UNIVERSITÀ DI PISA

Master's Degree in Computer Engineering
Computer Architecture

A parallel approach to edge detection

Students:
Taulant Arapi
Antonio Ciociola
Francesco Scarrone

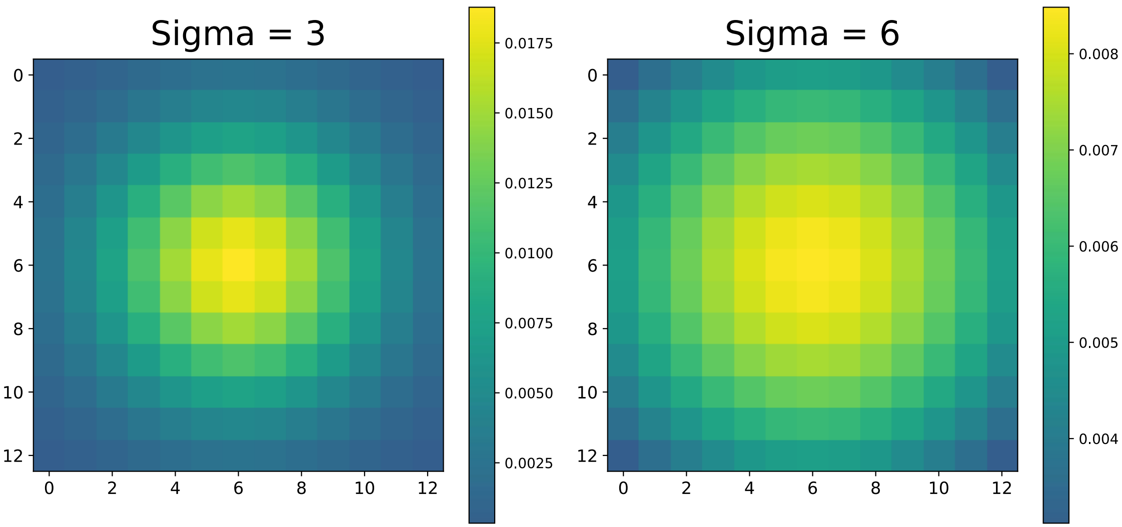
Academic Year 2024/2025

Edge detection

Initial Image → First Blur → Second Blur → Subtract



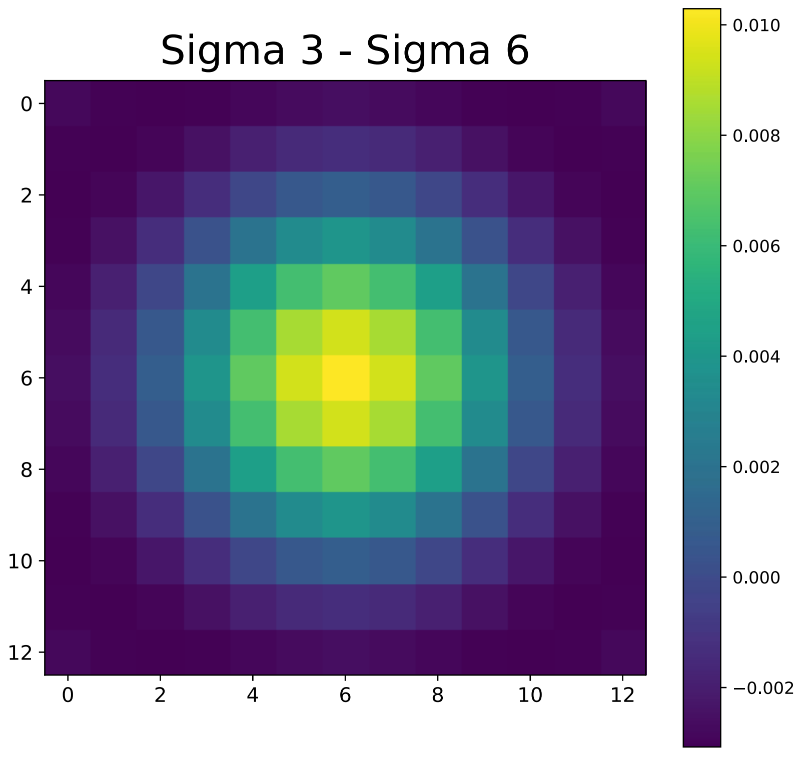

Gaussian Blur



Larger σ results in stronger blurring

Optimal filter dimension is **k×k, k=2×σ+1**

Difference of Gaussians



The difference operation results in a pass-band filter

Edge Detection
CPU

System specifications

Hardware (CPU)	
Name	**Intel Core i7-8750H**
Power	**45 W**
Launch	2018
Architecture	Coffee Lake
Process Size	14 nm
Clock Frequency	2.2 - 4.1 GHz
Cores	**6**
Threads	**12**
L1I cache	32 kB per core
L1D cache	32 kB per core
L2 cache	256 kB per core
L3 cache	9 MB shared

Software	
OS	Ubuntu 25.04
Kernel	Linux 6.14.0
Compiler	g++ 14.2.0
Profiler	**Intel VTune 2025.3**

Testing methodology

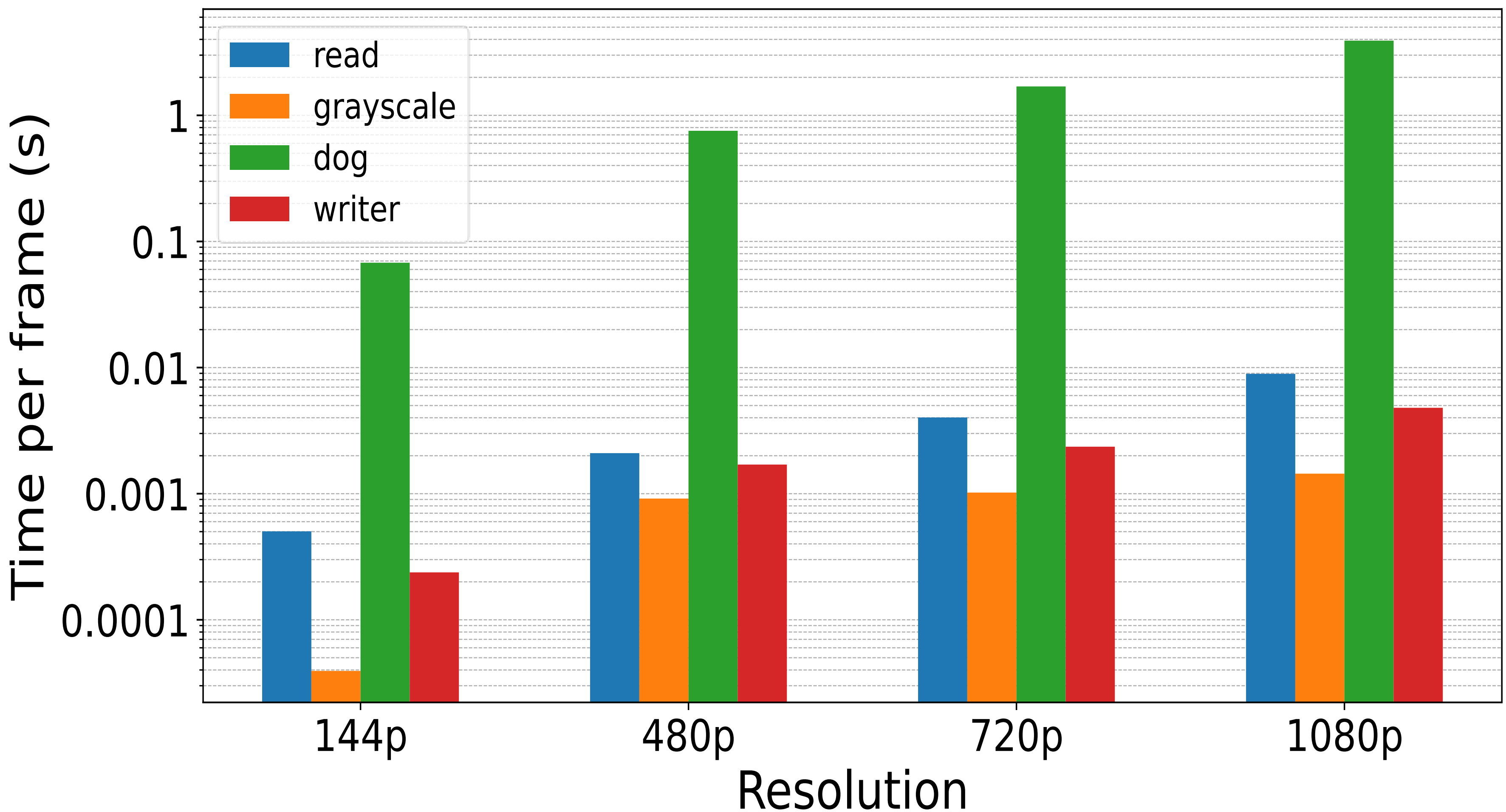
Parameters of the experiments:

- Resolution: **144p**, **480p**, **720p**, **1080p**
- Number of frames: **60**
- Gaussian blur kernels: $\boldsymbol{\sigma_1 = 3, \sigma_2 = 6}$
- Kernel size: **13x13**

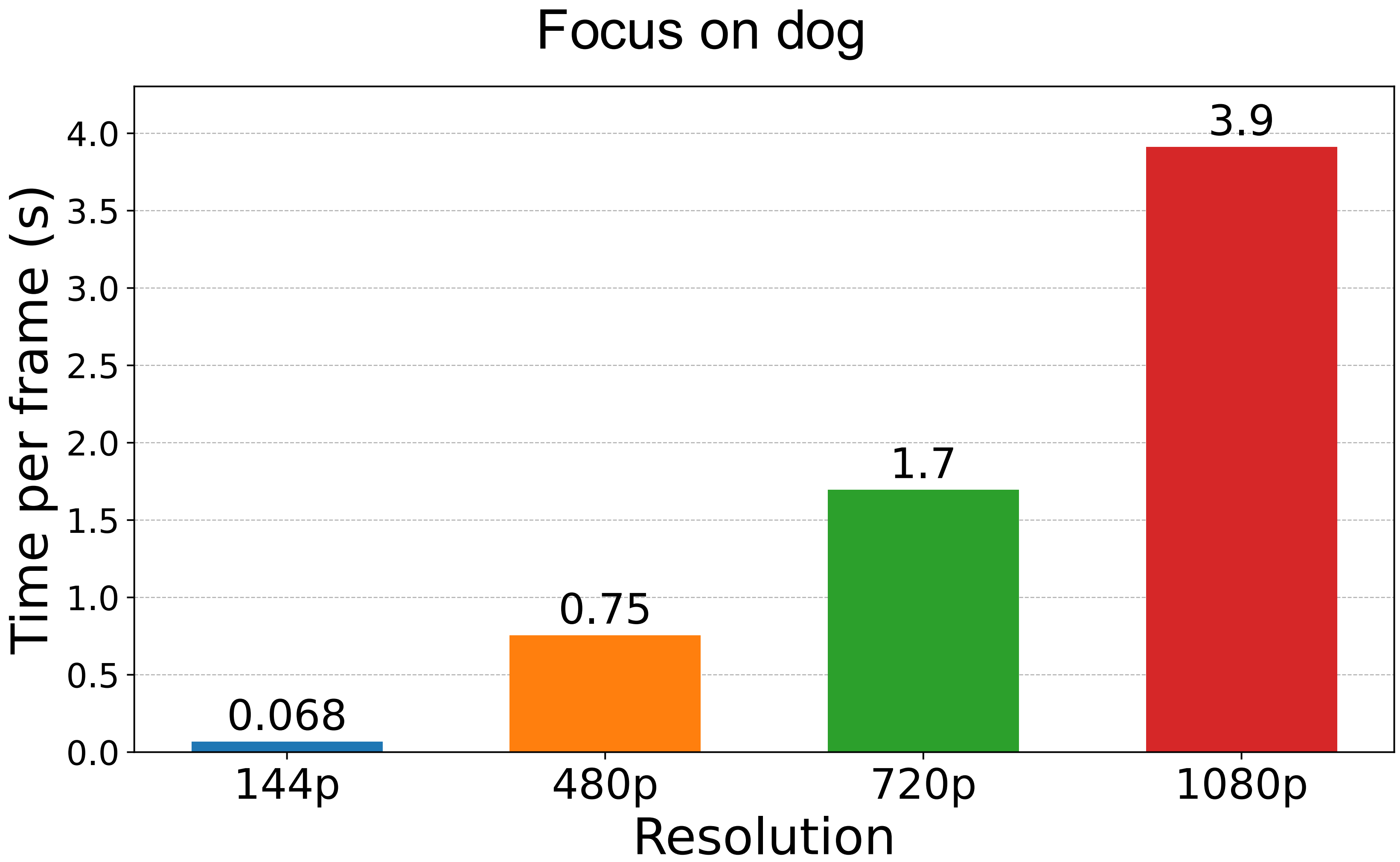
For each experiment, the mean of 10 independent runs is taken.

Goal: 30 FPS @ 1080p

Version 0 “Naïve”



Version 0 “Naïve”

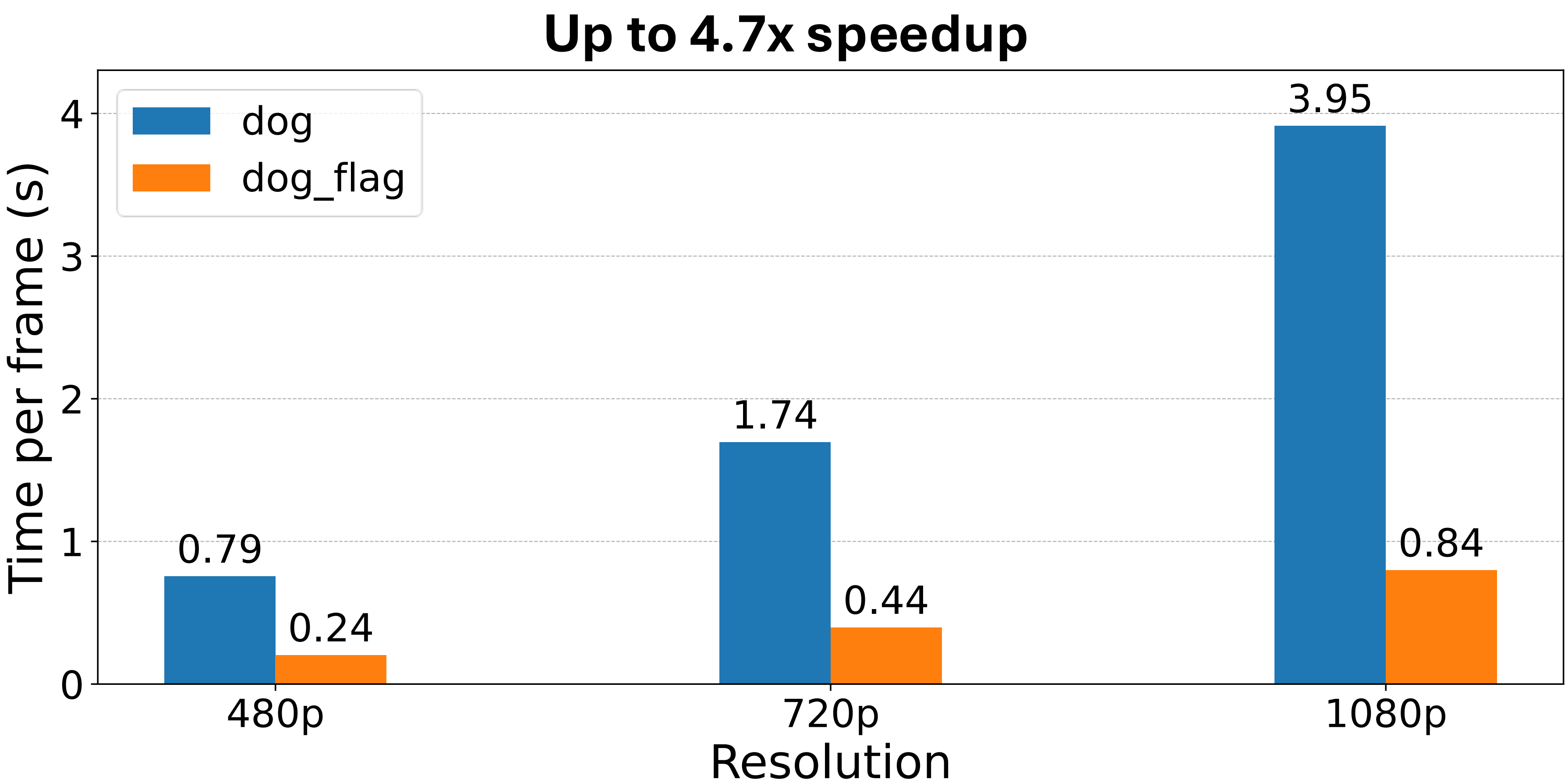


Compiler flags

Compilation flags matter! We used:

- **-O3**: Turn on most optimizations
- **-flto**: Optimize at link time, optimize all code together
- **-ffast-math**: Aggressive floating-point optimizations

Version 0 with flags



Where to improve?

The time complexity is
$O(H \cdot W \cdot k^2)$

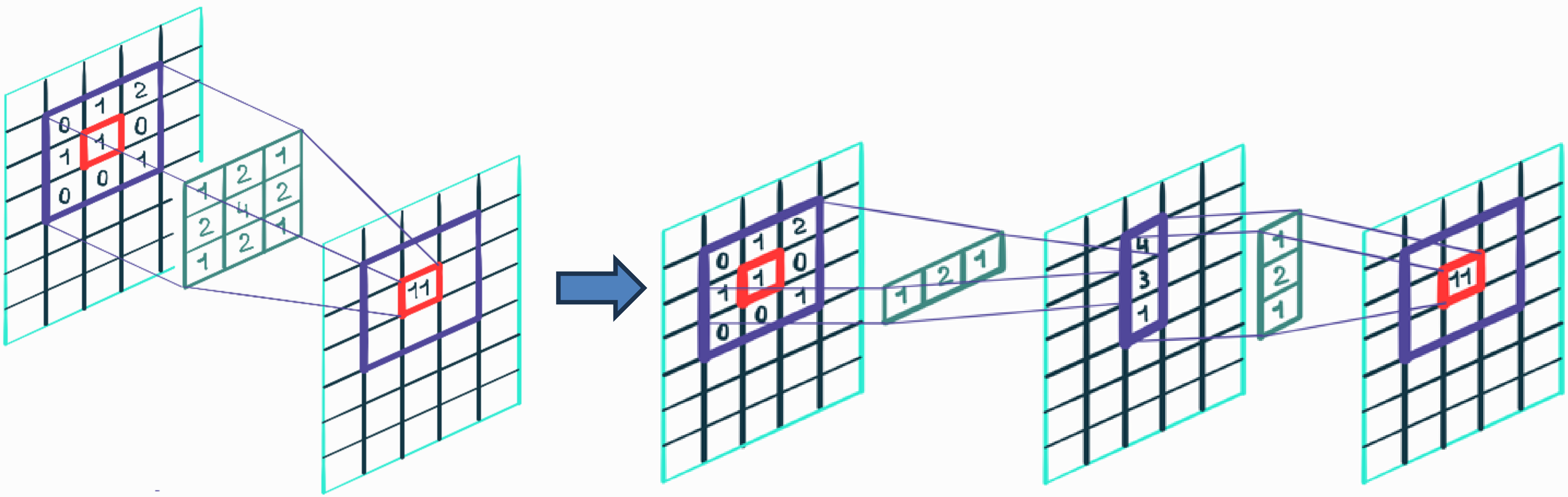
We can't reduce the filter size further
It's required for quality and accuracy

So... where can we optimize?

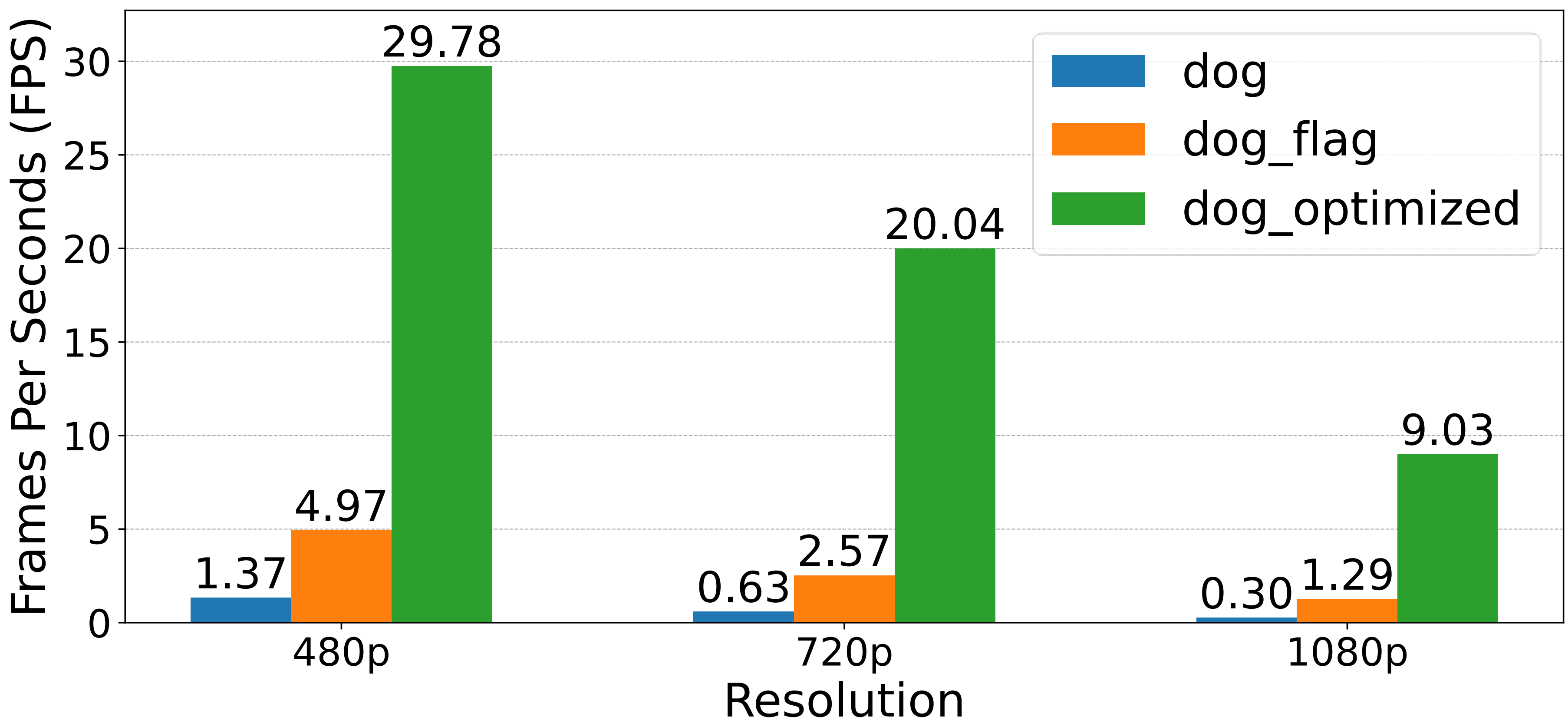
Solution: Separate the filter

The Gaussian blur is a separable filter:
2D blur → two 1D blurs

Reduces the number of operations from
$O(H \cdot W \cdot k^2)$ to **$O(H \cdot W \cdot (k+k))$**



Compare after optimization



After optimization is **7 times** faster @1080p

Multithread

The task is **easy to parallelize:**
we have many independent pixels, no synchronization needed
⇒ **assign some rows** of the image **to each thread**

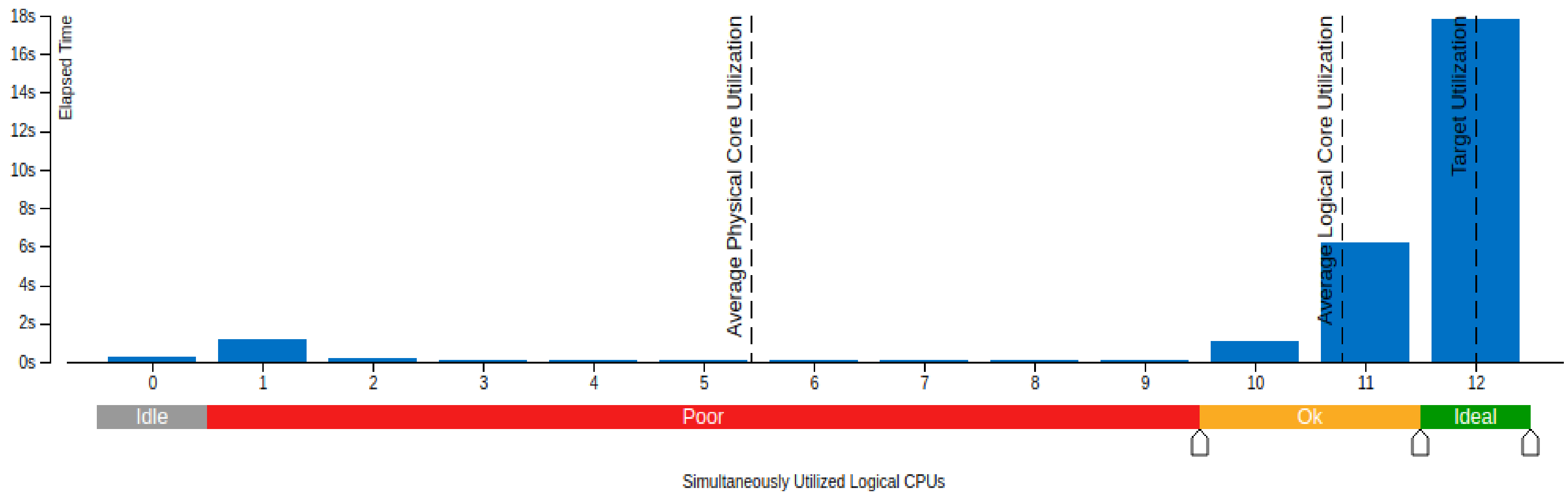
Multithread

Effective Physical Core Utilization: 90.5% (5.432 out of 6)

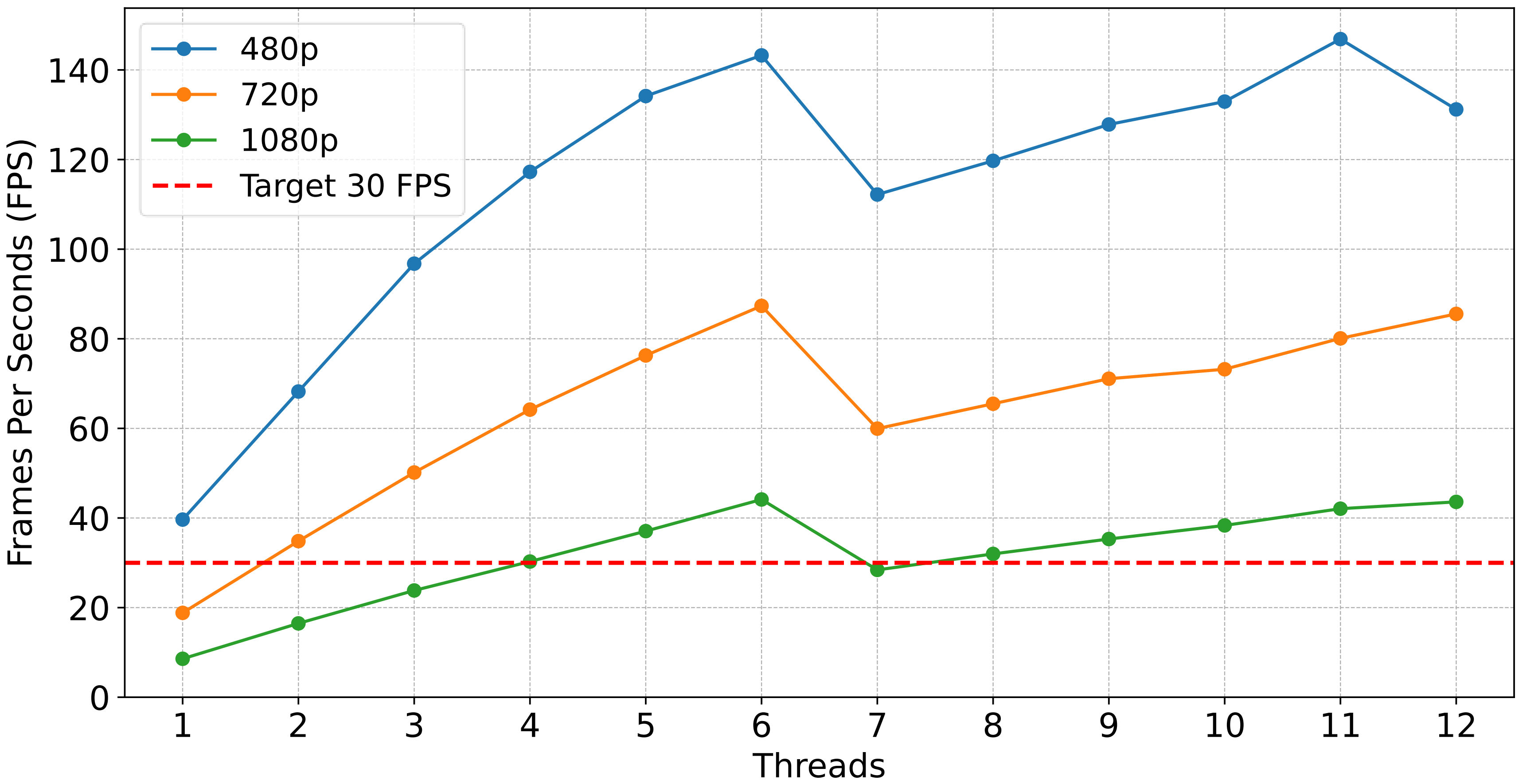
Effective Logical Core Utilization: 89.9% (10.793 out of 12)

Effective CPU Utilization Histogram

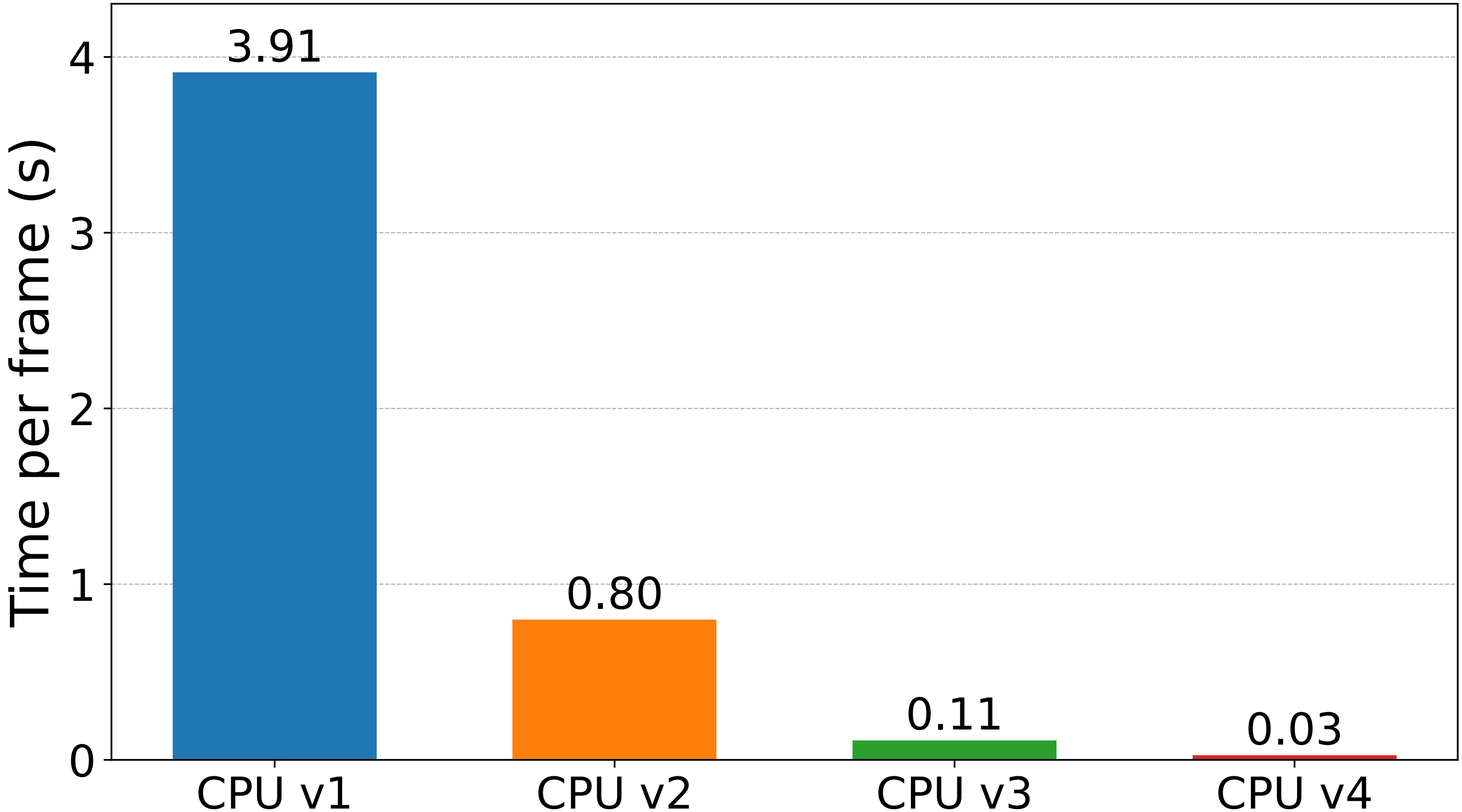
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



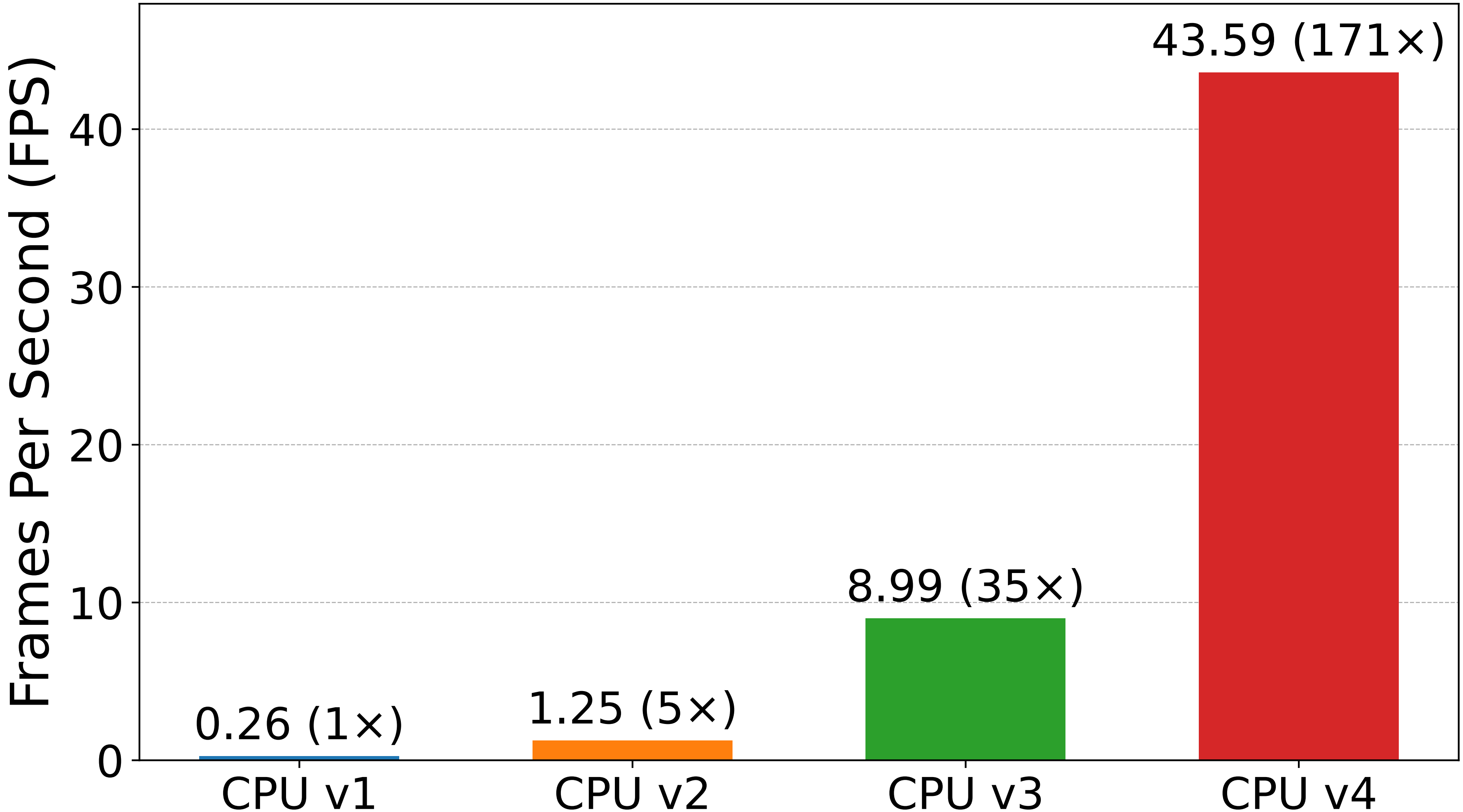
Multithread



Final results



Final results



Edge Detection

GPU

System specifications

Hardware (GPU)	
Name	**RTX 2060 Mobile**
Power	**90 W**
Launch	2019
Architecture	Turing
Process Size	12 nm
VRAM	6 GB GDDR6
CUDA cores	**1920**
Streaming Multiprocessors	**30**
Warp size	32
L1 cache	64 kB per SM
L2 cache	3 MB

Software	
OS	Ubuntu 25.04
Kernel	Linux 6.14.0
CUDA toolkit	12.9
Profiler	**Nsight Compute 2025.2**

Testing methodology

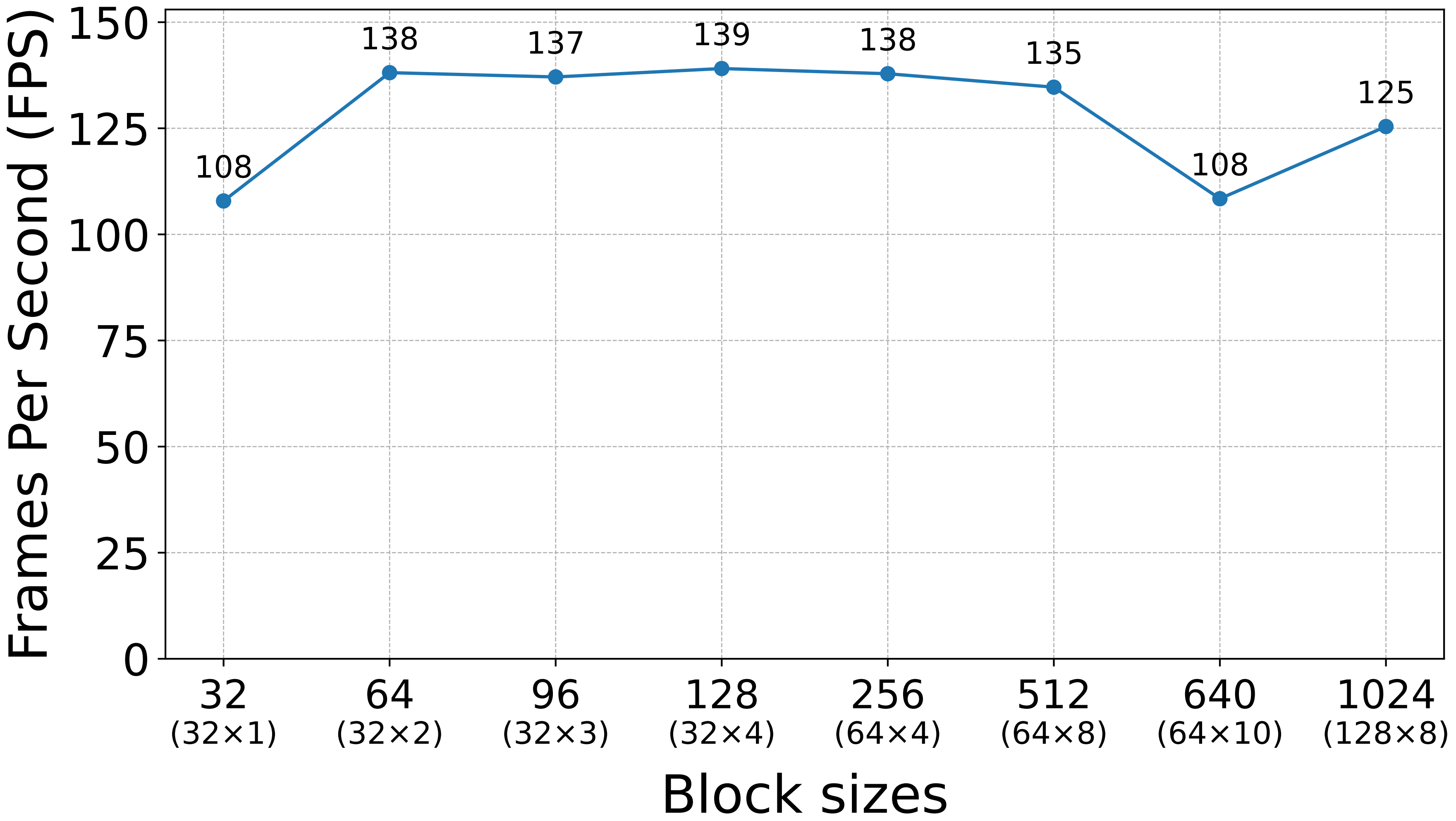
The experiments are performed on a **4K video** from a dashcam (and 1080p for comparison with CPU benchmark).

For each experiment, the mean of 10 independent runs is taken.

Timer starts after the video is loaded in memory

Goal: 180 FPS @ 4K

Version 1 performance

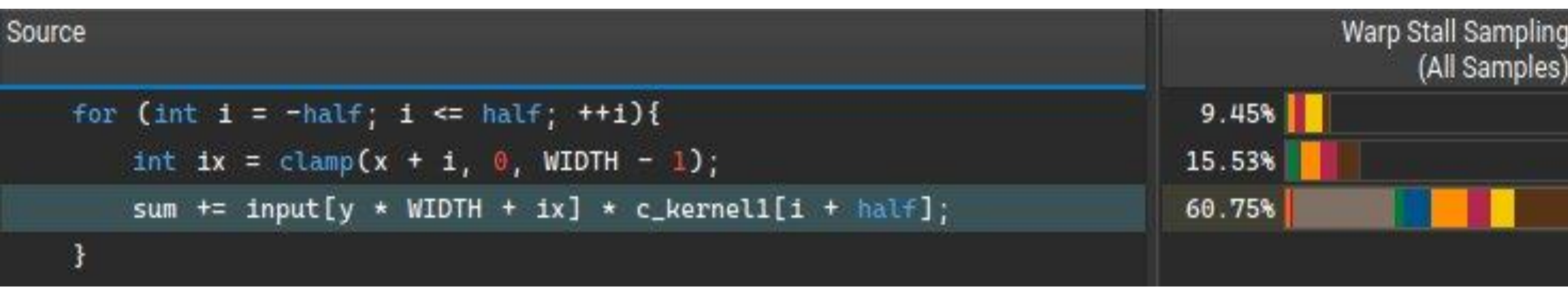


Where to improve?

- The main issue right now are global accesses to the input image
- We do a horizontal and a vertical pass for each Gaussian filter

For a total of **4 passes**

- Each time the image must be loaded from global memory



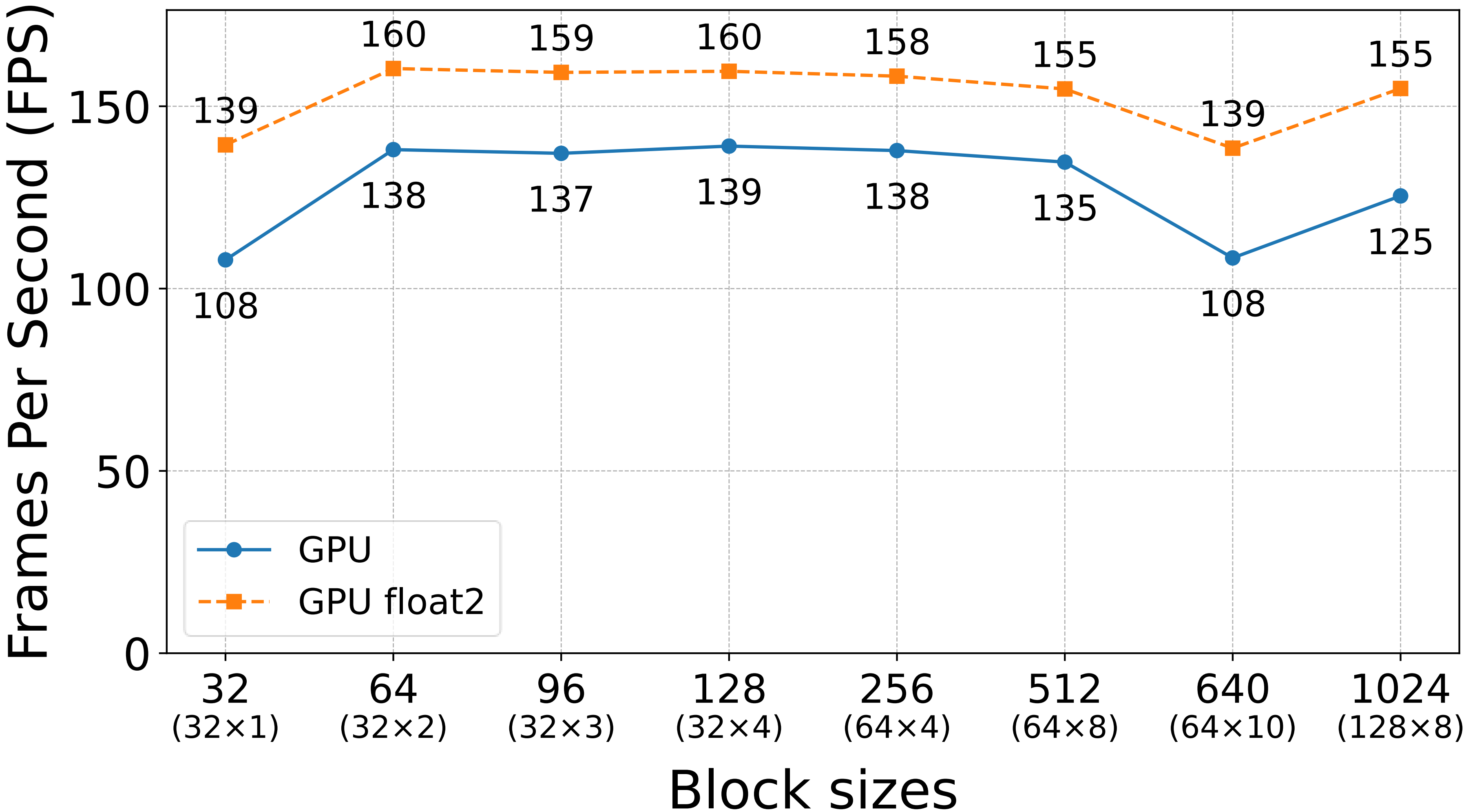
Solution: Data reuse

- We convolve by both kernels in the same function
- In the horizontal pass the number of reads is halved
- In the vertical pass accessing a matrix of float2 is faster

Compute (SM) Throughput [%]	52.70	(+1.70%)
Memory Throughput [%]	79.01	(+67.72%)
L1/TEX Cache Throughput [%]	65.90	(+58.91%)
L2 Cache Throughput [%]	79.01	(+67.72%)
DRAM Throughput [%]	24.96	(+0.57%)

Gains in vertical pass after using float2

V2 (float2) vs V1 performance



Where to improve?

- A lot of time is spent by copying the image from host to device and then from device to host
- For each frame 2 memcpy are needed, each of them adds overhead

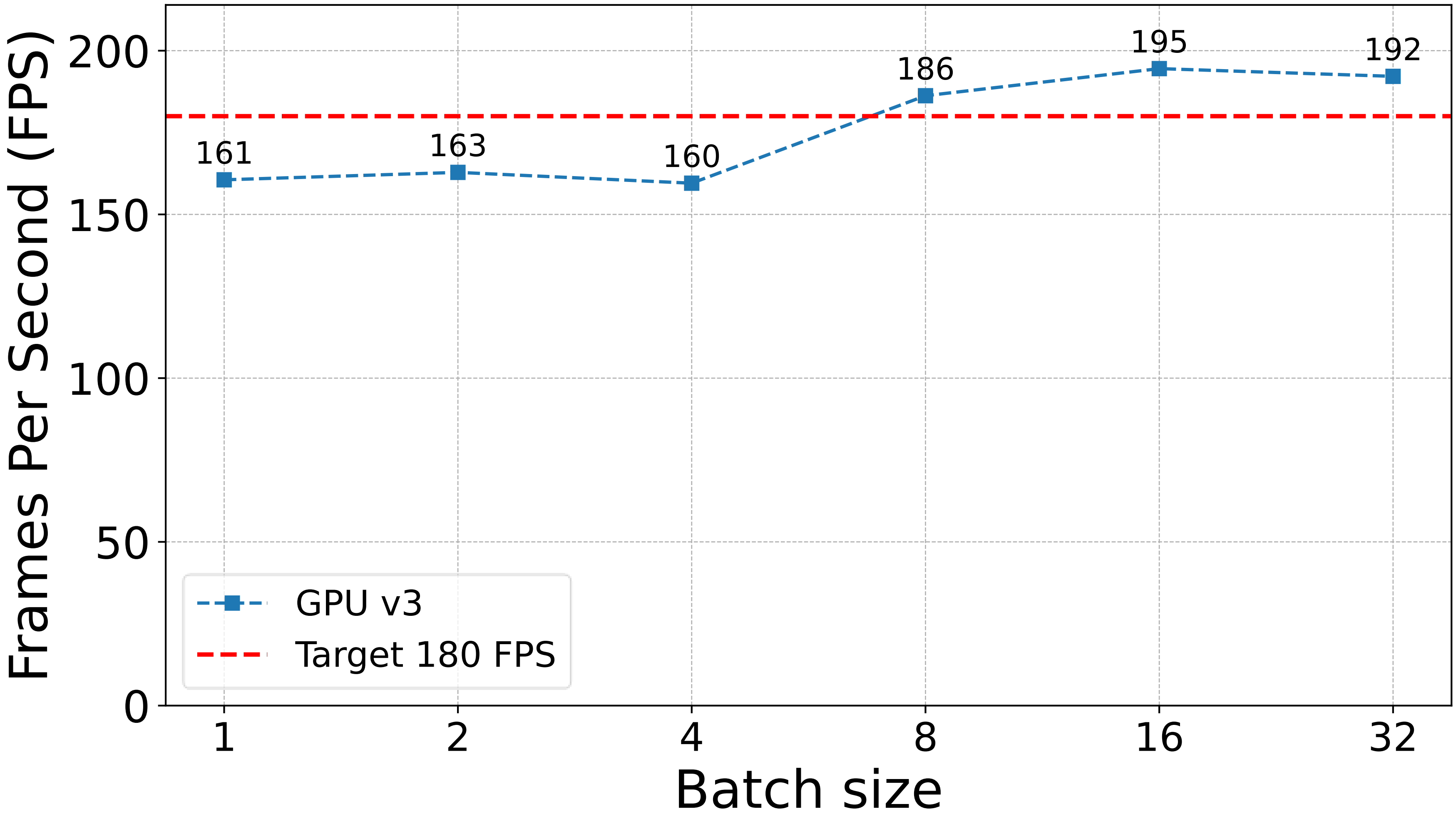
Time(%)	Time	Calls	Avg	Min	Max	Name
29.76%	97.212ms	250	388.85us	319.24us	453.01us	[CUDA memcpy DtoH]
27.82%	90.875ms	255	356.37us	511ns	1.7548ms	[CUDA memcpy HtoD]
25.47%	83.213ms	250	332.85us	325.96us	348.14us	blur_horizontal(unsigned char const *, float2*)
16.95%	55.366ms	250	221.47us	216.46us	233.00us	blur_vertical(float2 const *, unsigned char*)

Solution: Multiple Streams

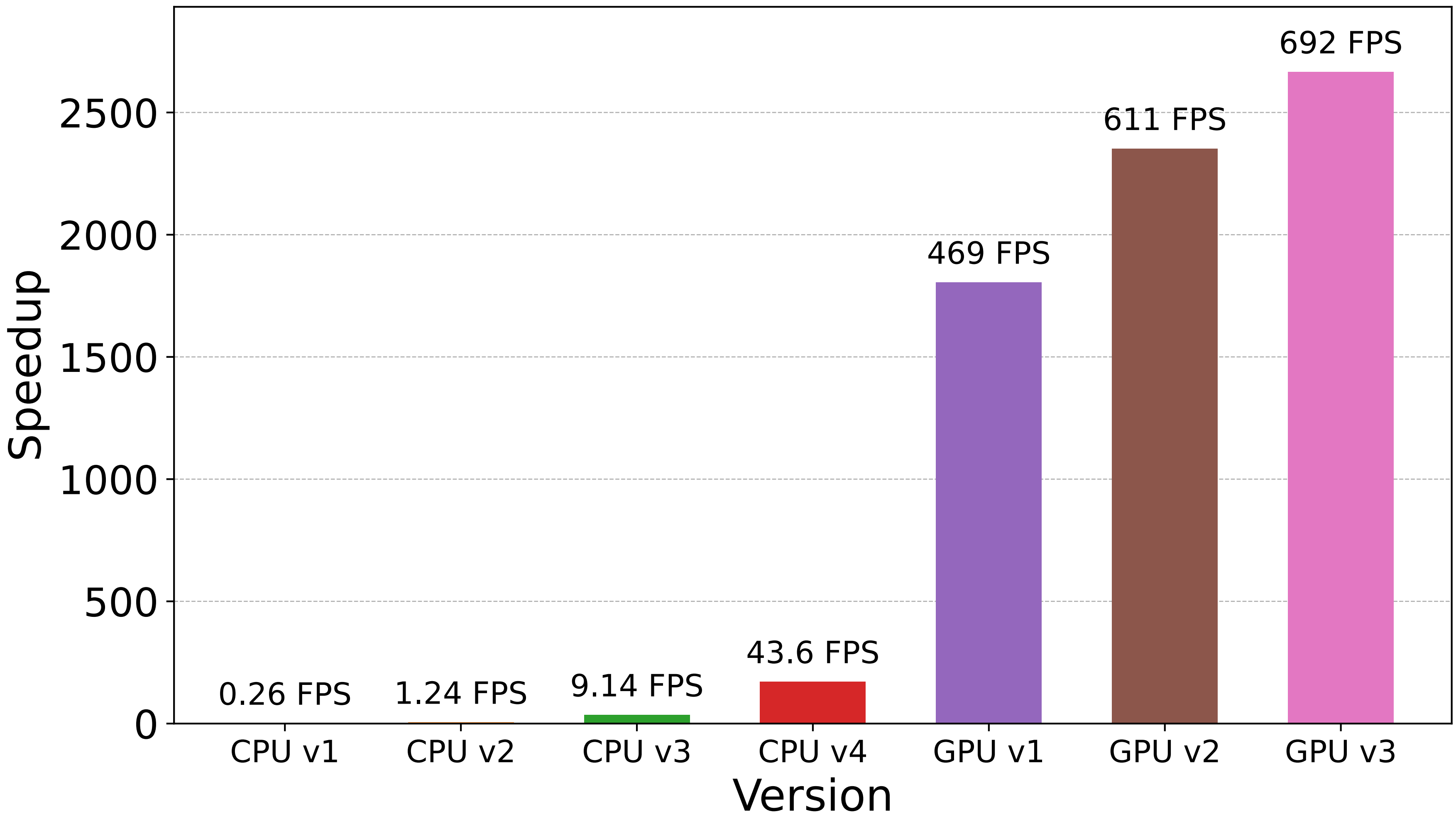
- Multiple frames are passed in a single function call
- Multiple streams and cudaMemcpyAsync make it so computation and memory transfer of different images can overlap

```
for (int i = 0; i < batchSize; ++i)
{
    int offset = i * img_size;

    cudaMemcpyAsync(&d_input[offset], &input[offset], img_size, cudaMemcpyHostToDevice, streams[i]);

    blur_horizontal<<<grid, block, 0, streams[i]>>>(&d_input[offset], &d_temp[offset]);
    blur_vertical<<<grid, block, 0, streams[i]>>>(&d_temp[offset], &d_output[offset]);

    cudaMemcpyAsync(&output[offset], &d_output[offset], img_size, cudaMemcpyDeviceToHost, streams[i]);
}
```

Results with Multi-Stream

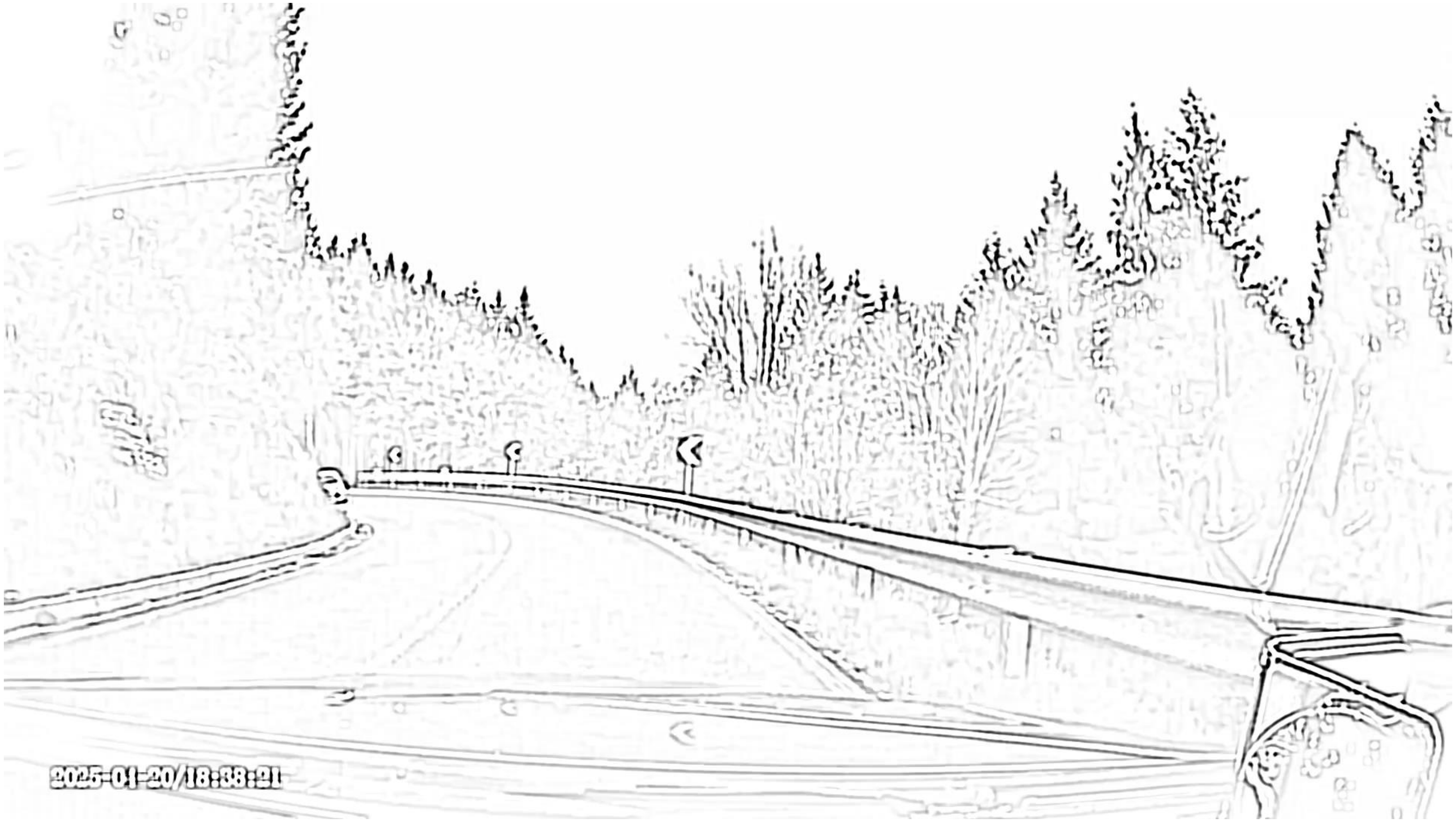


Final Results



Gpu v3 is **16 times** faster than Cpu v4 in 1080p

Thanks for the attention



Master’s Degree in Computer Engineering
Academic Year 2024/2025