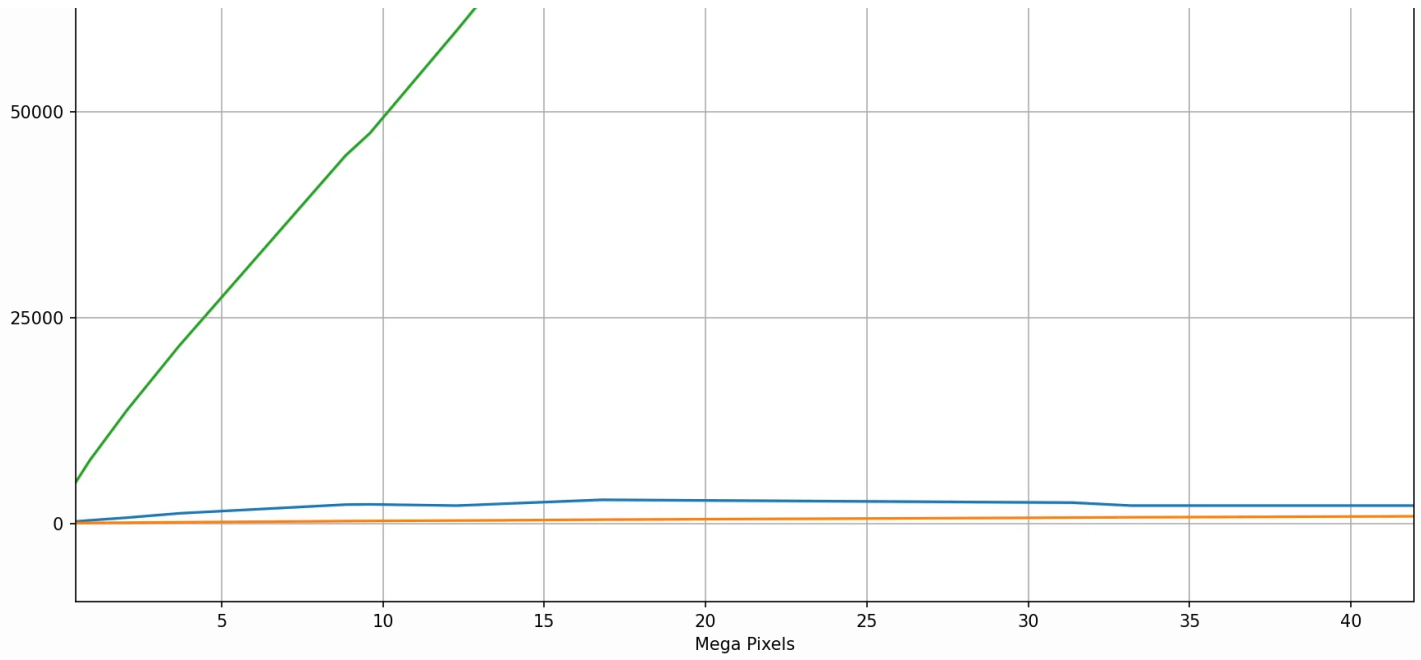
I used a GeForce RTX 3090 to obtain the GPU results. Other NVIDIA GPUs such as A5000s or older generation GeForce cards show similar results. Depending on algorithm and image size results vary a bit.

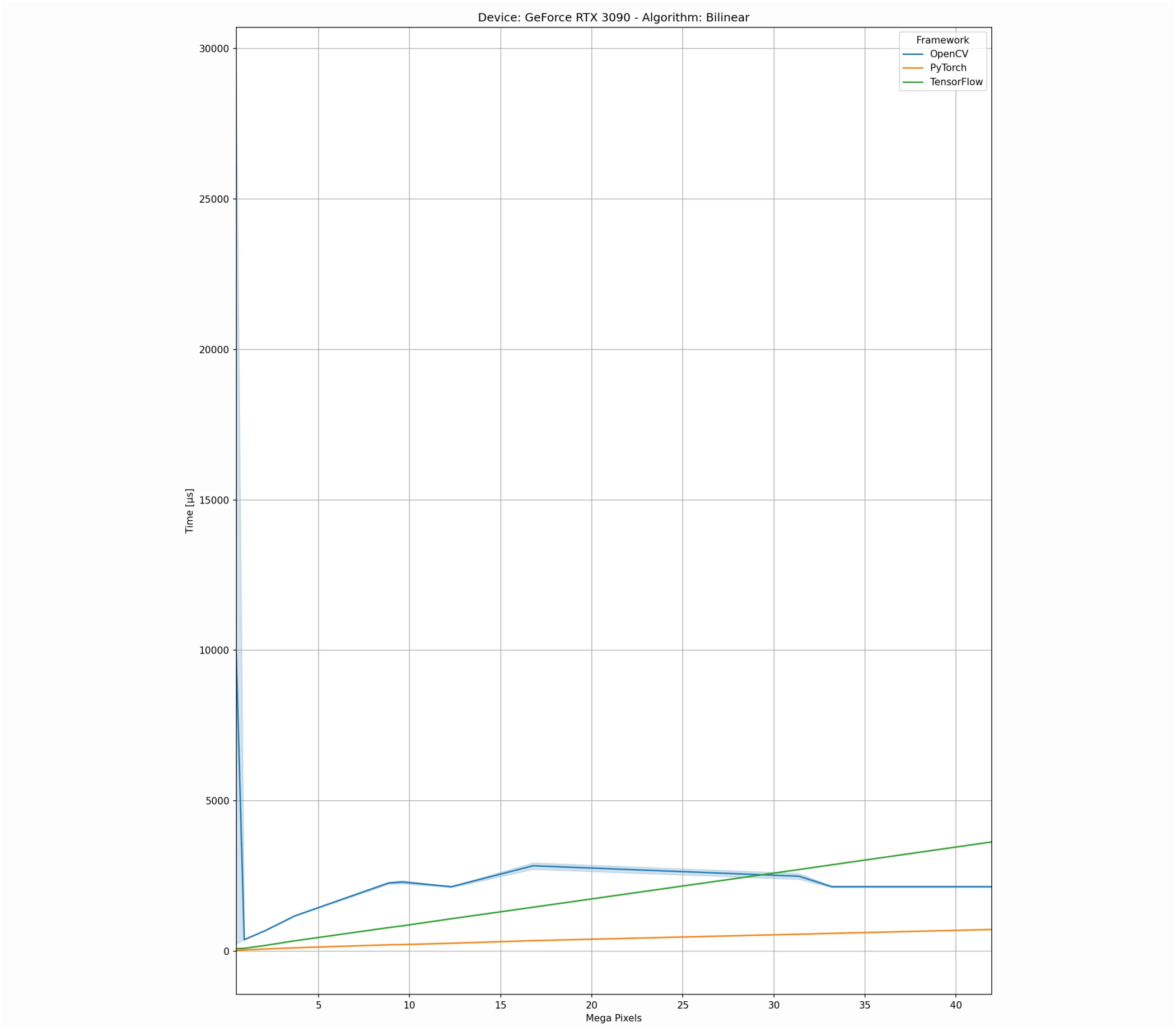
At first glance, we can observe some form of slow first iteration which indicates some JIT creates PTX code for the resize function when running for the first time on an image of a certain dimension:

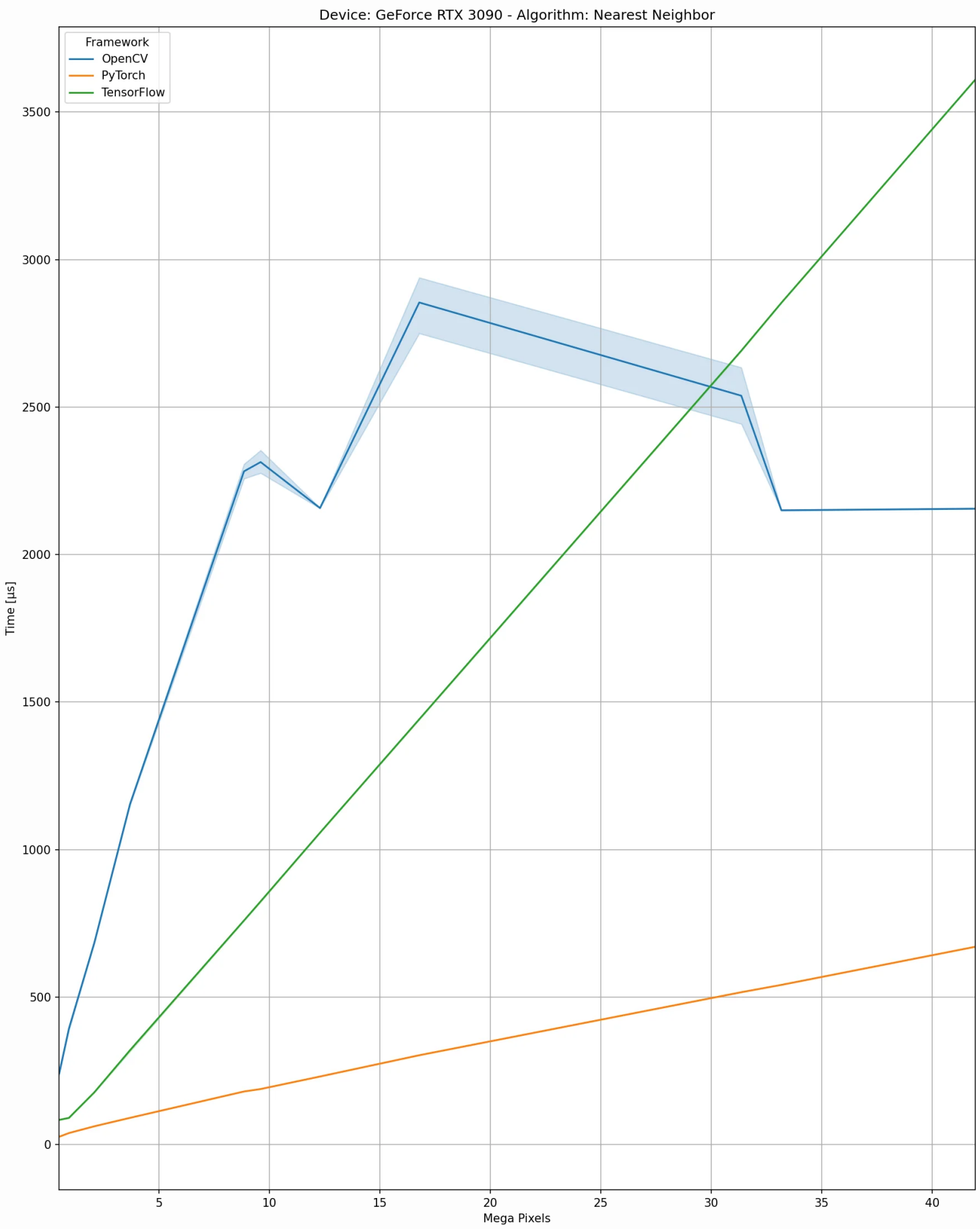

Device: GeForce RTX 3090 - Algorithm: Area
Framework
OpenCV
PyTorch
TensorFlow
Time [µs]
200000
150000
100000
50000
0
0.48
0.9216
2.0736
3.6864
8.84736
9.6
12.288
16.782656
31.363328
33.1776
41.94304
Mega Pixels
Device: GeForce RTX 3090 - Algorithm: Bilinear
1e6
1.0
0.8
0.6
0.4
0.2
0.0
Framework
OpenCV
PyTorch
TensorFlow
Time [µs]
0.48
0.9216
2.0736
3.6864
8.84736
9.6
12.288
16.782656
31.363328
33.1776
41.94304
Mega Pixels

## Results


Device: GeForce RTX 3090 - Algorithm: Area
Framework
OpenCV
PyTorch
TensorFlow
Time [µs]
200000
150000
100000
50000
0
5
10
15
20
25
30
35
40
Mega Pixels
Device: GeForce RTX 3090 - Algorithm: Bicubic
Framework
OpenCV
PyTorch
TensorFlow
Time [µs]
200000
175000
150000
125000
100000
75000

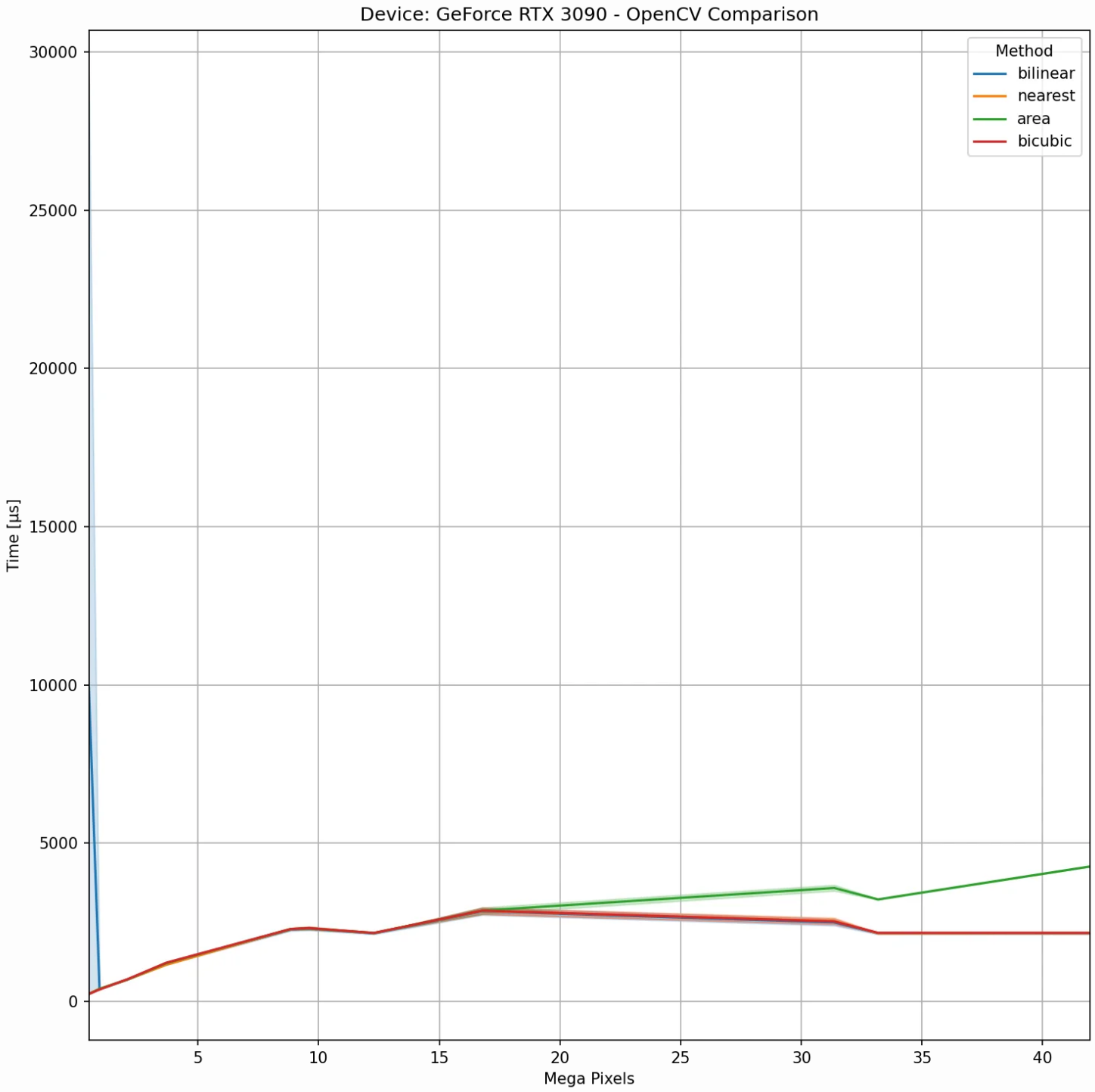

50000
25000
0
5
10
15
20
25
30
35
40
Mega Pixels

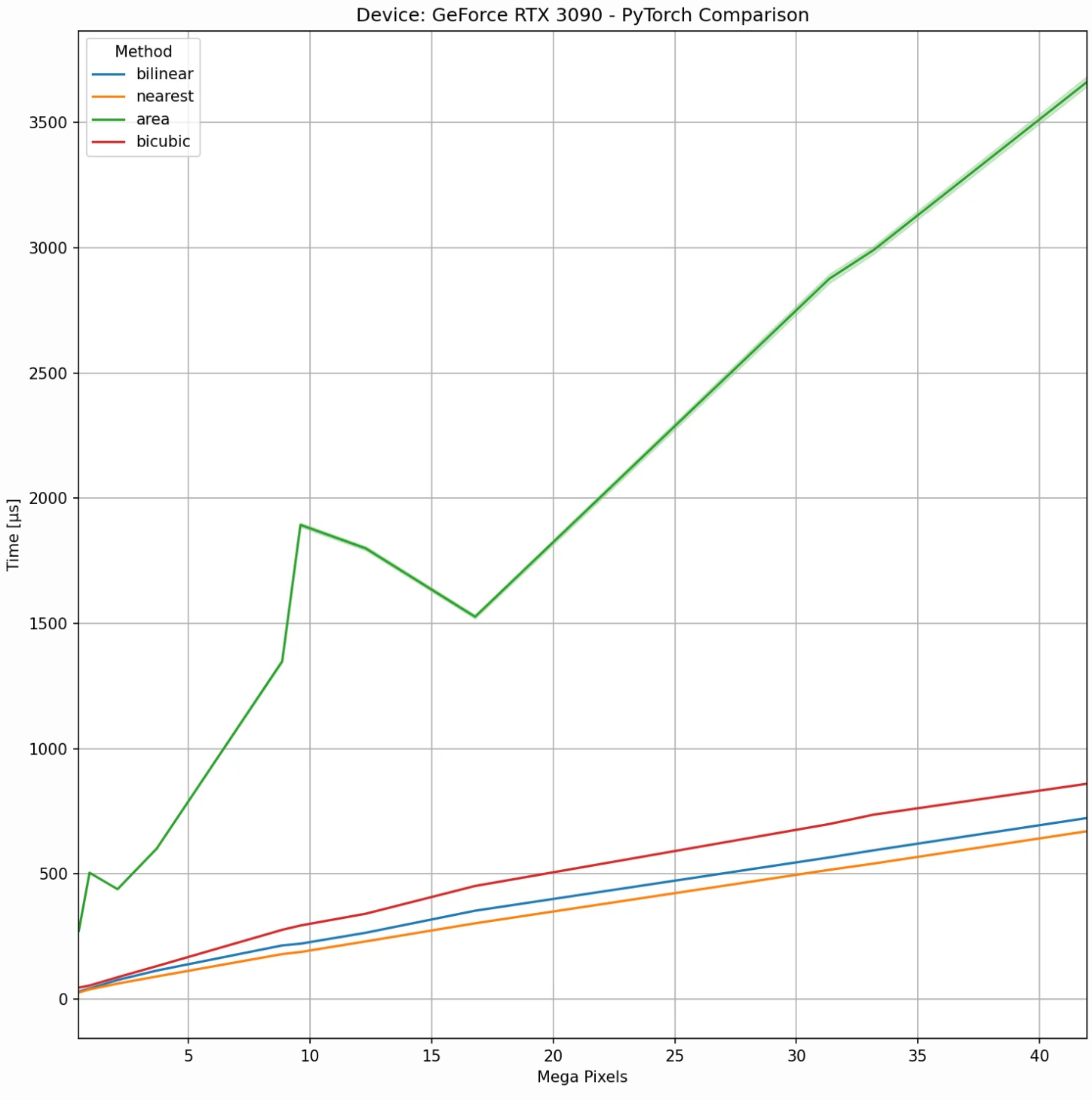

Device: GeForce RTX 3090 - Algorithm: Bilinear
Framework
OpenCV
PyTorch
TensorFlow
Time [µs]
30000
25000
20000
15000
10000
5000
0
5
10
15
20
25
30
35
40
Mega Pixels

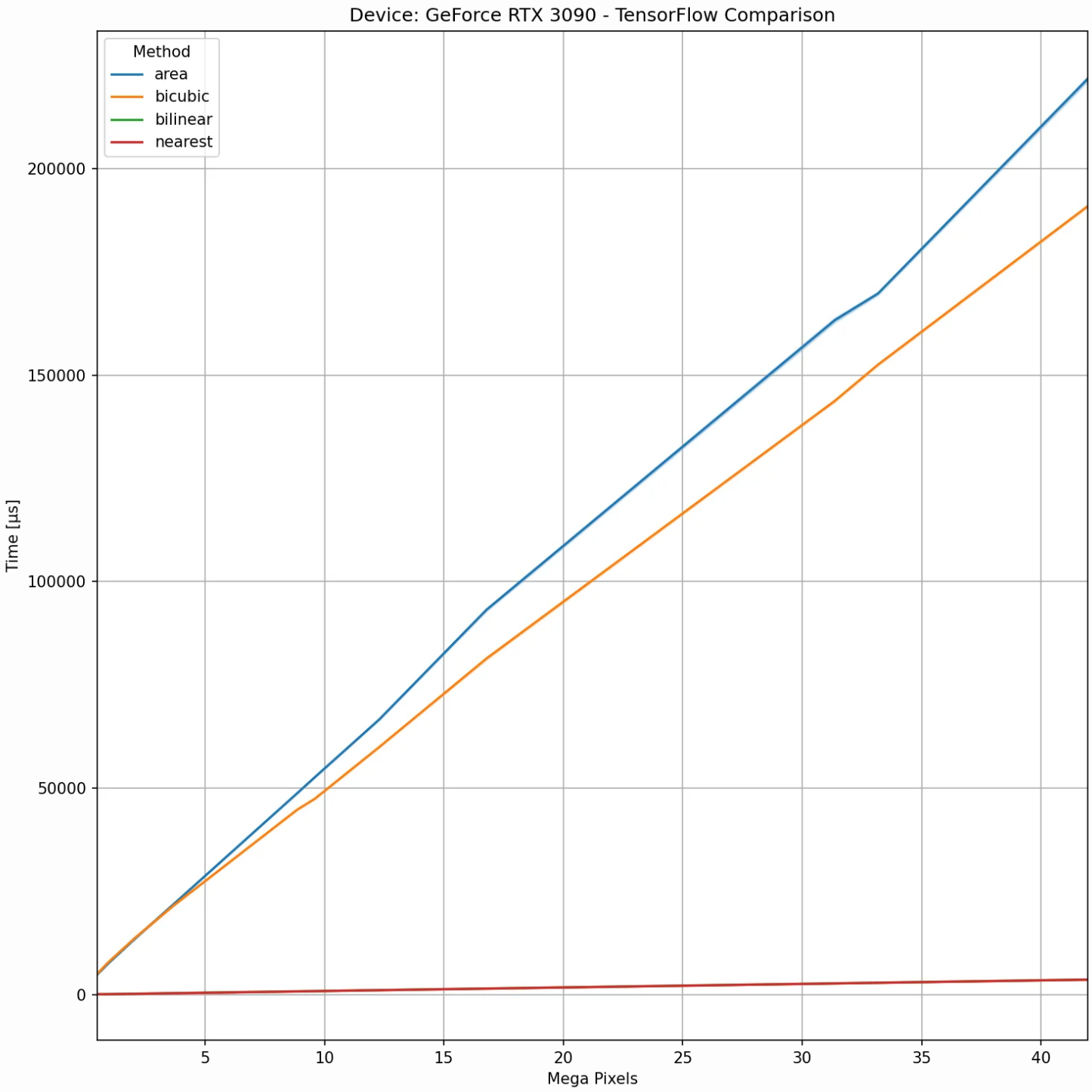



What can we observe? First and foremost that PyTorch seems to outperform both OpenCV as well as TensorFlow with respect to run time but more importantly consistency across different interpolation methods. Secondly, it should be obvious that we can ruin any inference pipeline easily when selecting a wrong library/toolkit.

Finally, we can have a look at a comparison of performance differences of different algorithms per framework:

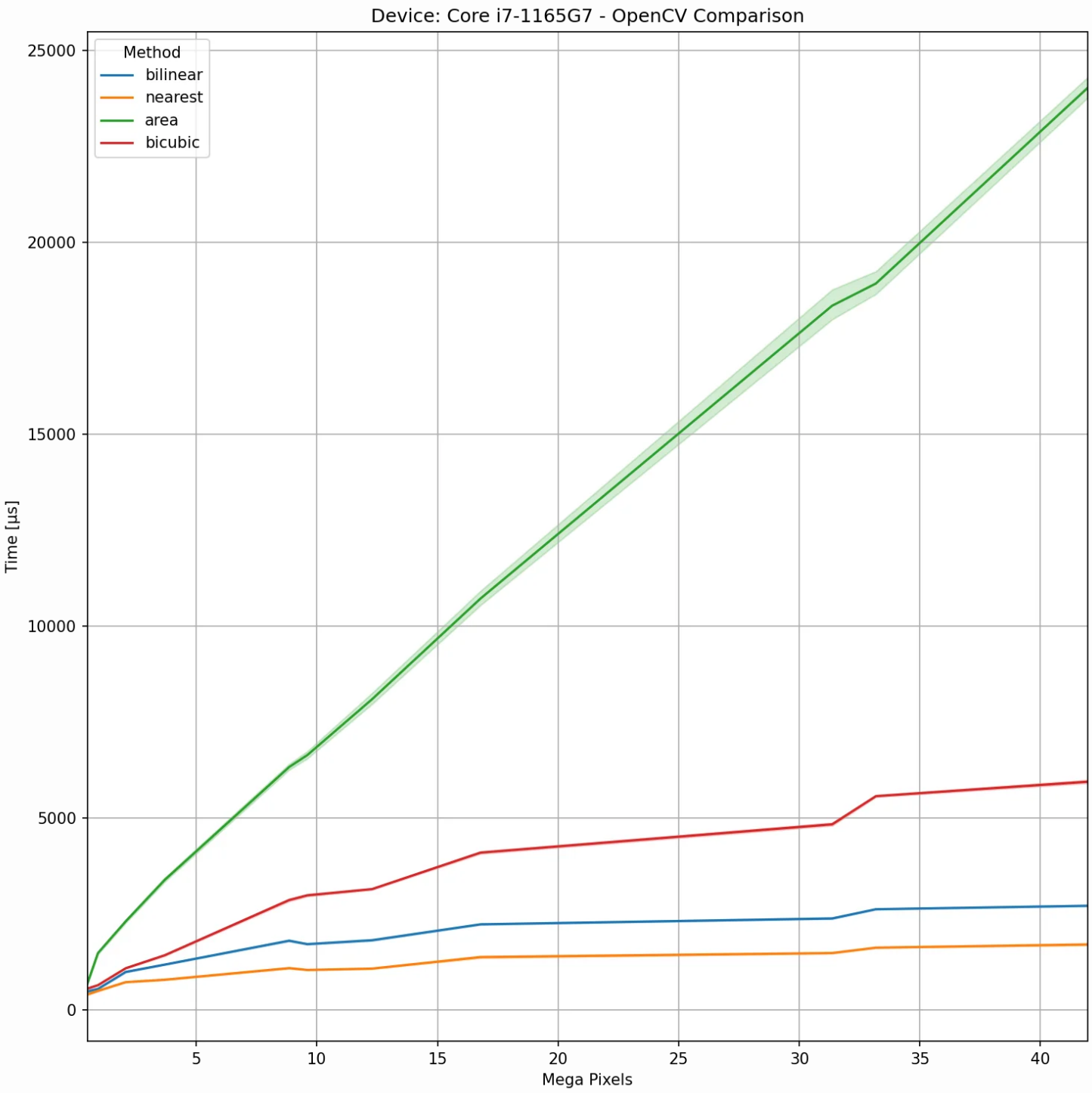

Device: GeForce RTX 3090 - OpenCV Comparison
Method
bilinear
nearest
area
bicubic
Time [µs]
30000
25000
20000
15000
10000
5000
0
5
10
15
20
25
30
35
40
Mega Pixels

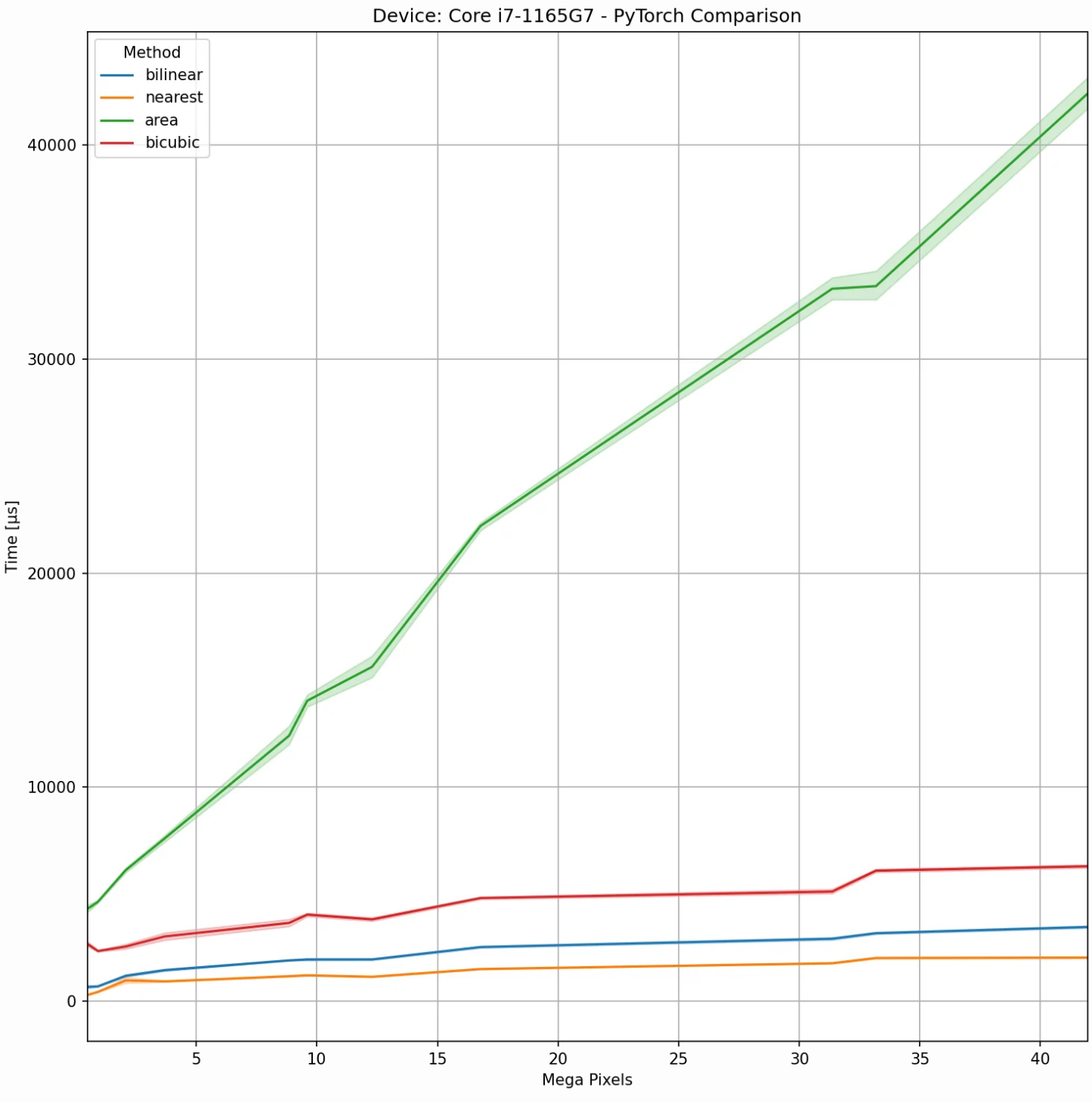

Device: GeForce RTX 3090 - PyTorch Comparison
Method
bilinear
nearest
area
bicubic
Time [µs]
3500
3000
2500
2000
1500
1000
500
0
5
10
15
20
25
30
35
40
Mega Pixels

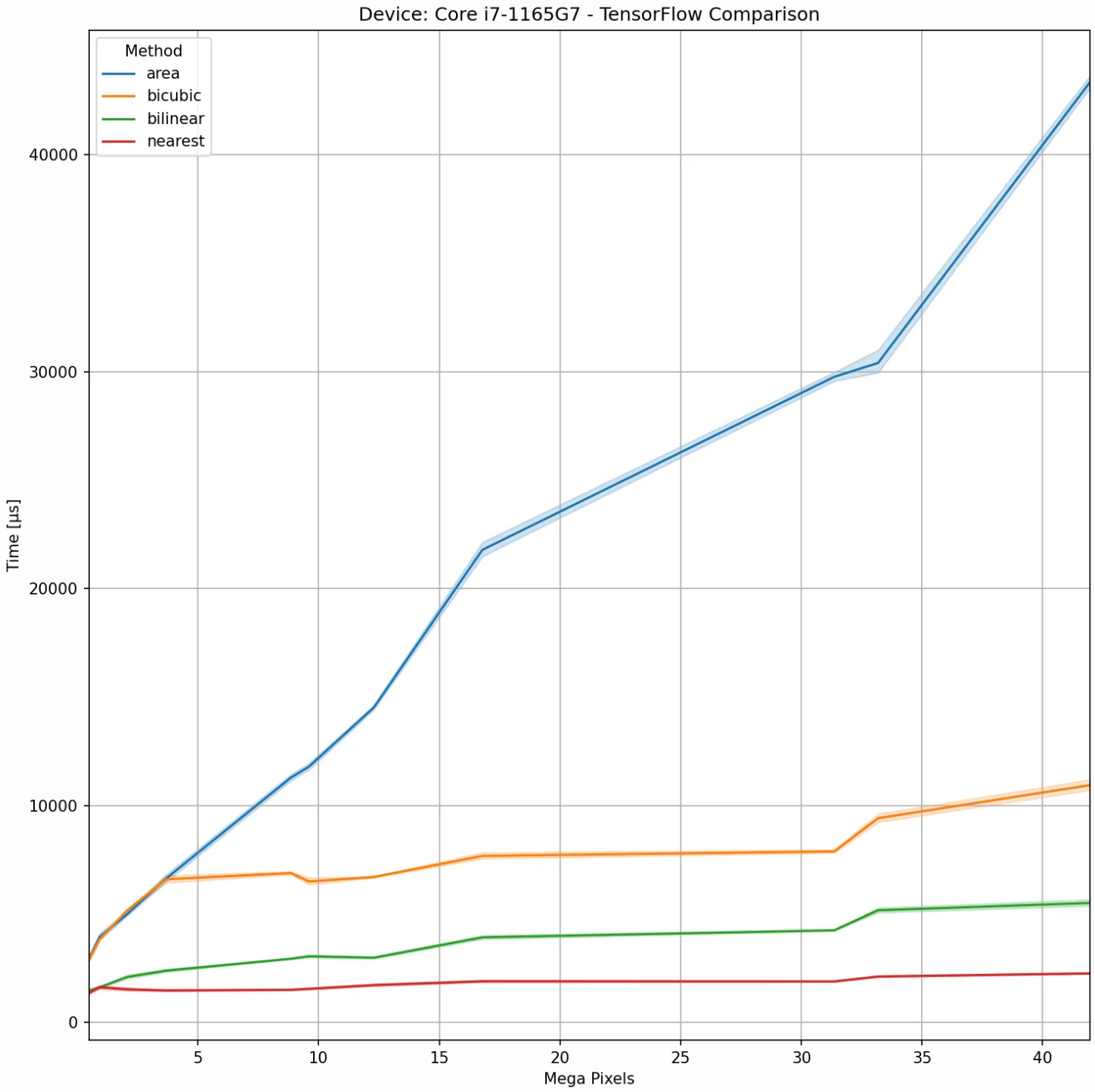



# CPU Results

For comparision we can have a look at results obtained using an Intel Core i7-1165G7. I did test a couple of Xeons but it seems like none of the implementations really benefit from a high core count as they seem to be single thread implementations (mostly?).


Device: Core i7-1165G7 - OpenCV Comparison
Method
bilinear
nearest
area
bicubic
Time [µs]
0
5000
10000
15000
20000
25000
5
10
15
20
25
30
35
40
Mega Pixels

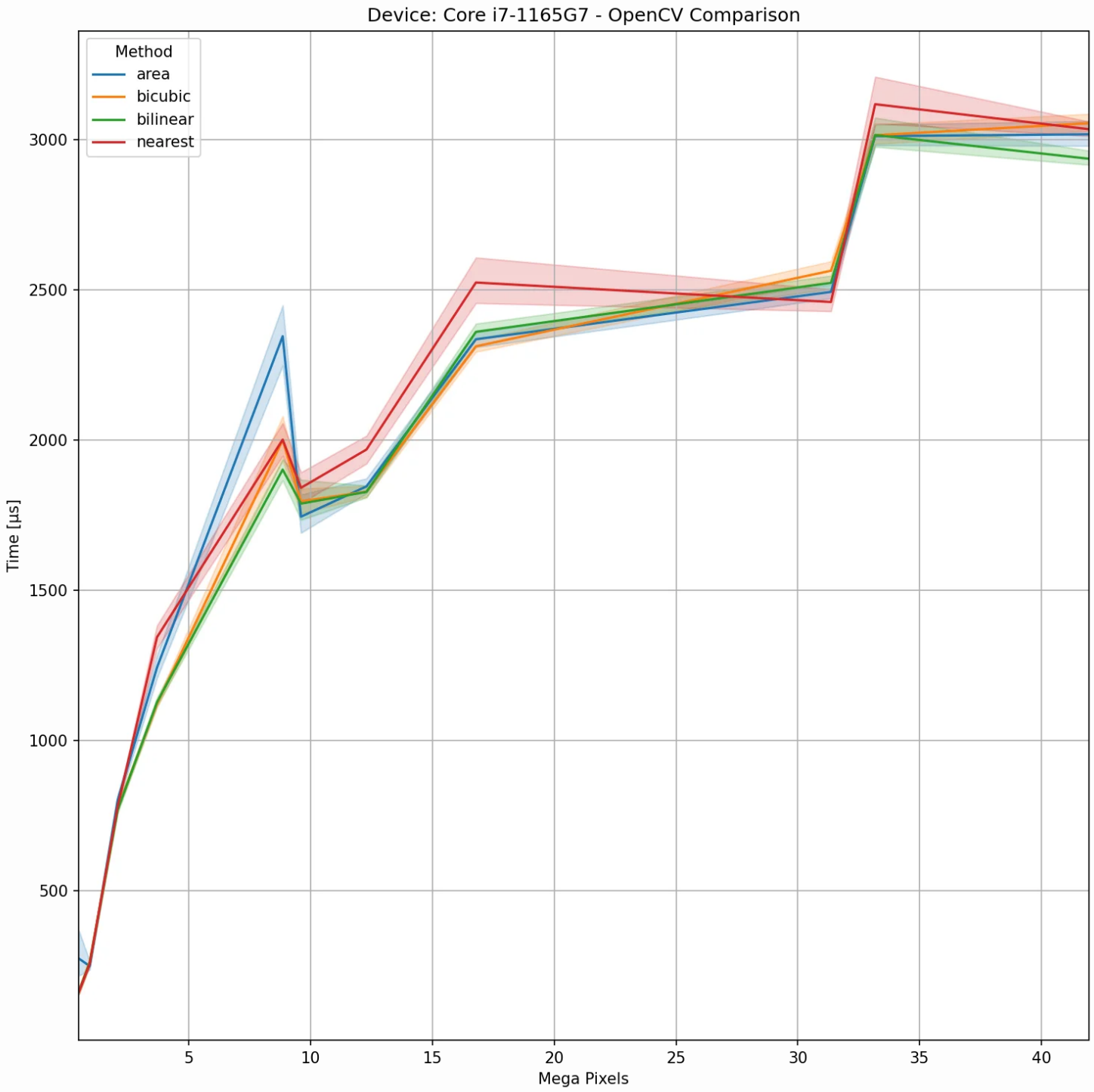

Device: Core i7-1165G7 - PyTorch Comparison
Method
bilinear
nearest
area
bicubic
Time [µs]
0
10000
20000
30000
40000
5
10
15
20
25
30
35
40
Mega Pixels

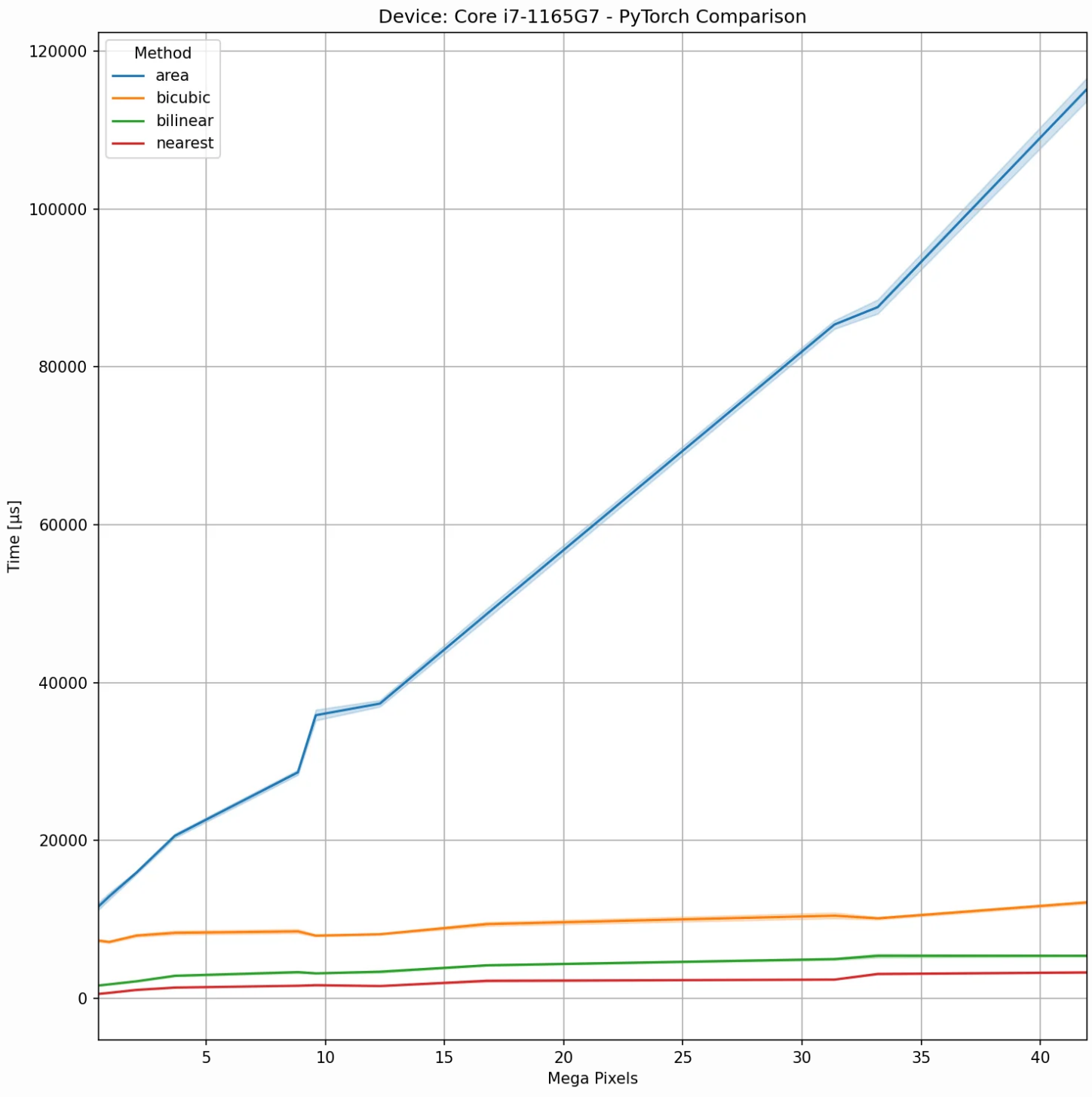



# Python vs. CPP

It seems like many people use Python to deploy deep learning pipelines. The results obtained and presented above are using the Python wrappers provided by these frameworks. As they are basically written in C++ it is worth to check if some aspects of Python and its memory management could be a reason for some seemingly weird and slow behavior. **NB!:** I used the standard libtorch C++ API and did not spent time on handcrafting anything using `ATen`. `g++` 11.2.0 was used to compile it with optimization setting `-O0`.

Let's start with CPU results.


Device: Core i7-1165G7 - OpenCV Comparison
Method
area
bicubic
bilinear
nearest
Time [µs]
3000
2500
2000
1500
1000
500
5
10
15
20
25
30
35
40
Mega Pixels

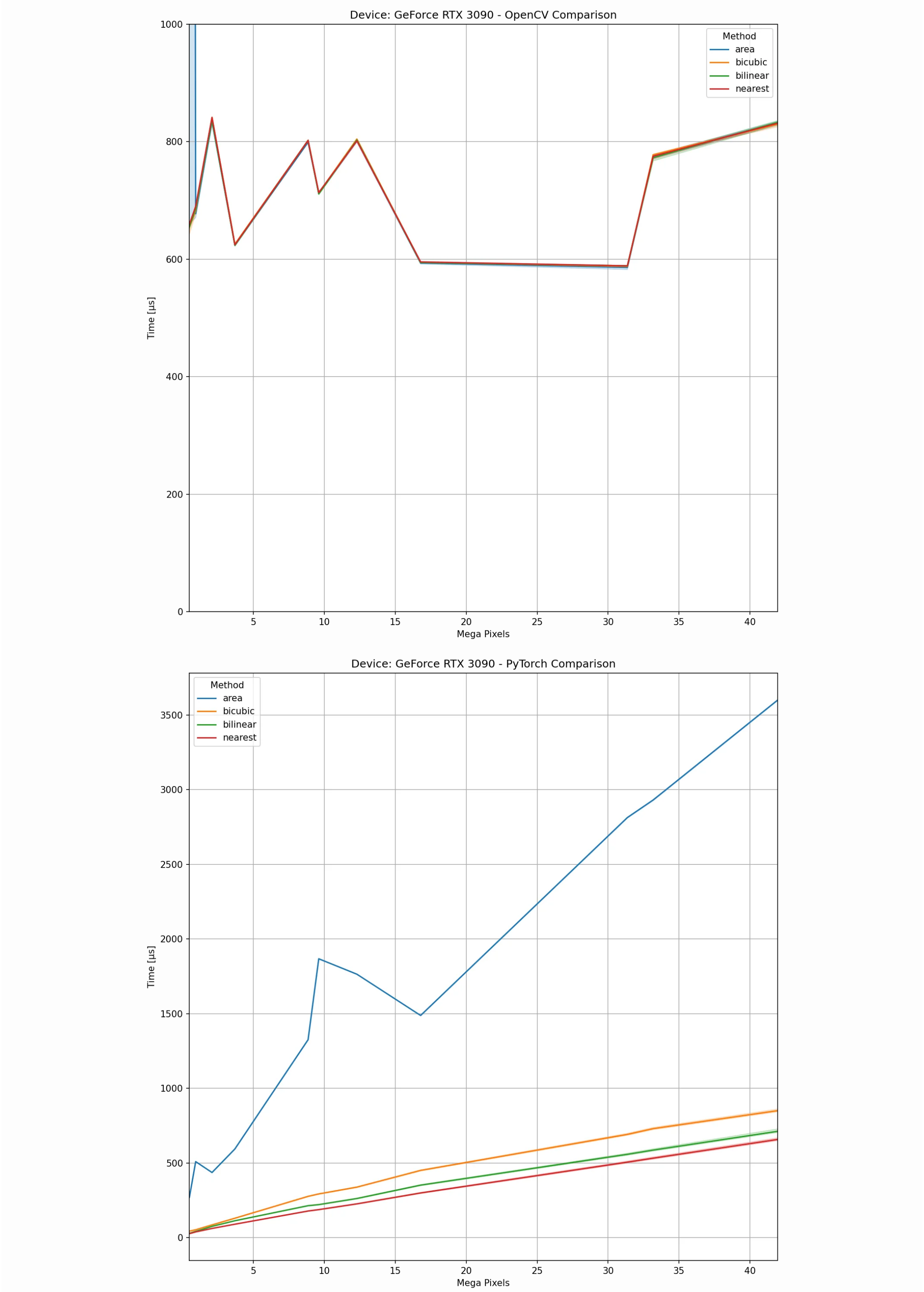

Device: Core i7-1165G7 - PyTorch Comparison
Method
area
bicubic
bilinear
nearest
Time [µs]
120000
100000
80000
60000
40000
20000
0
5
10
15
20
25
30
35
40
Mega Pixels

The OpenCV results are much closer together now and significantly faster. This clearly indicates some weird Python overhead though that does not explain the non-linearity inscaling nor does it explain away the standard deviation we can observe. Compared to libtorch it is much faster. Some causes for massive performance differentces betwee OpenCV and libtorch on this CPU might be caused by the fact that I compile OpenCV myself and with this CPU it made sense to compile it against Intel MKL and with AVX 512 instruction set extensions enabled though I'm not sure if these are actually used for resizing images.

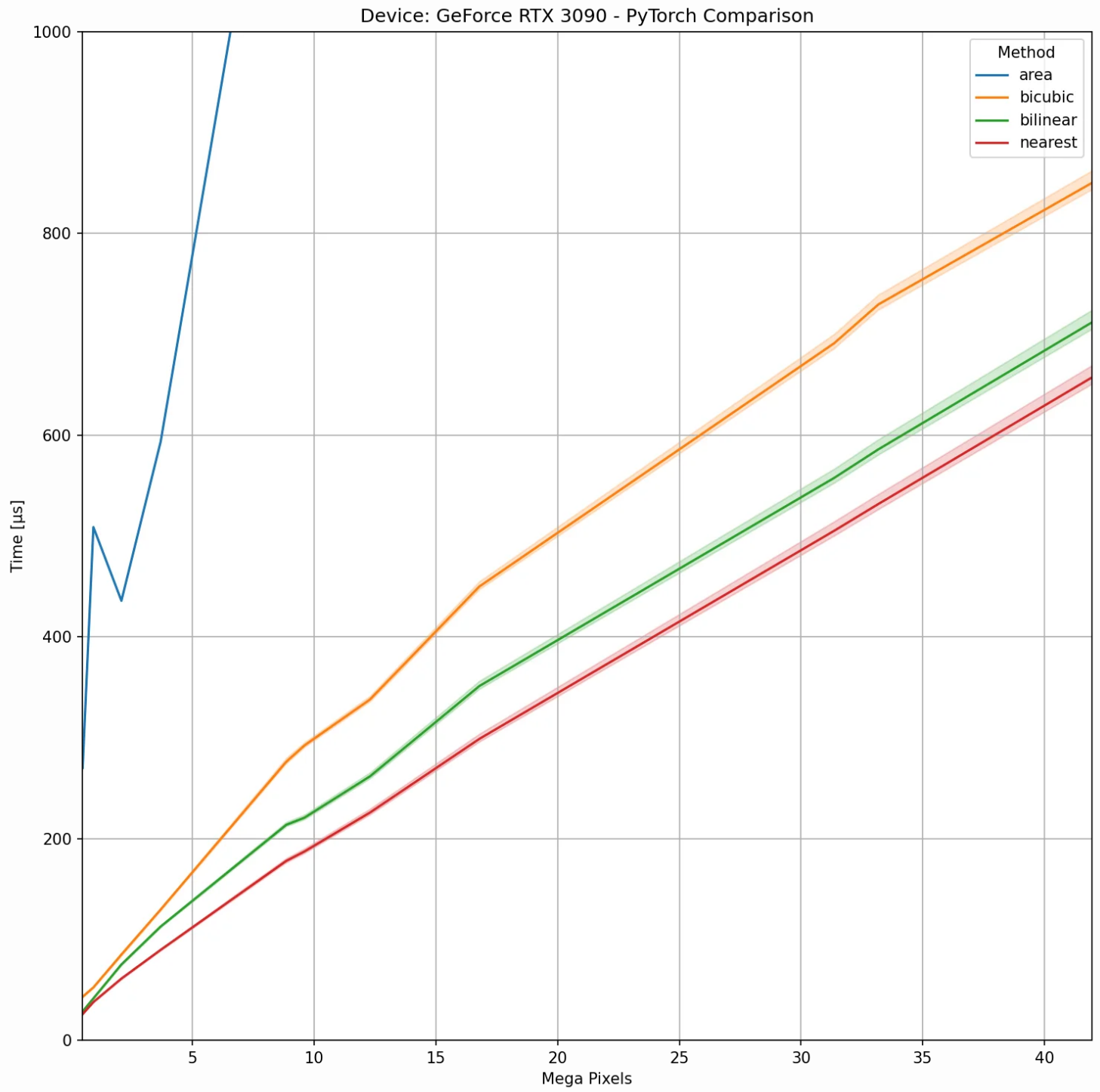
The source looks similar to python. Please not that libtorch requires the usage of `torch::nn::functional::InterpolateFuncOptions` instead of using function arguments.

```
...

cv::cuda::resize(frame_cu, frame_out, cv::Size(608,608), cv::INTER_NEAREST);

...

frame_out = F::interpolate(frame, F::InterpolateFuncOptions()
                                                    .size(std::vector<int64_t>
({608,608}))
                                                    .mode(torch::kNearest));
```

Let's look on the GPU side.

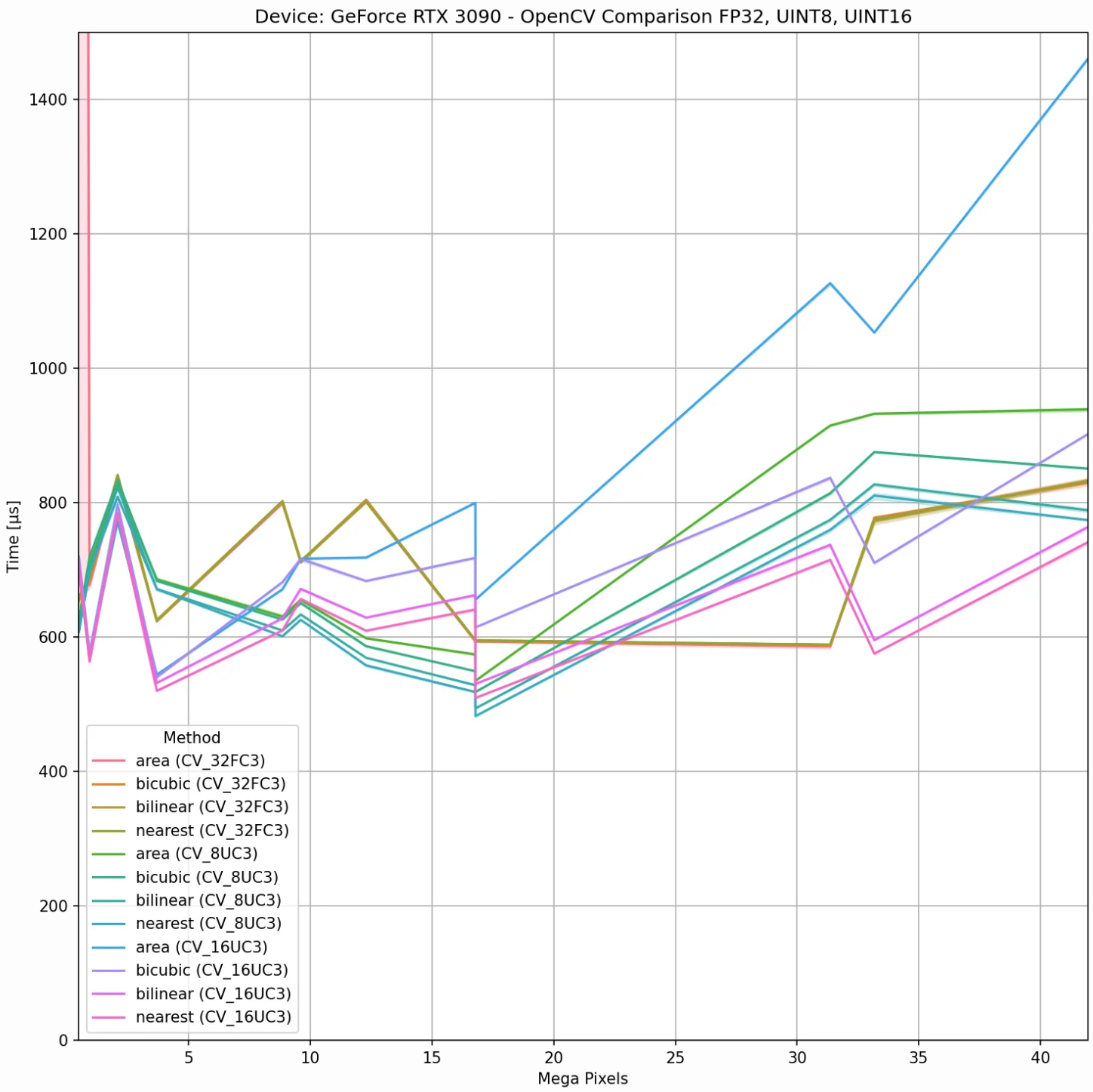

Device: GeForce RTX 3090 - OpenCV Comparison
Method
area
bicubic
bilinear
nearest
Time [µs]
Mega Pixels
Device: GeForce RTX 3090 - PyTorch Comparison
Method
area
bicubic
bilinear
nearest
Time [µs]
Mega Pixels

Some of this weird behavior can't be explained switching from Python to C++. In both cases CUDA streams were synched properly. The differences between algorithms are much smaller with OpenCV again but at least 'area' shows another weird spike with small resolutions as did with Python as well but in general it seems to be a lot faster. Regarding libtorch and performance differences to PyTorch we can observe that 'area' is a bit faster but the other three algorithms are similarly fast. When looking at the other three algorithms, then we can observe almost linear scaling with image sizes:

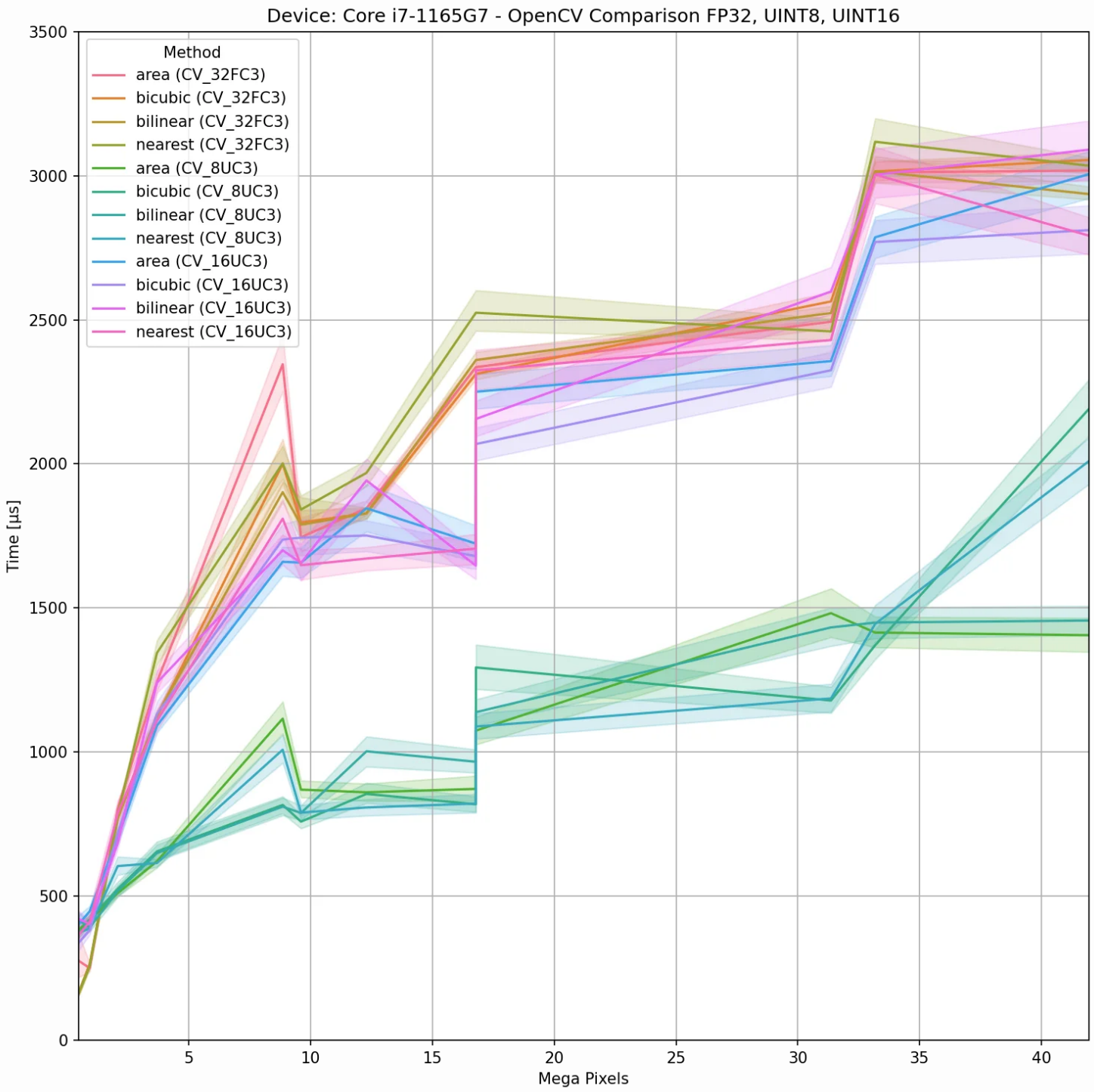



## UINT vs. FP32

PyTorch and TensorFlow are limited to floating point arrays. However, OpenCV can process signed and unsigned integer arrays as well. The two most common input formats are `uint8_t` and `uint_16` (more common in industrial and medical applications). Let's see how these data types perform (C++ implementation used).




The GPU results are a mess. `CV_32FC3` seems to remain most consistent whereas other data types vary quite a bit. From my understanding of NVIDIA's GPU architechture it could be caused by type casting. As I understand it the "normal CUDA cores/stream processors" can handle `fp32` only and everything else is casted. Other data types such as INT4,INT8, FP16 and TF32 are mainly primitives of the tensor cores on modern NVIDIA GPUs. It could explain this inconsistency but I might be wrong here and at least when dealing with less than 15 Mega pixels, what most people are probably doing these days, doing rescaling with `CV_8UC3` arrays seems to be beneficial.



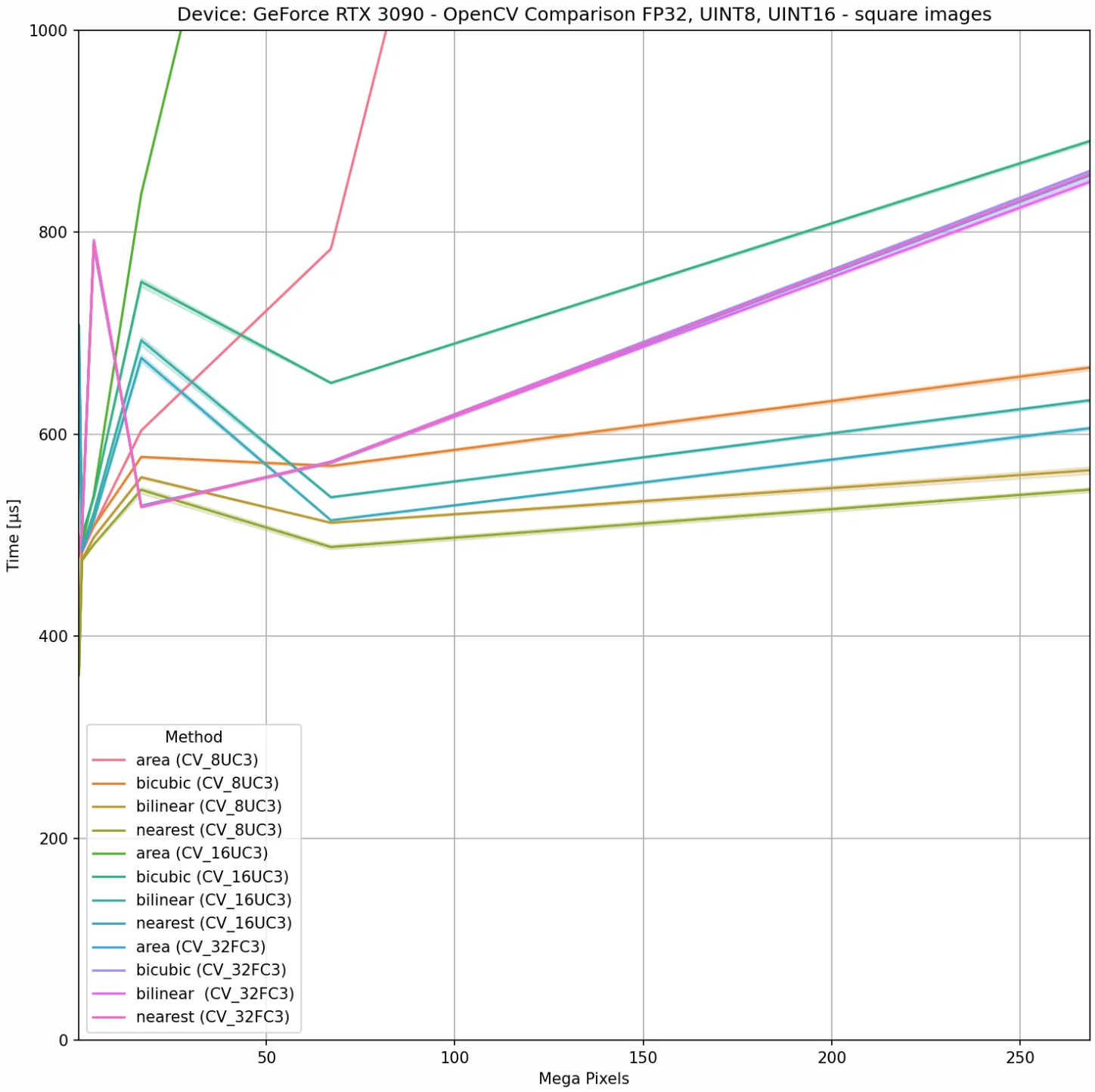

The differences between `CV_32FC3` and `CV_16UC3` seem to be rather small with slight performance advantages for the latter. However, `CV_8UC3` seems to be much faster.

## Square Images

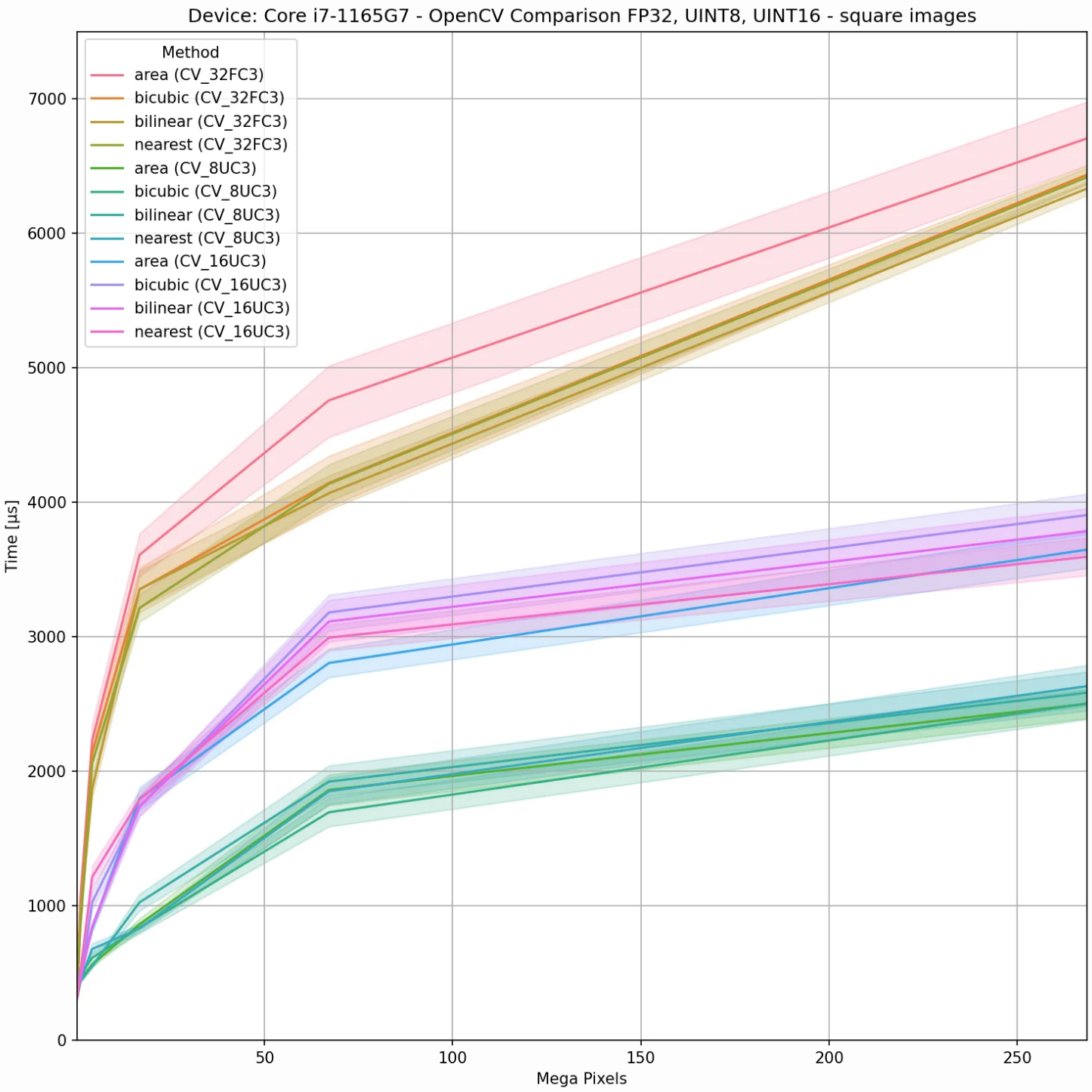
The image resolutions used above originate from common video resolutions (consumer and industrial cameras). However, it is known that e.g. certain CUDA operations are optimized for tensors with dimensions multiple of 8 or multiple of 32. Since the CPU uses is able to utilize AVX2 and AVX 512 instruction set extensions it makes sense if there is a performance increase to use square inputs as well. In order to make things as easy as possibles, the following image dimensions were used:

```
std::vector<cv::Size> resolutions_cv;
resolutions_cv.push_back(cv::Size(512,512));
resolutions_cv.push_back(cv::Size(1024,1024));
resolutions_cv.push_back(cv::Size(2048,2048));
resolutions_cv.push_back(cv::Size(4096,4096));
resolutions_cv.push_back(cv::Size(8192,8192));
resolutions_cv.push_back(cv::Size(16384,16384));
```

**NB!:** When comparing these figures with the graphs shown above, please remember that the x-axis changed quite a bit.




With respect to the GPU results we may conclude that there clear performance improvements using unsinged integer types when dealing with more than 50 Mega pixels. Using less than 50 Mega pixels results vary a bit for unknown reasons but there are some trends visible.




The CPU results are very clear. `CV_32FC3` is much slower than `CV_16UC3` which is slower than `CV_8UC3` especially at when dealing with large images.

Hence, depending on pipeline requirements etc. might be beneficial to pad an image first to make it square and apply resize operations later. Many neural networks require square inputs anyhow and usually padding is preferred over cropping depending on the application in question.

# Conclusions

Rescaling images on the GPU is often required and as long as we are using FP32 images we observe clear advantages using GPUs. In a different/earlier stage in a pipeline resizing `uint8_t` or `uint16_t` on a CPU or with custom CUDA kernels that are able to handle non-FP32 might be beneficial. Using C++ shows some advantages with respect to performance however it does not get rid of rather unexplainable performance differences of various algorithms with respect to various image resolutions.