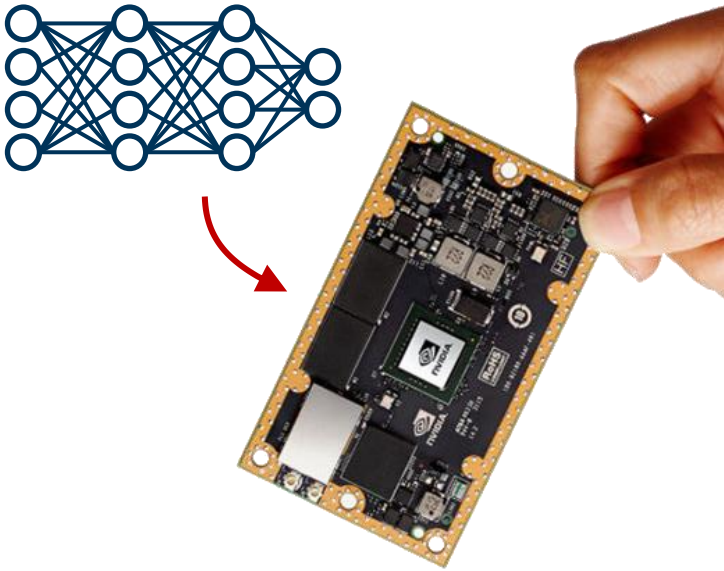
Speeding up Deep Neural Networks on the Jetson TX1



Markus Eisenbach

Ilmenau University of Technology (Germany)

Neuroinformatics and Cognitive Robotics Lab
markus.eisenbach@tu-ilmenau.de
www.tu-ilmenau.de/neurob

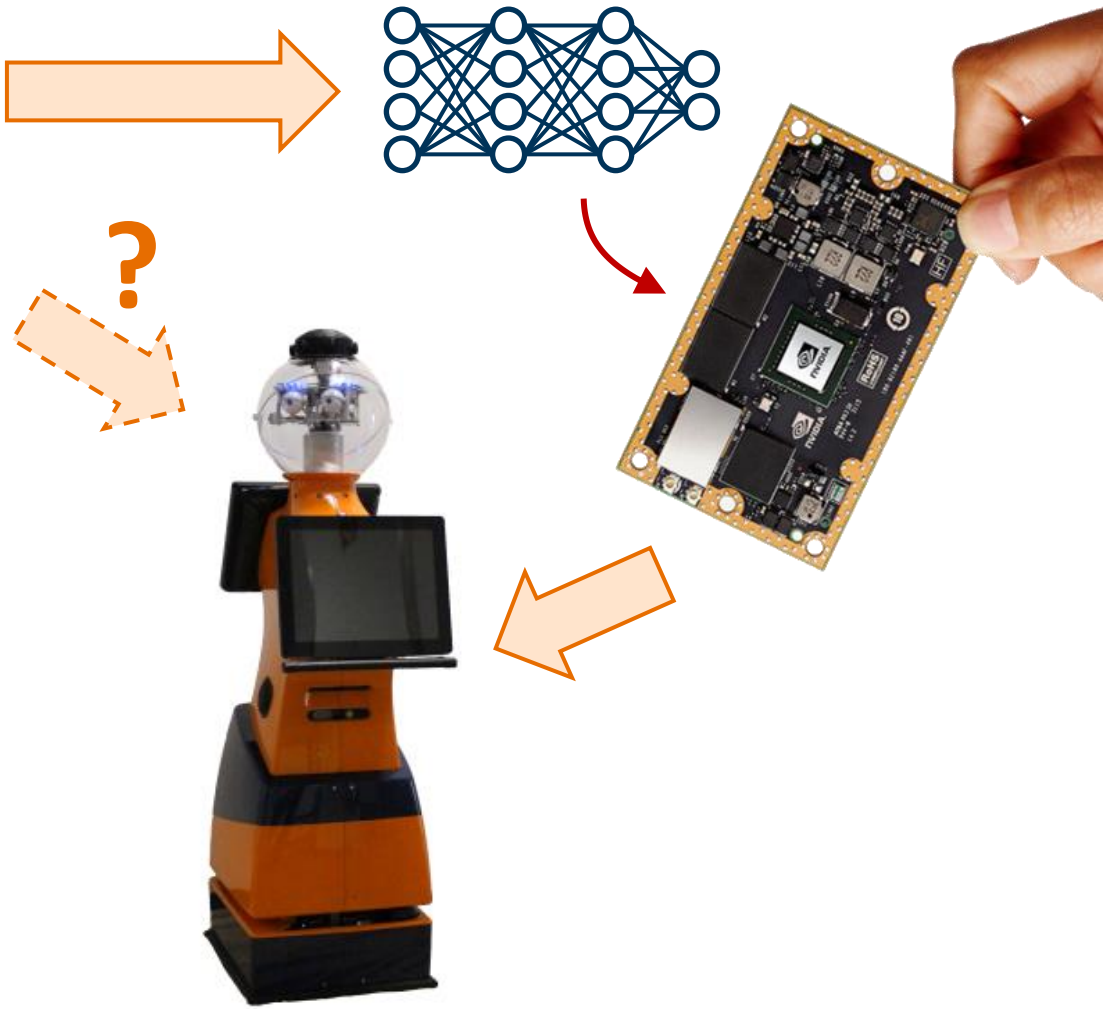




R. Stricker, D. Seichter, A. Vorndran, T. Wengefeld, H.-M. Gross
Ilmenau University of Technology
Germany

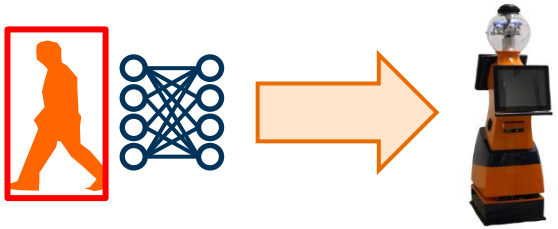
Motivation

[IJCNN 2016]



Motivation

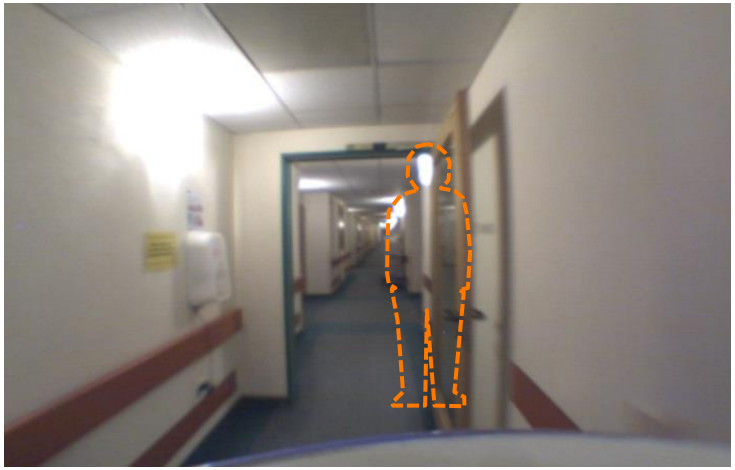
Why do we need a DL-based person detector on a robot?



Why a DL-based person detector?
– Motivation from a robotic viewpoint

Do state of the art person detectors perform well in real-world applications?

Example: Navigation



Many **false detections near doors** (vertical gradients)

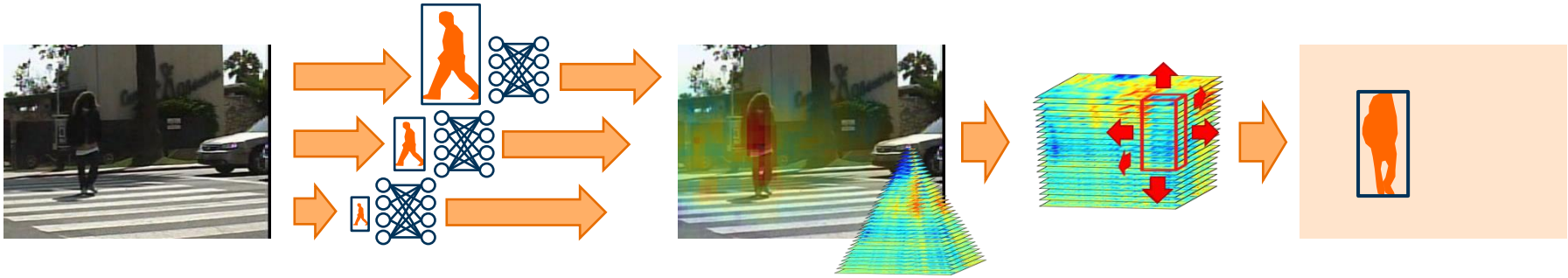
→ **Robot does not pass** the door due to possible personal space violation

Outline

What can you expect?

Outline

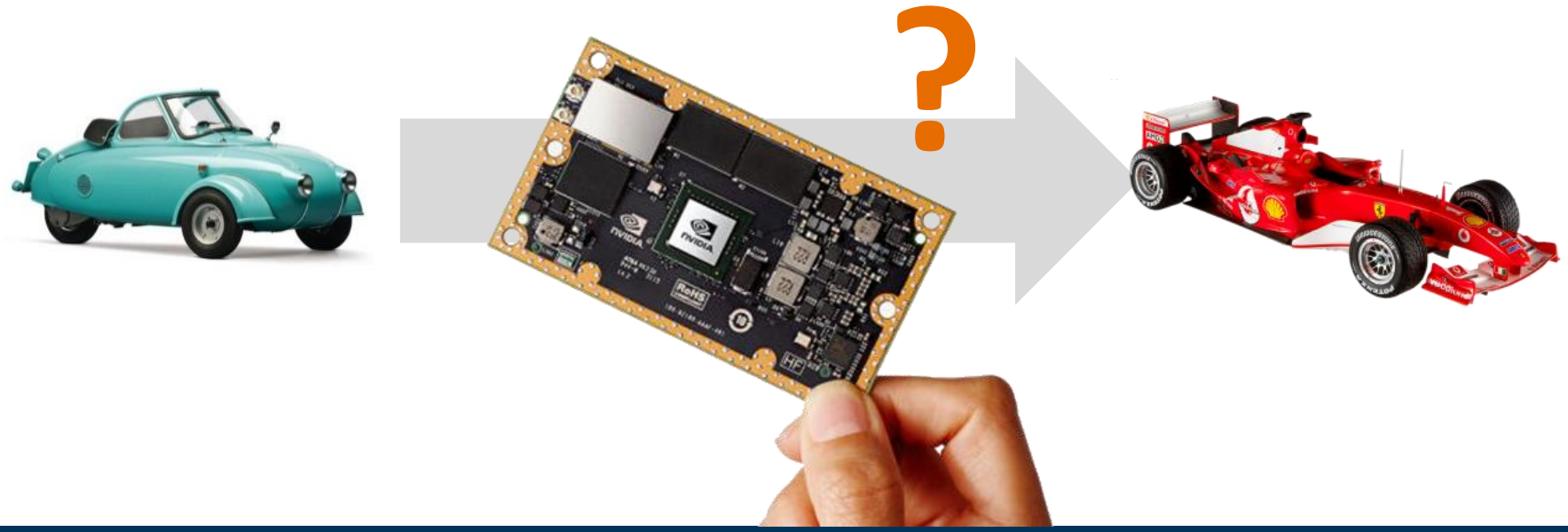
Person detector in a nutshell



Is the Jetson TX1 really necessary?

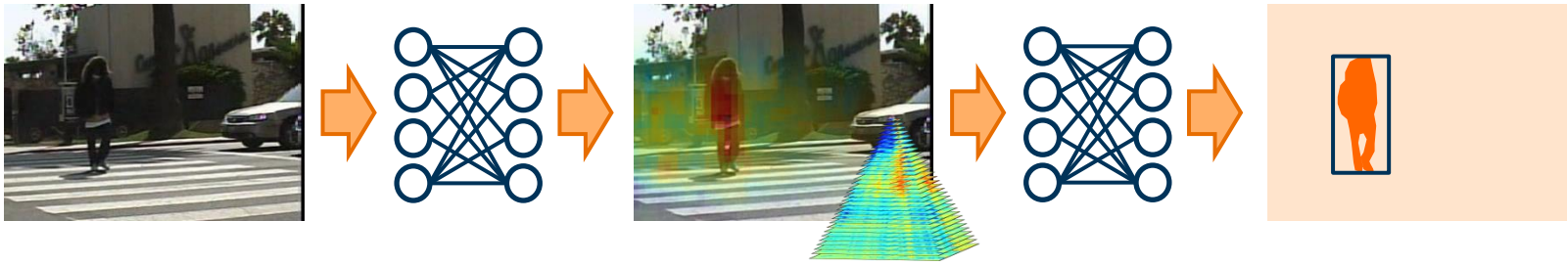
Outline

How can the run time be improved on this device without a loss in accuracy?

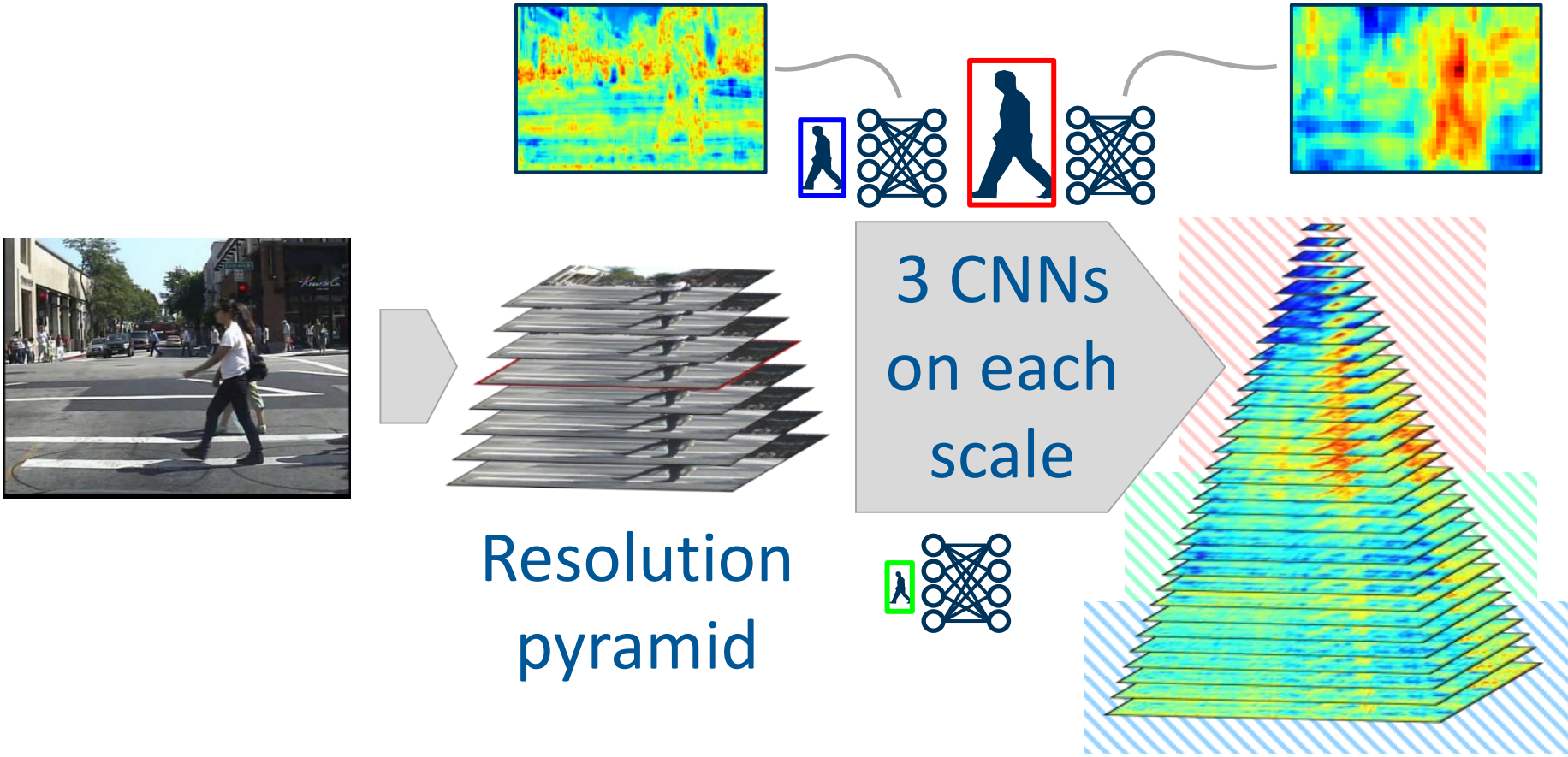


Our person detector in a nutshell

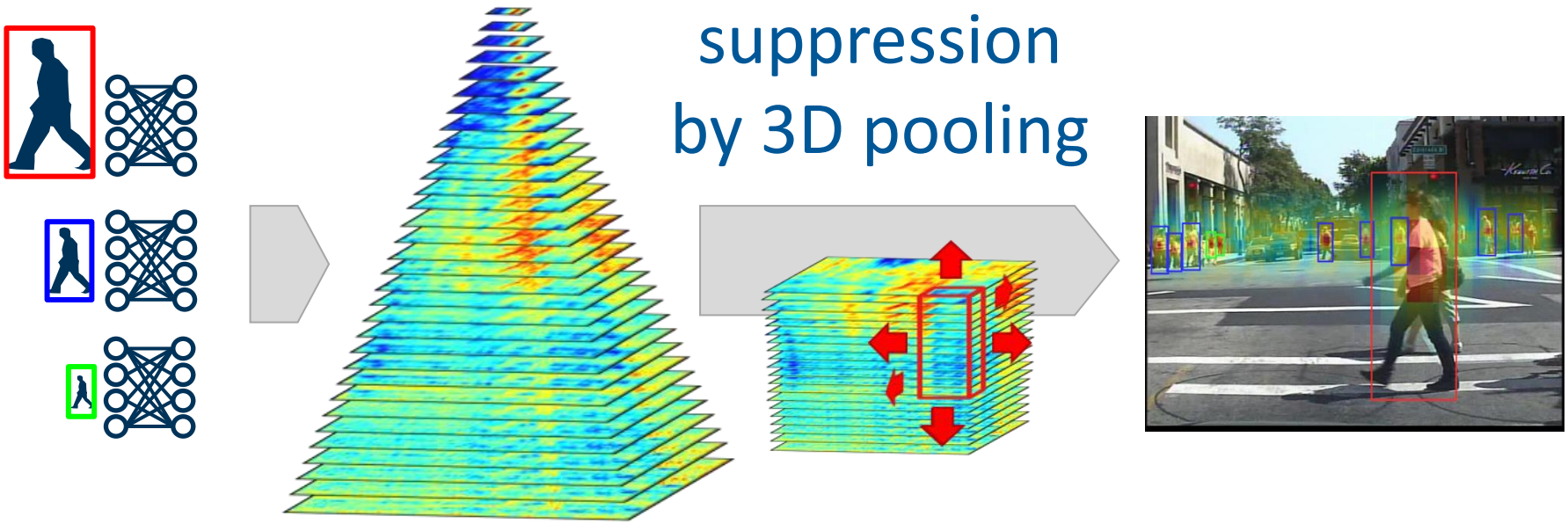
What is the basic idea?



Person detector in a nutshell



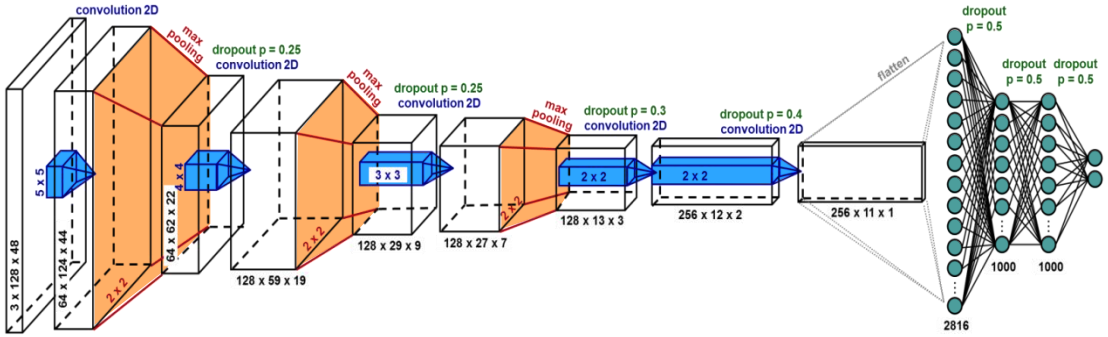
Person detector in a nutshell



Person detector in a nutshell

Input 128 x 48 x 3

Output



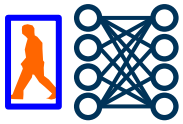
5 Conv layers: sizes 2x2 – 5x5

(1 – 3) 2x2 Max pooling layers with **stride 2**

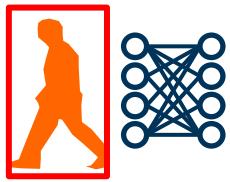
2 FC layers + Softmax output layer → **Conv layers**

Detector – Visual Examples

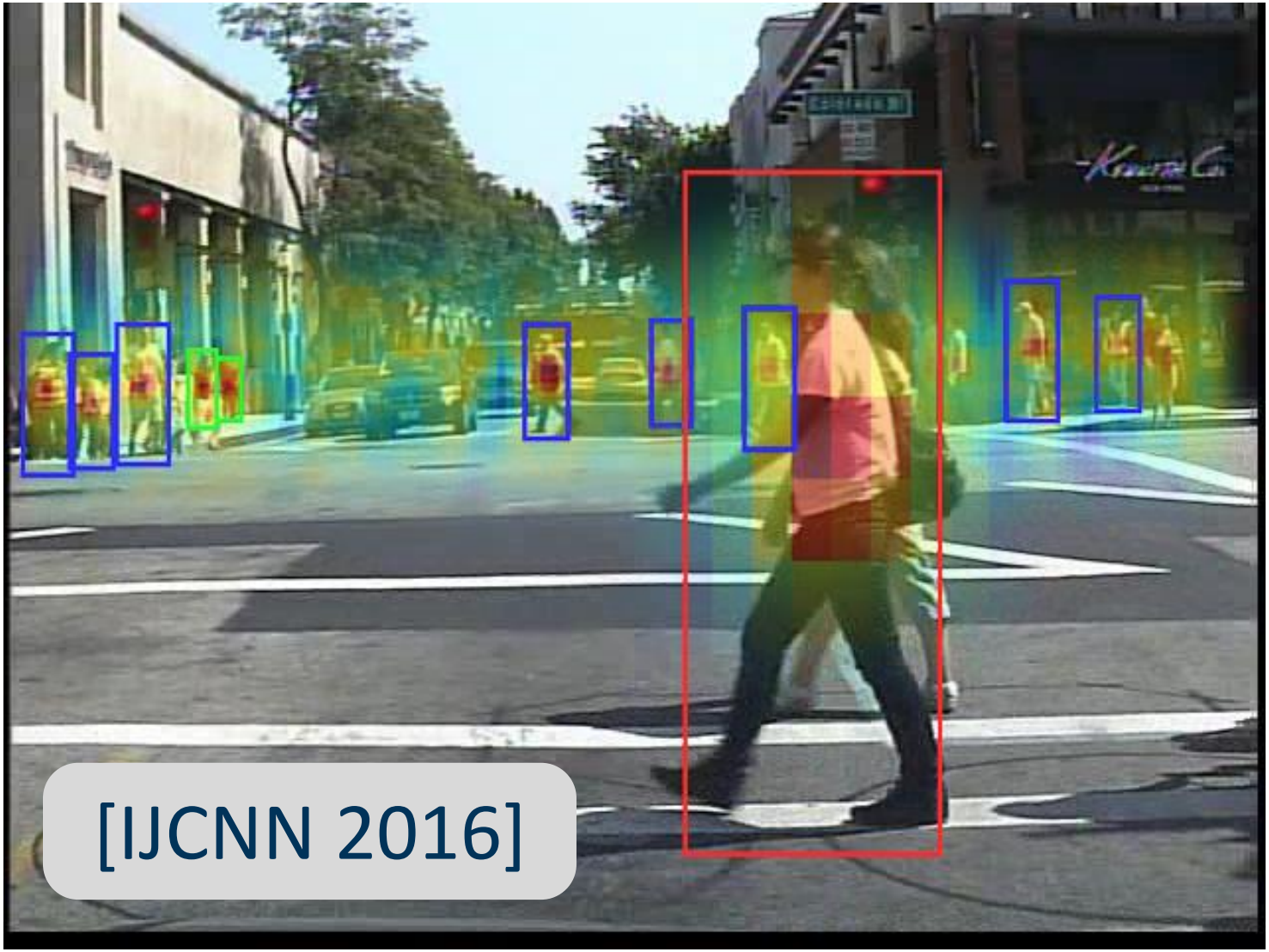
Far



Medium



Near

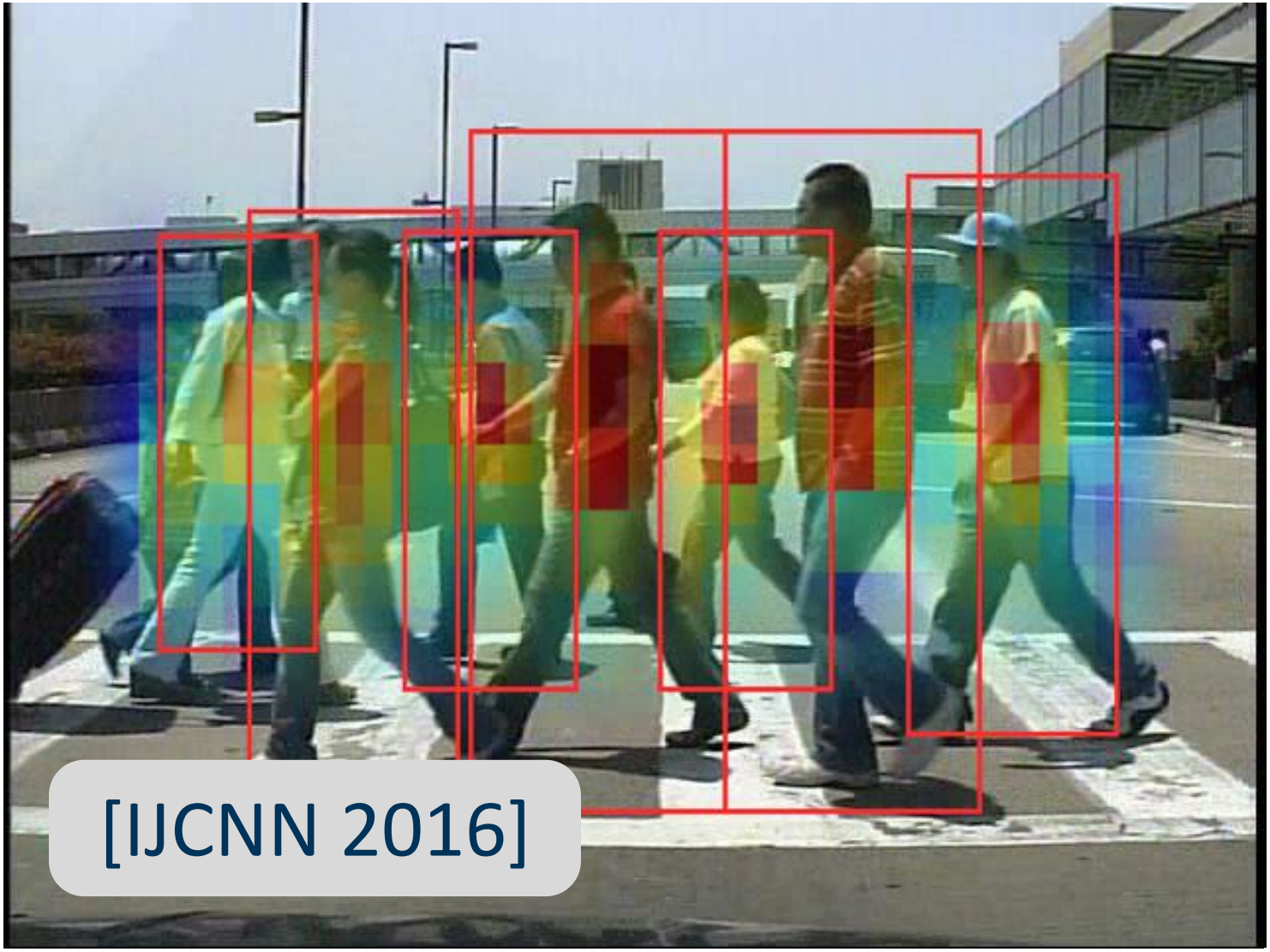


Detector – Visual Examples

Far

Medium

Near



Why do we need the Jetson TX1?

Why do we need the Jetson TX1?

Why do we need the Jetson TX1?

Why do we need the Jetson TX1?

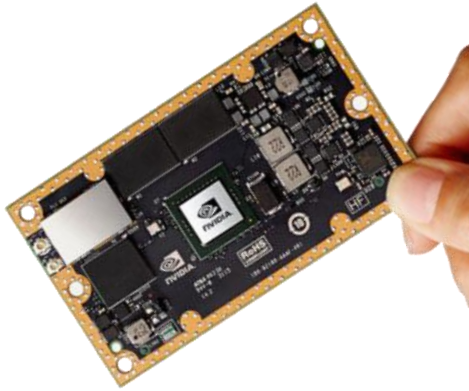
… and run time?

8.391 s

Jetson TX1

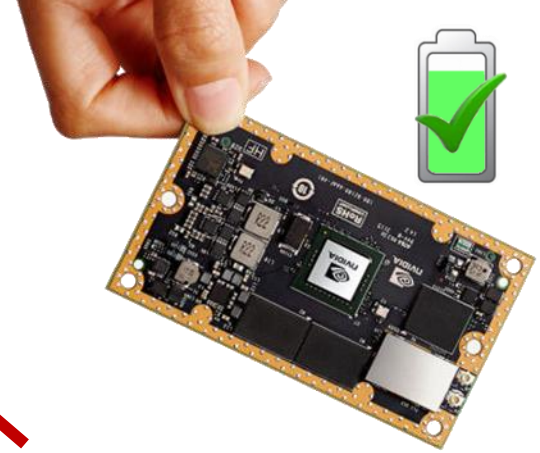
0.462 s

Titan X



8.391 s

Jetson TX1



?

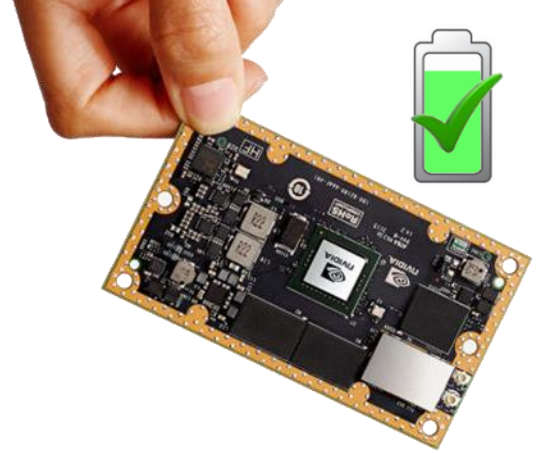
How can we improve the run time?

0.462 s

Titan X

8.391 s

Jetson TX1

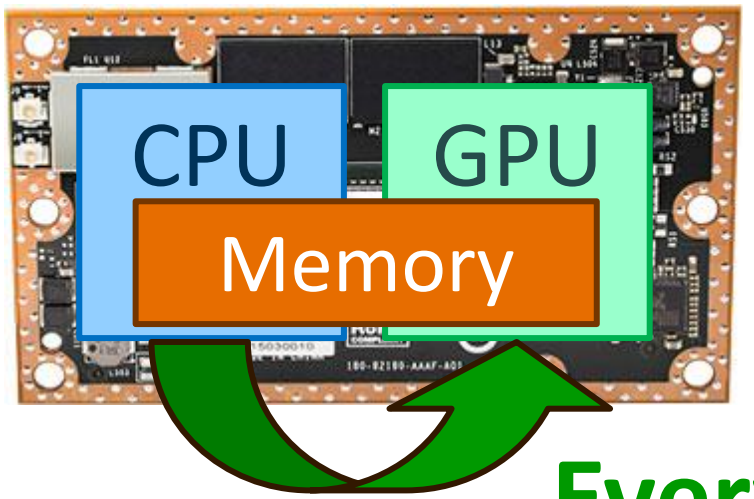


Computation graph optimization and overhead removal

0.462 s

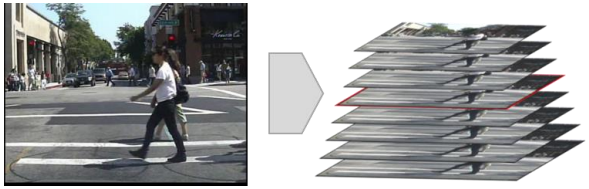
Titan X

Computation graph optimization and overhead removal

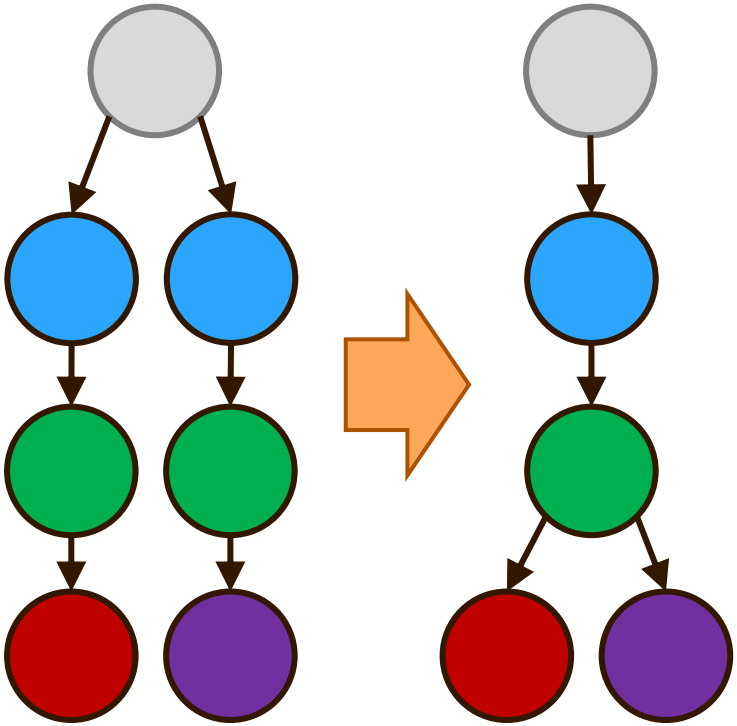


Copy operations between CPU and GPU (Frameworks)

Every operation on GPU (incl. non-DL operations)



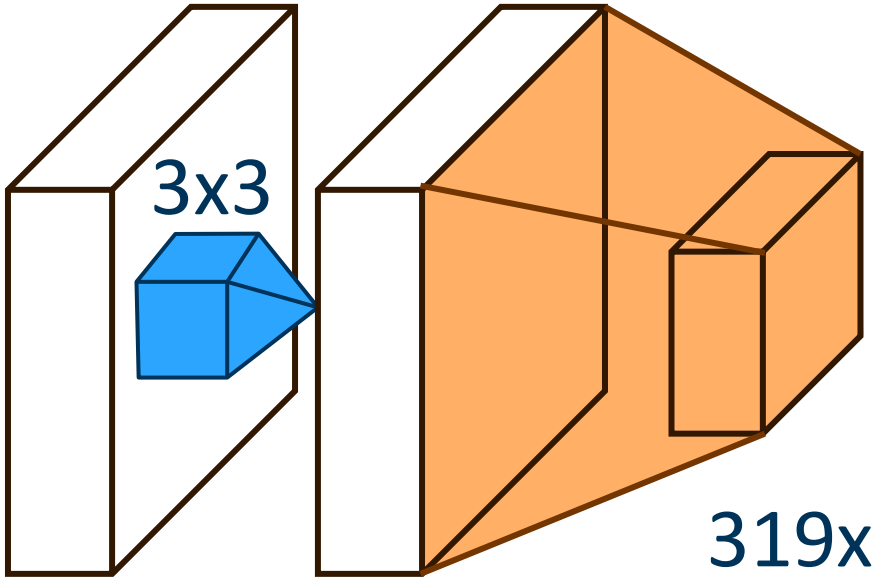
Computation graph optimization and overhead removal



Remove redundancies from the computation graph

Computation graph optimization and overhead removal

640x480x3



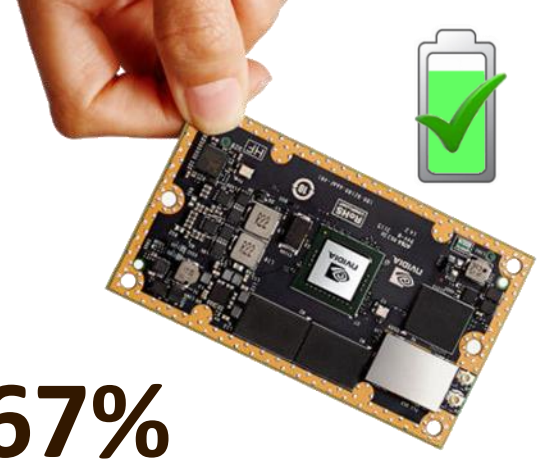
319x239x128

638x478x128

Specify exact shapes to make all tensor sizes static

8.391 s

Jetson TX1



67%

2.8 s

Remove Overhead

Reduce floating point precision

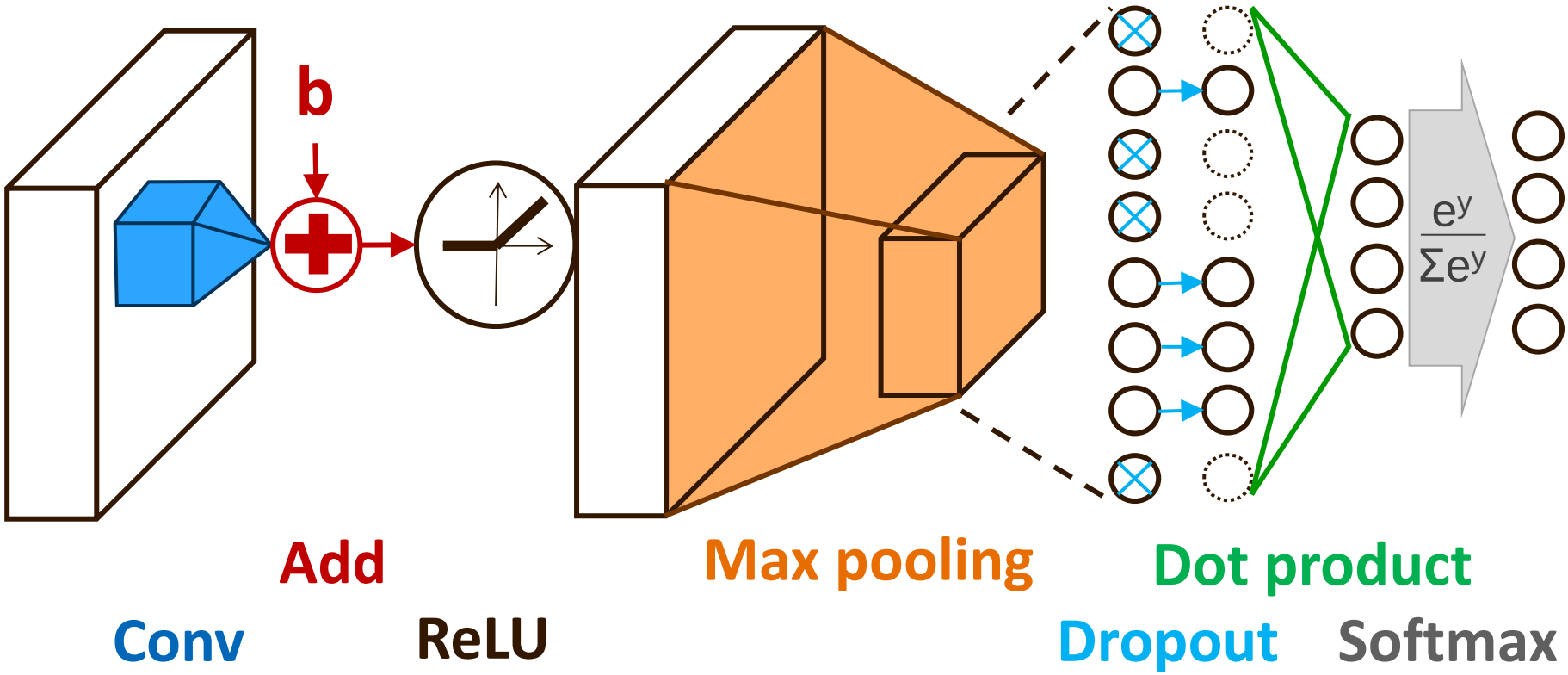
0.462 s

Titan X

Reducing floating point precision

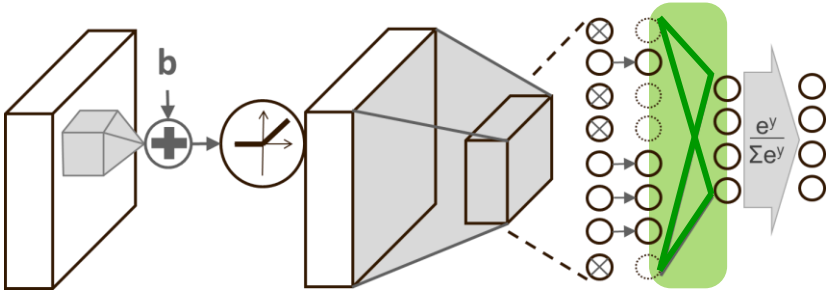
float 32 → float 16

Typical Deep Learning operations:

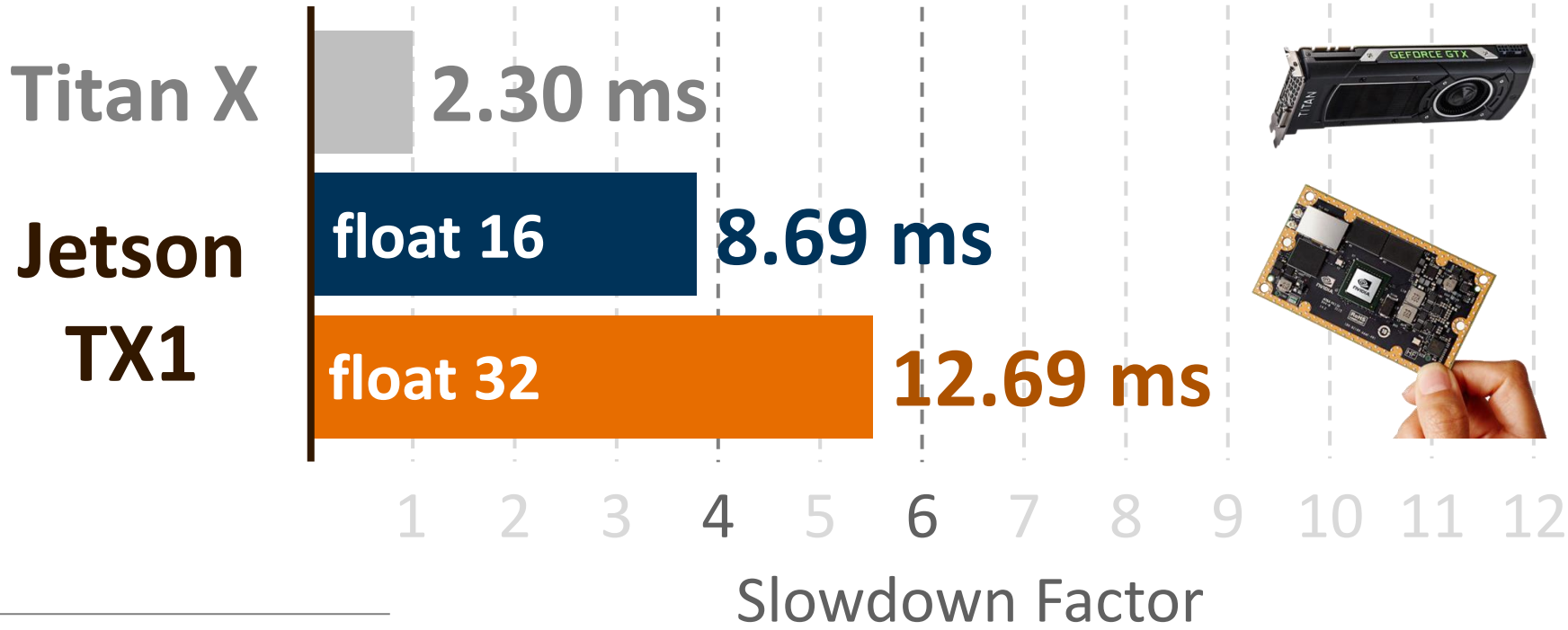


Reducing precision

float 32 → float 16



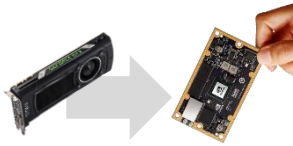
Dot product (fully connected layers)



(Matrix: 1000x1000)

Reducing floating point precision

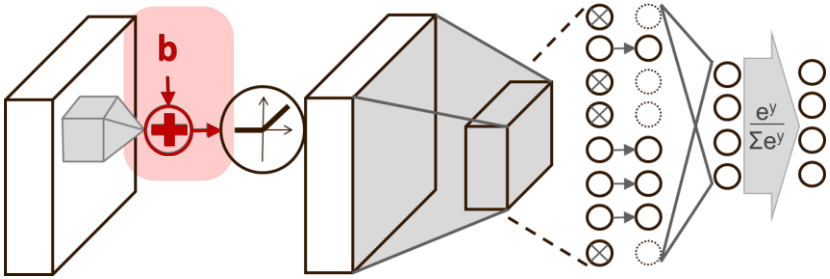
Summary

Slowdown factor 

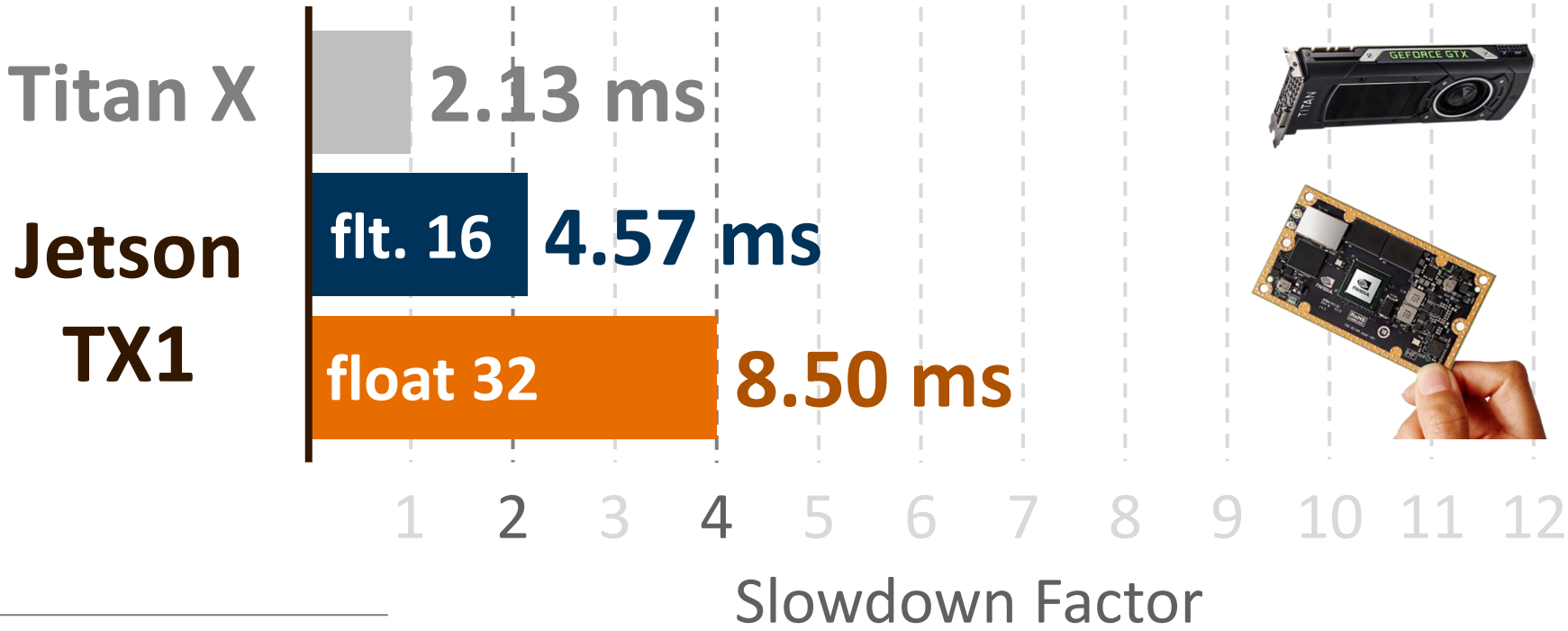
	float 32	→	float 16
Matrix multiplication	6	– 33% →	4

Reducing precision

float 32 → float 16



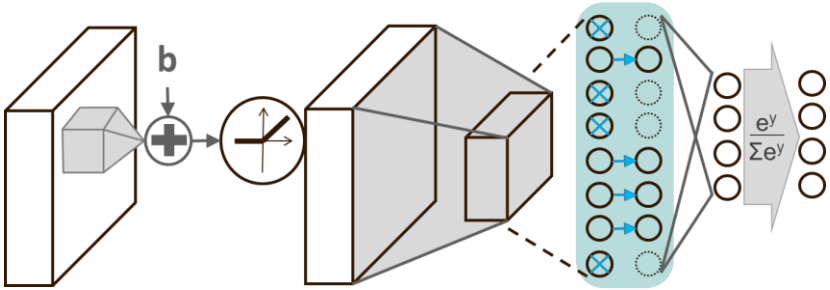
Element-wise add (bias)



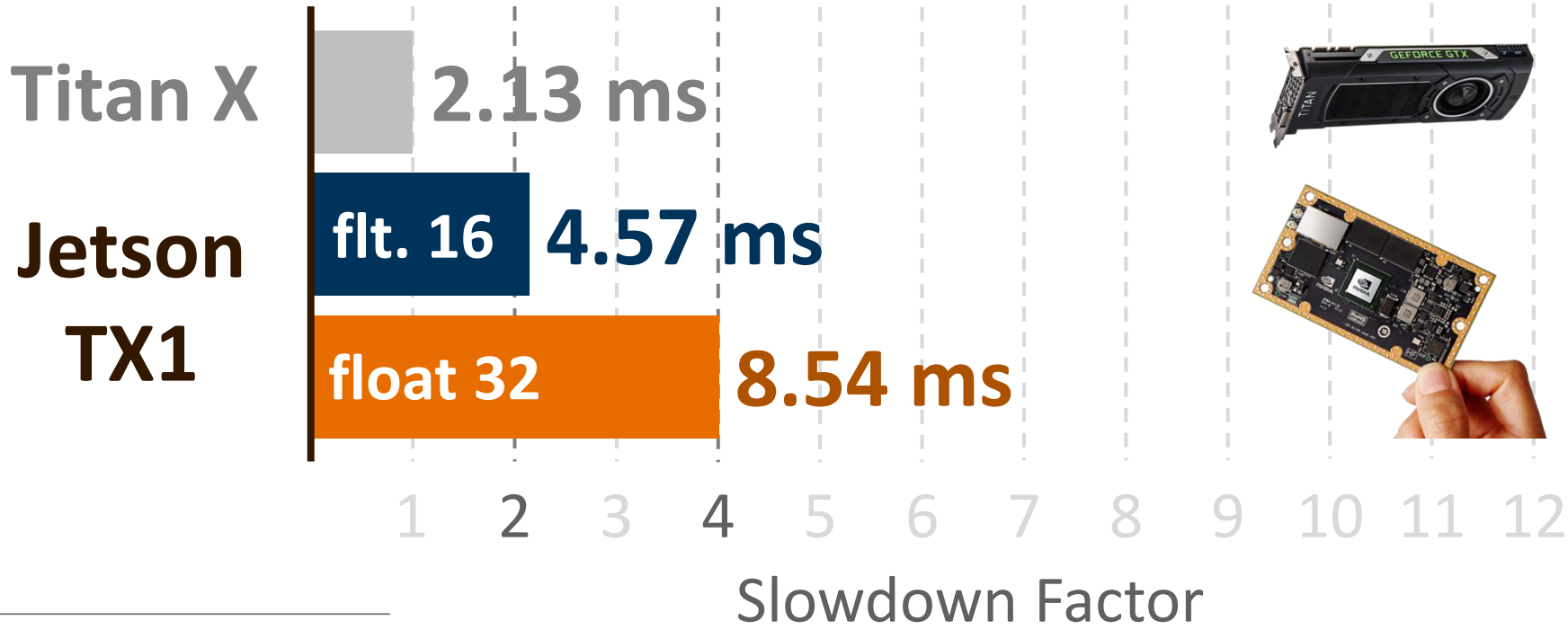
(Matrix: 1000x1000)

Reducing precision

float 32 → float 16



Element-wise multiplication (dropout)

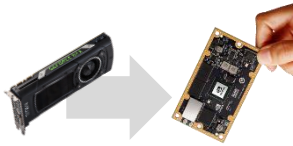


(Matrix: 1000x1000)

Reducing floating point precision

Summary

Slowdown factor

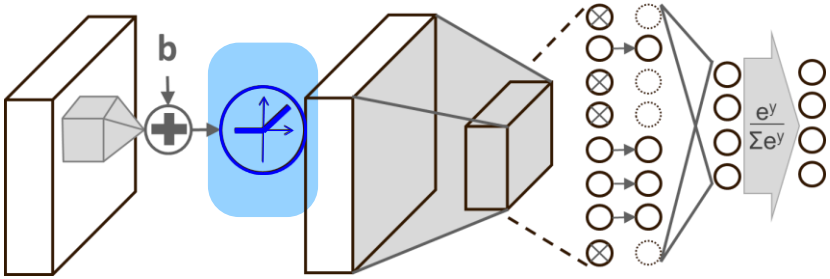


float 32 → **float 16**

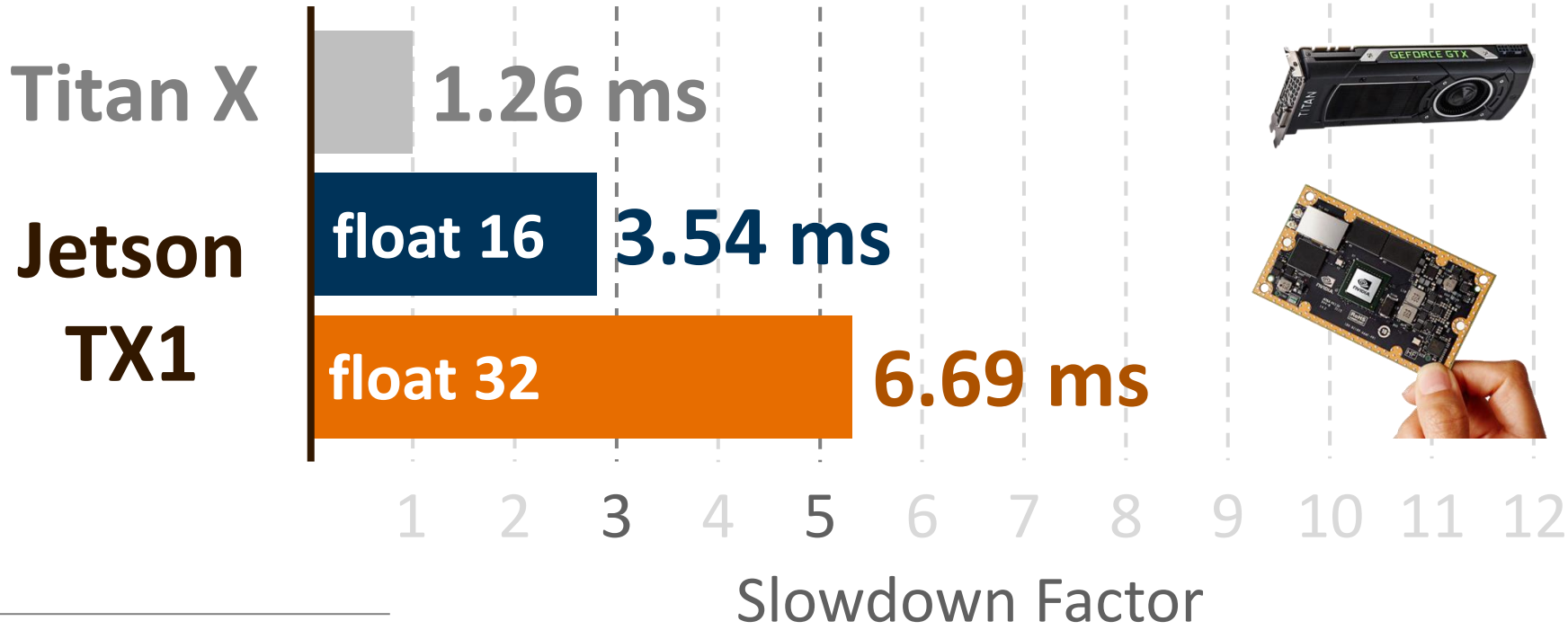
Matrix multiplication	6	– 33% →	4
Element-wise operations	**4**	**– 50%** →	**2**

Reducing precision

float 32 → float 16



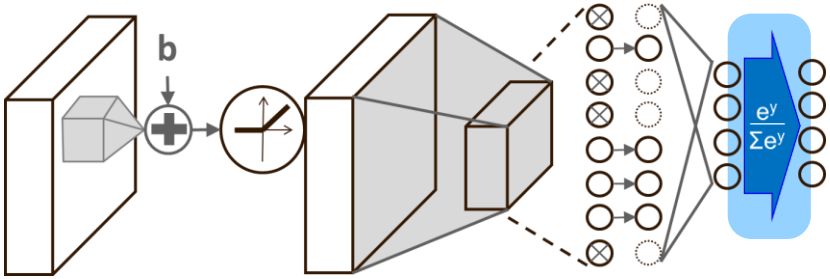
Element-wise non-linearity (ReLU activation)



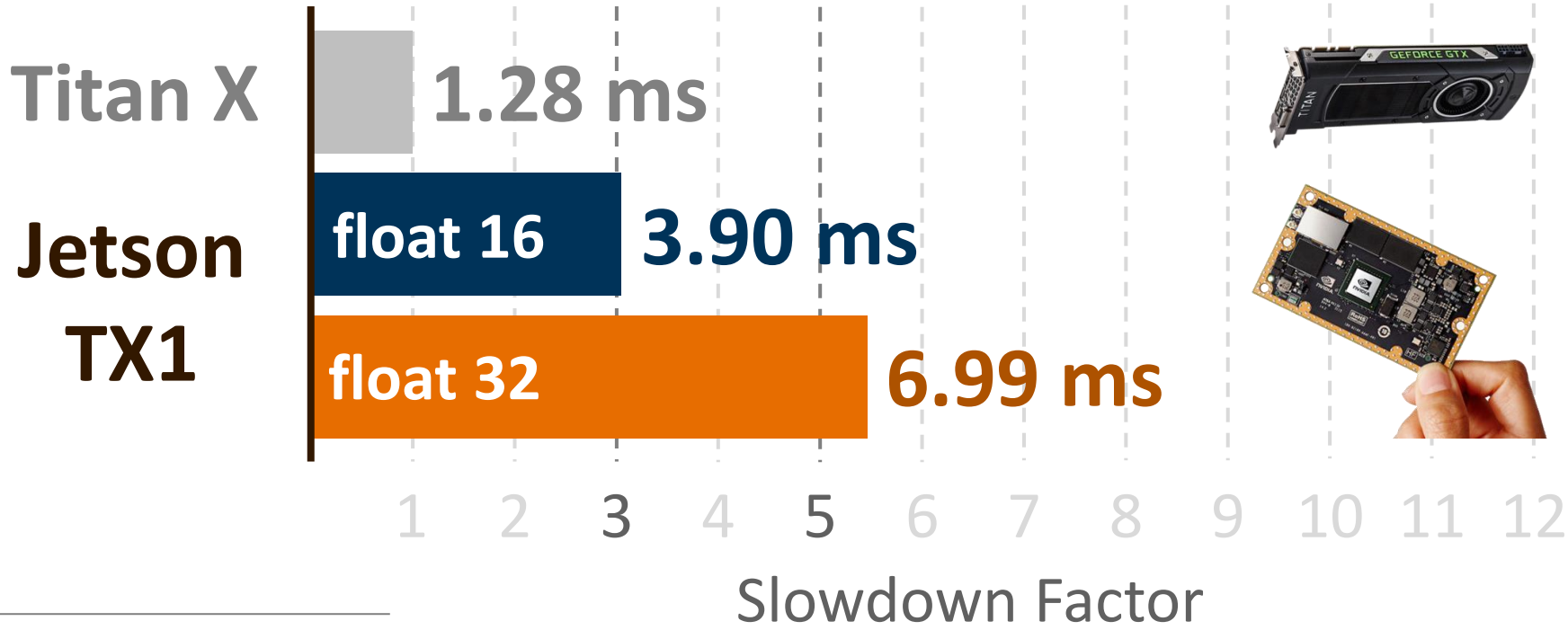
(Matrix: 1000x1000)

Reducing precision

float 32 → float 16



Element-wise non-linearity (Softmax output)

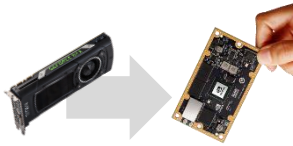


(Matrix: 1000x1000)

Reducing floating point precision

Summary

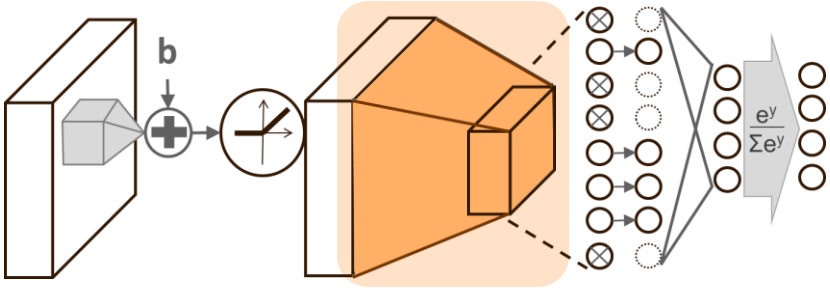
Slowdown factor



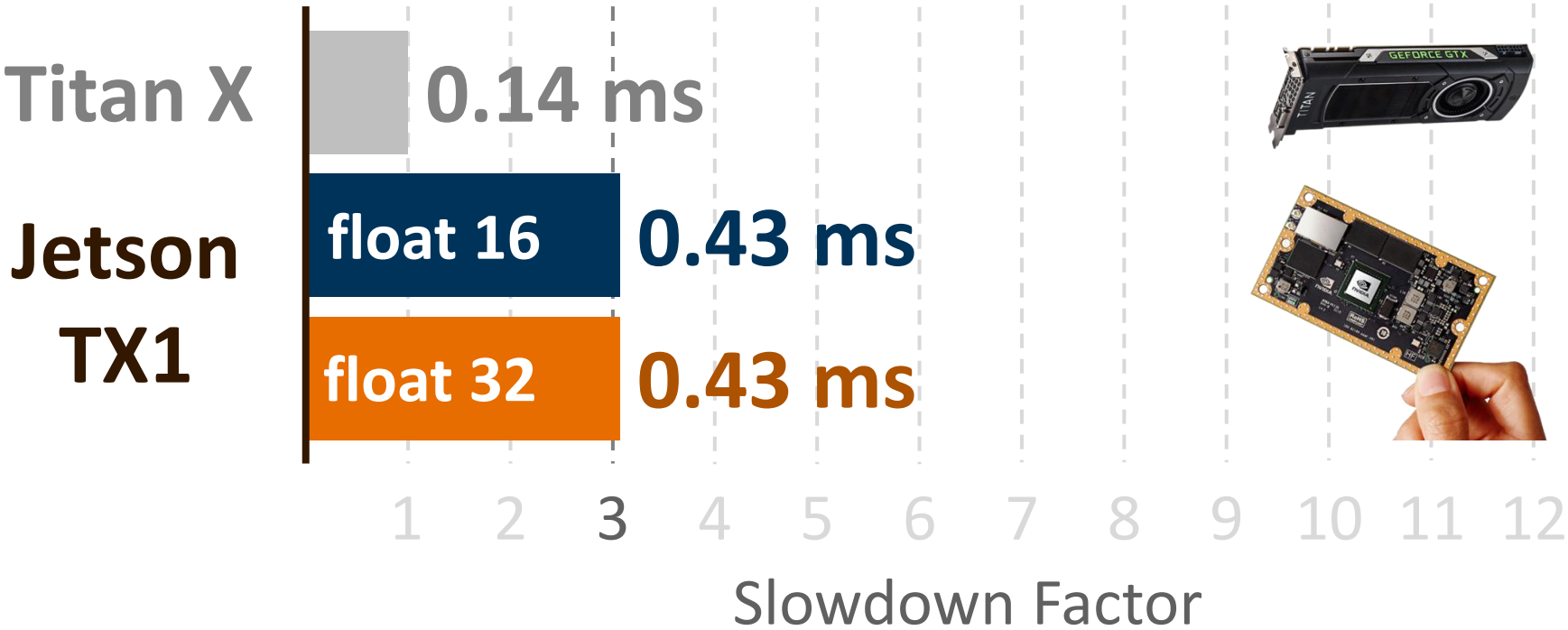
	float 32	→	float 16
Matrix multiplication	6	– 33% →	4
Element-wise operations	4	– 50% →	2
Non-linear activations	5	– 40% →	3

Reducing precision

float 32 → float 16



Max pooling

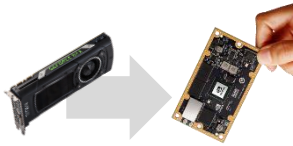


Input Volume: 3 x 320 x 240, 200 filters, pooling/stride 2x2

Reducing floating point precision

Summary

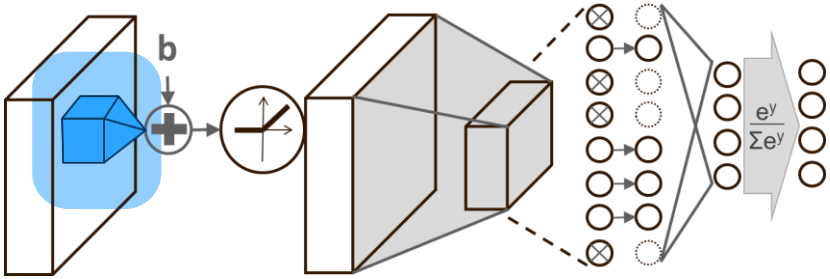
Slowdown factor



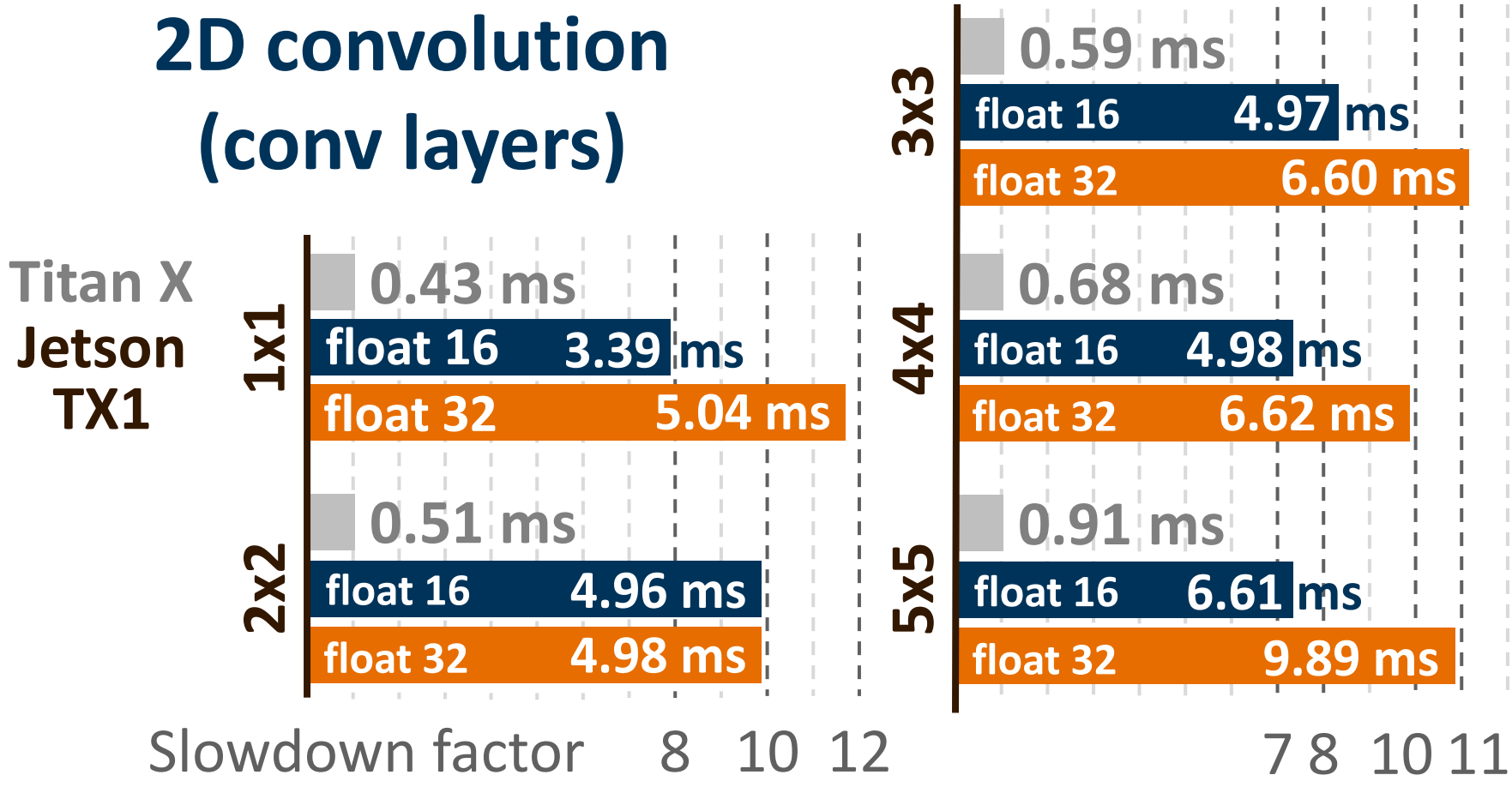
	float 32	→	float 16
Matrix multiplication	6	– 33% →	4
Element-wise operations	4	– 50% →	2
Non-linear activations	5	– 40% →	3
Pooling	3	= →	3

Reducing precision

float 32 → float 16



2D convolution (conv layers)

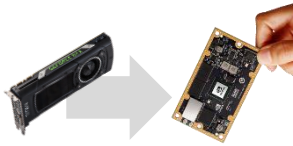


Input Volume: 3 x 320 x 240, no padding, 200 filters, batch size 1

Reducing floating point precision

Summary

Slowdown factor



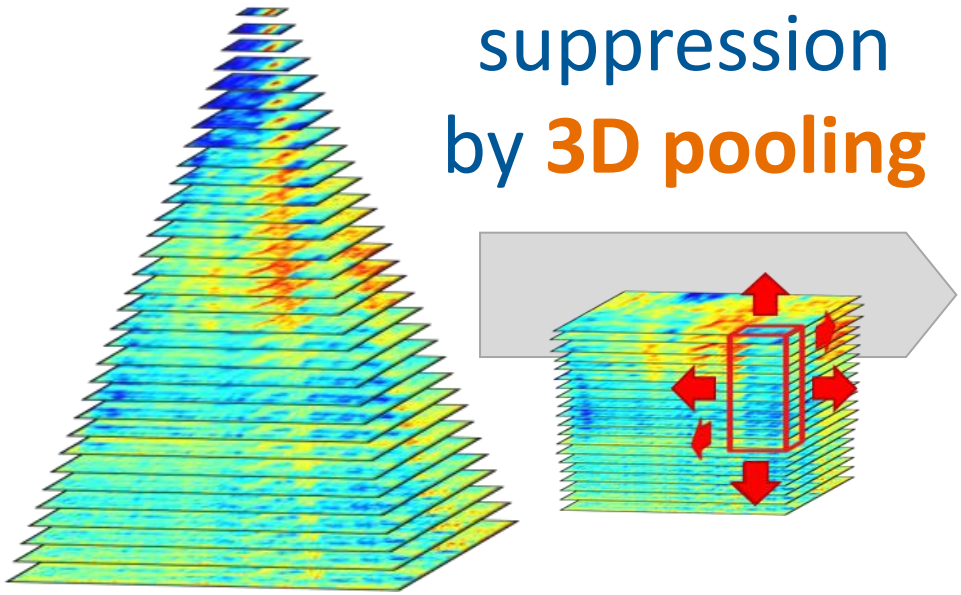
	float 32	→	float 16
Matrix multiplication	6	– 33% →	4
Element-wise operations	4	– 50% →	2
Non-linear activations	5	– 40% →	3
Pooling	3	= →	3
Convolutions	**10-11**	**– 25-30% →**	**7-8**

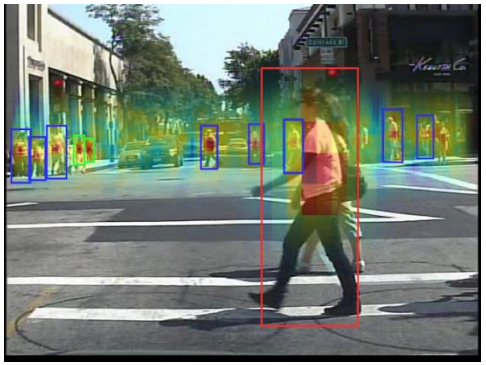
Reducing floating point precision

float 32 → float 16

Rarely used Deep Learning operations

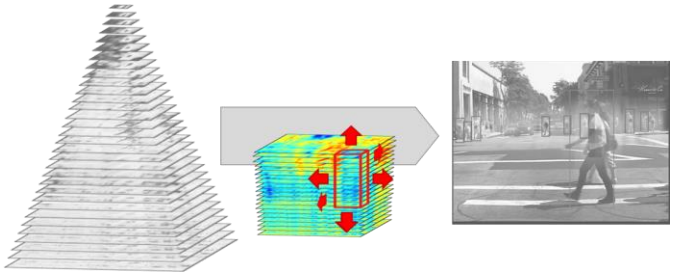
Non-maximum suppression by **3D pooling**



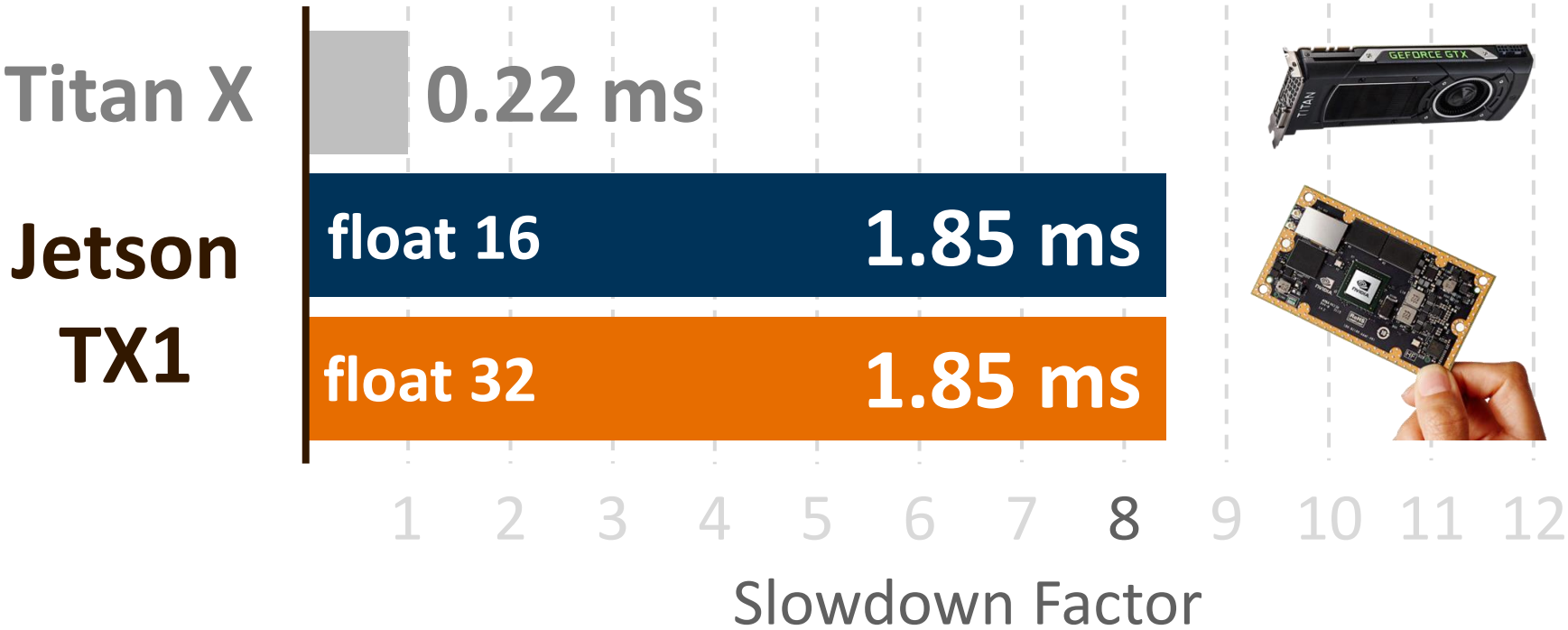


Reducing precision

float 32 → float 16



3D pooling (Non-maximum suppression)

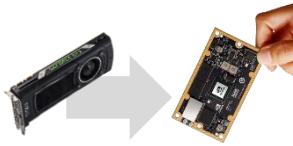


Input Volume: 3 x 320 x 240, pooling 3x7x3, stride 1x1x1

Reducing floating point precision

Summary

Slowdown factor

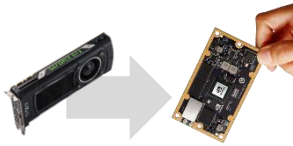


	float 32	→	float 16
Matrix multiplication	6	– 33% →	4
Element-wise operations	4	– 50% →	2
Non-linear activations	5	– 40% →	3
Pooling	3	= →	3
Convolutions	10-11	– 25-30% →	7-8
3D Pooling	8	= →	8

Reducing floating point precision

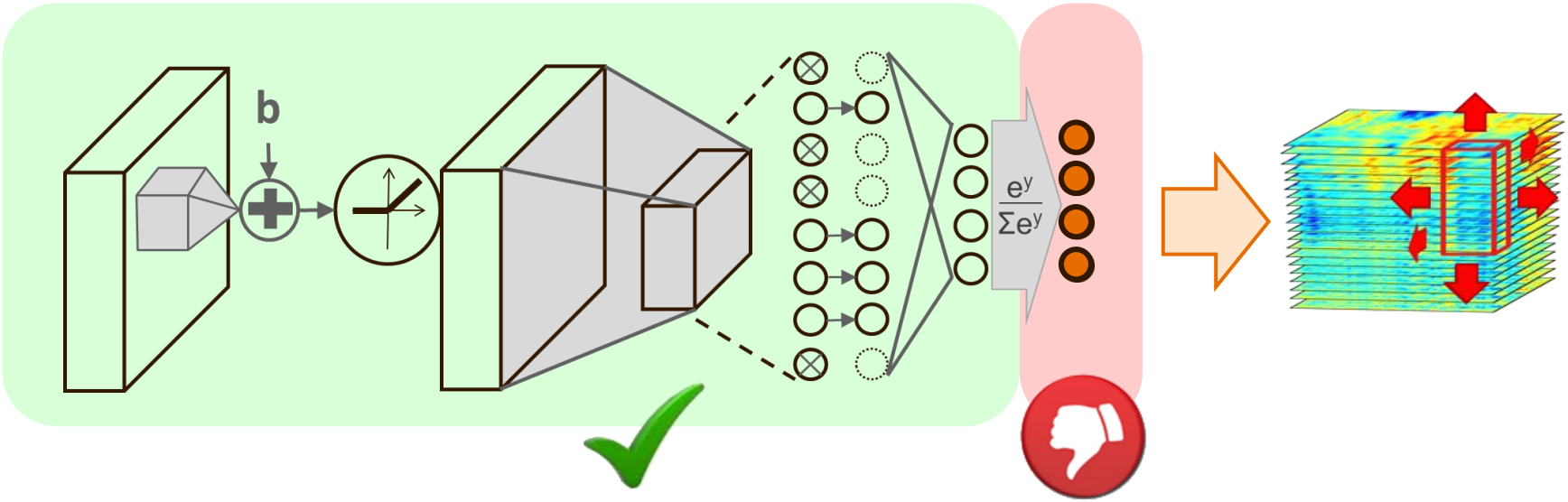
Summary

Slowdown factor



	float 32	→	float 16
Matrix multiplication	6	– 33% →	4
Element-wise operations	4	– 50% →	2
Non-linear activations	5	– 40% →	3
Pooling	3	= →	3
Convolutions (>90%)	10-11	– 25-30% →	7-8
3D Pooling	8	= →	8
Overall	10	– 25% →	7.5

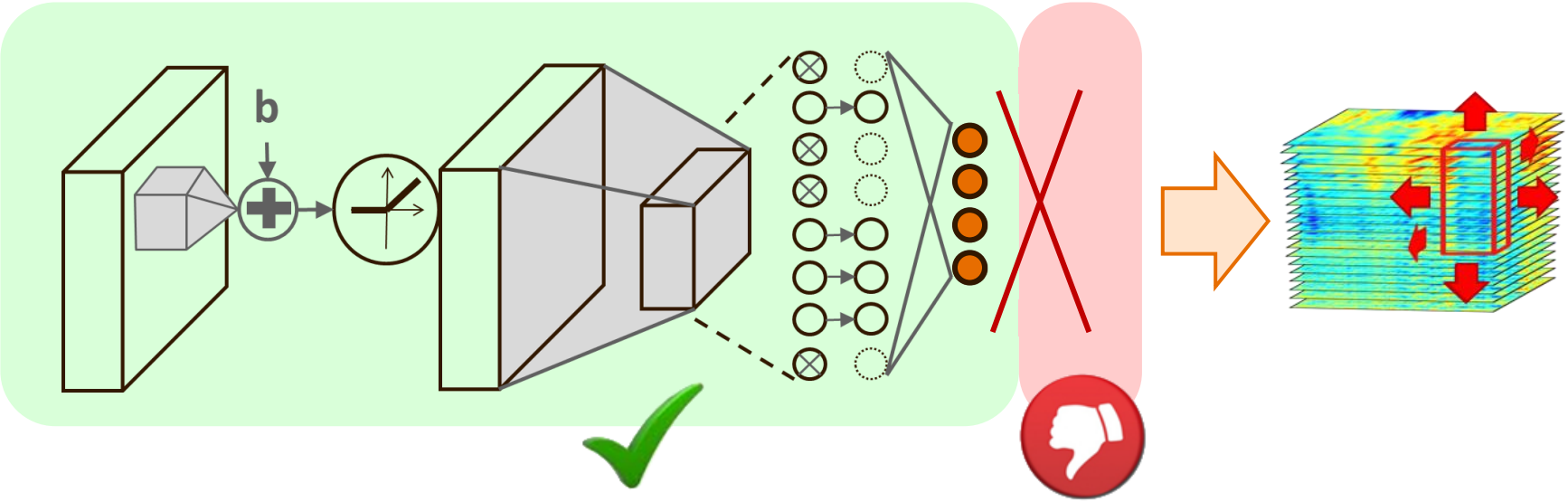
Loss of precision / accuracy ?



Loss of precision after exponential function

Problematic for NMS

Loss of precision / accuracy ?



Loss of precision after exponential function

Problematic for NMS

Solution: Use linear output for NMS

Loss of precision / accuracy ?

Which precision do you need?

→ It depends on dataset

[Rzayev et al., IJCNN 2017]

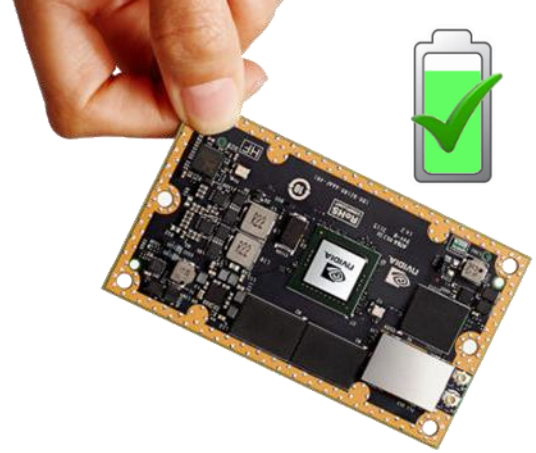
On ImageNET:

Model	**32-float**	12-bit	**10-bit**	8-bit	6-bit
VGG19	**71.36%**	71.35%	**71.34%**	70.88%	56.00%
ResNet152	**77.55%**	77.51%	**77.40%**	74.95%	9.29%
SqueezeNetV1	**56.52%**	56.52%	**56.24%**	54.56%	17.10%
InceptionV3	**76.41%**	76.43%	**76.44%**	73.67%	1.50%

https://github.com/aaron-xichen/pytorch-playground/blob/master/README.md

8.391 s

Jetson TX1



67%

25%

Task specific: ground plane assumption

2.8 s

Remove Overhead

2.103 s

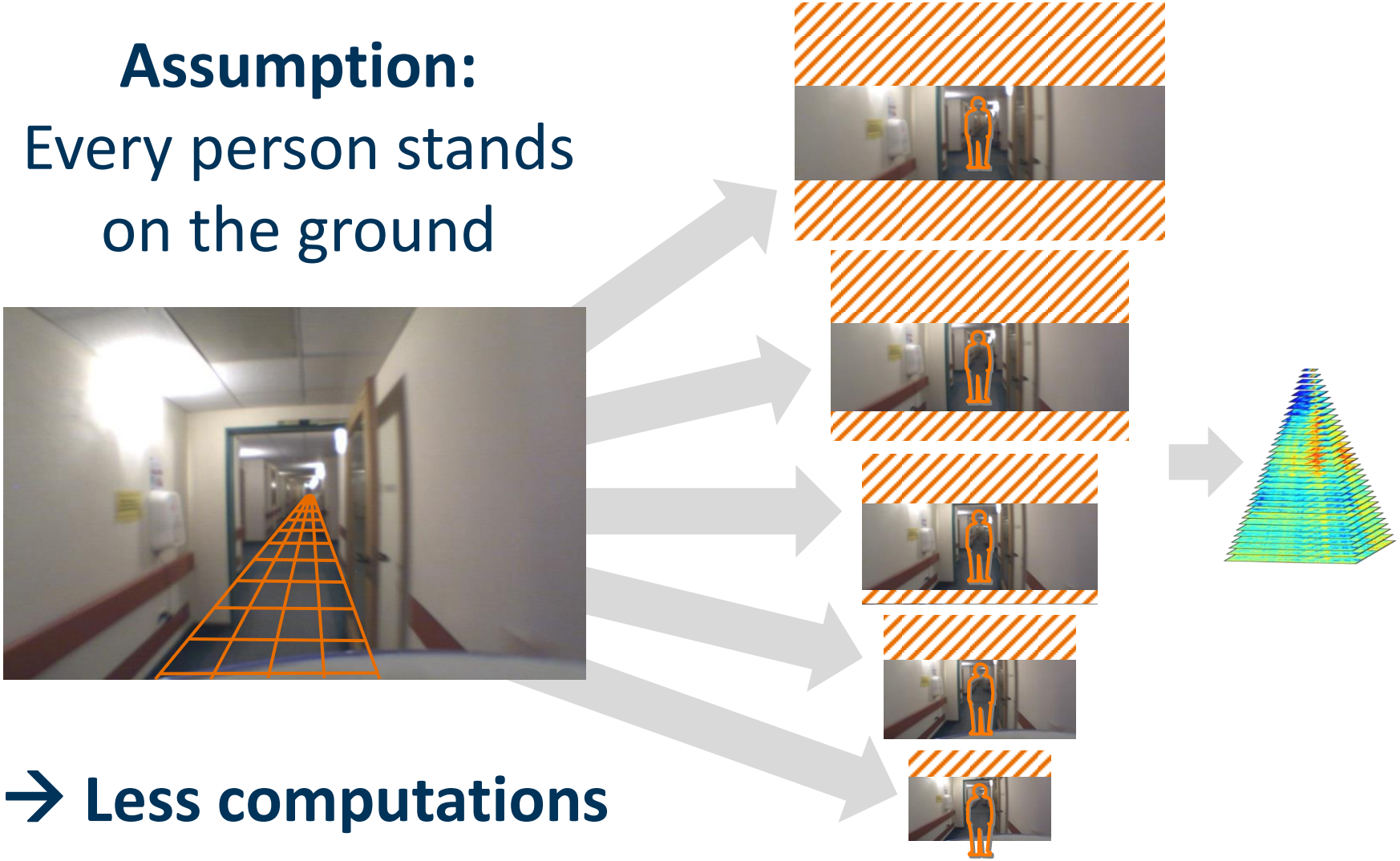
Float 16

0.462 s

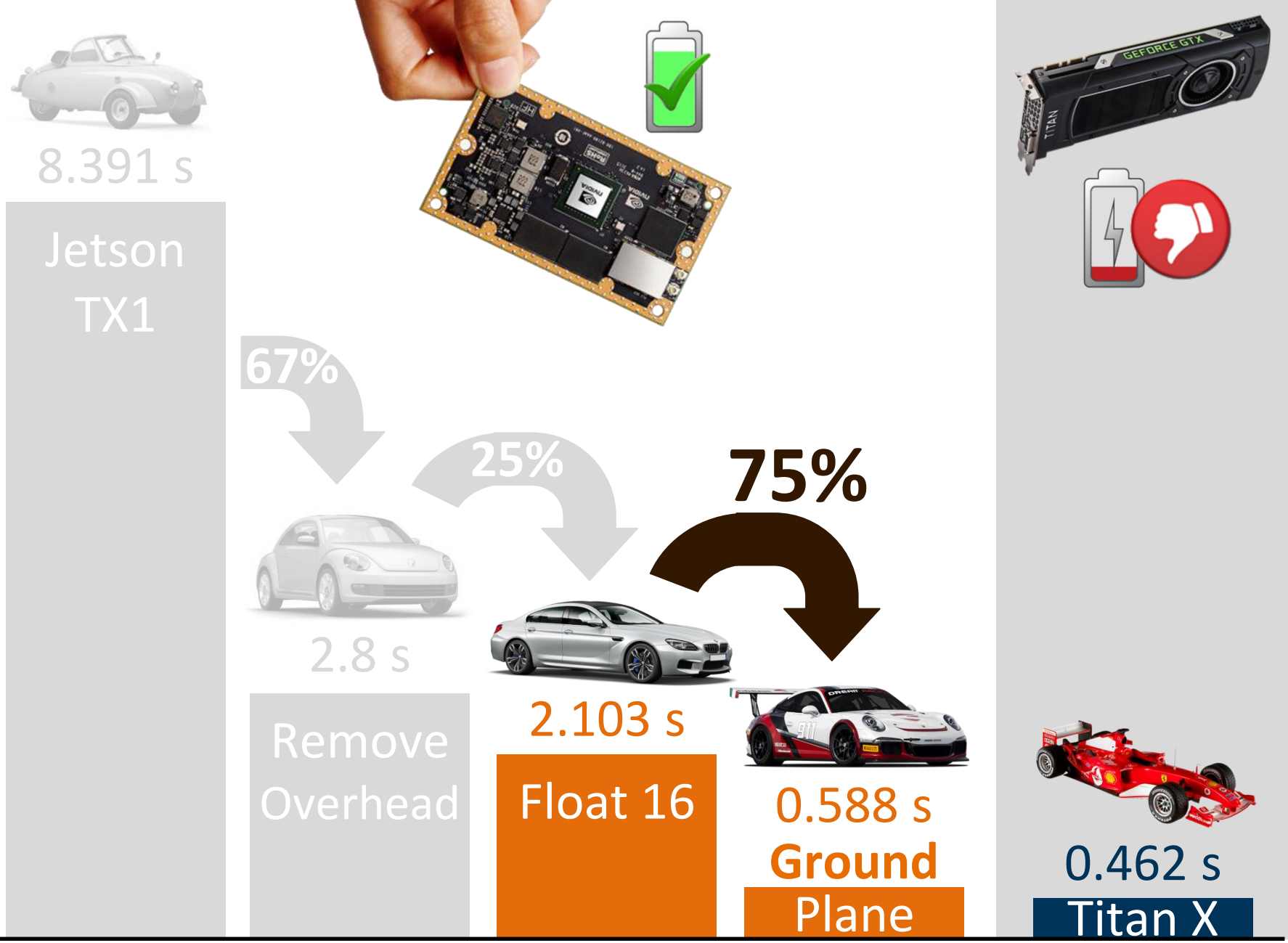
Titan X

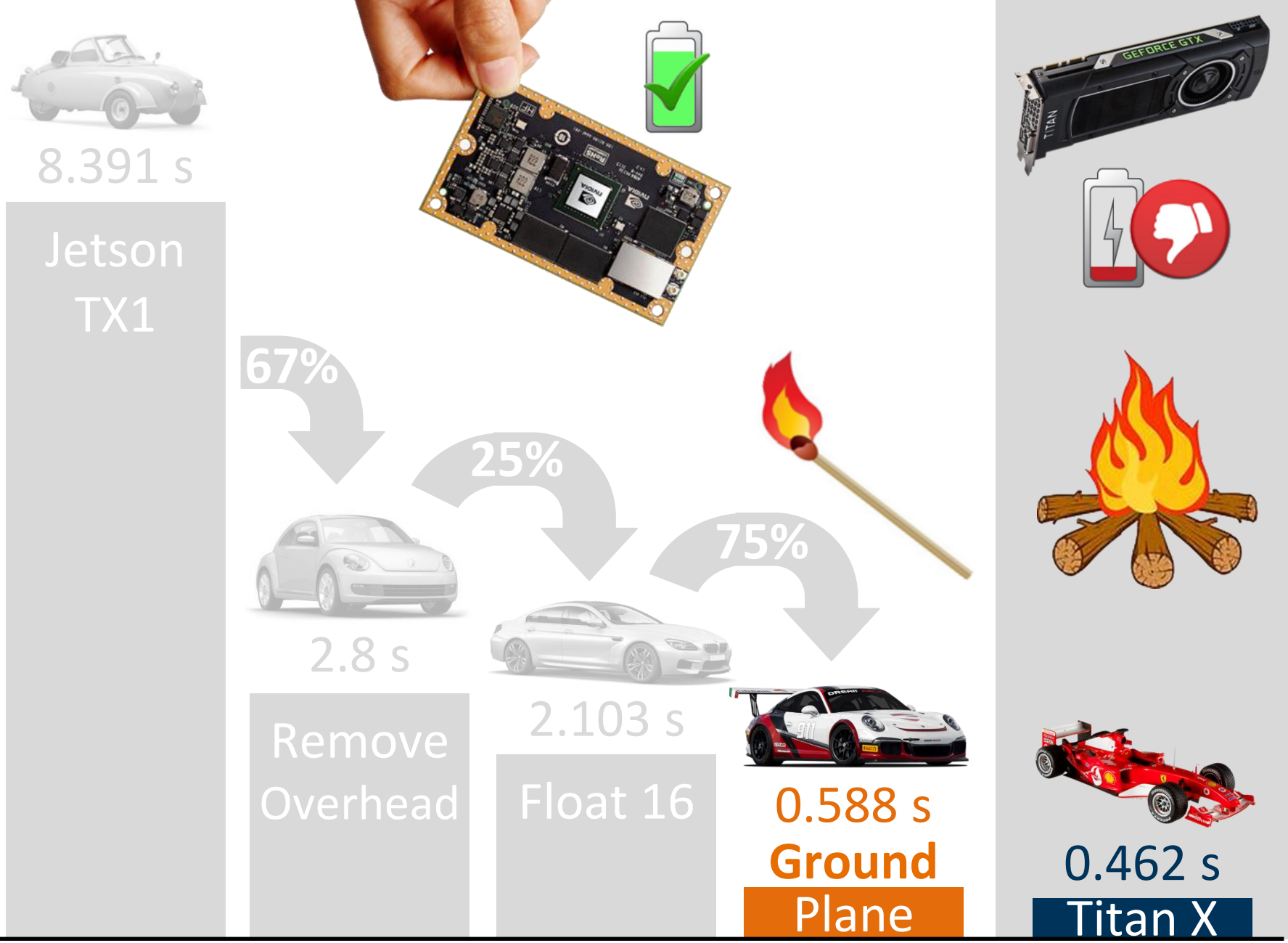
Ground plane assumption

Assumption:
Every person stands on the ground

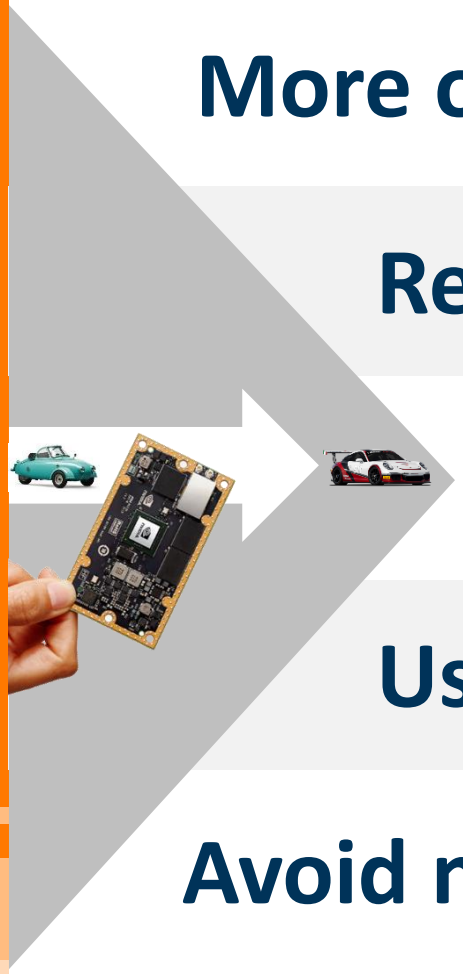


→ **Less computations**

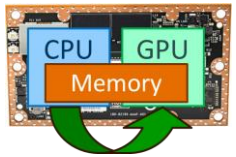

8.391 s
Jetson TX1
67%
2.8 s
Remove Overhead
25%
2.103 s
Float 16
75%
0.588 s
Ground Plane
0.462 s
Titan X


8.391 s
Jetson TX1
67%
25%
75%
2.8 s
Remove Overhead
2.103 s
Float 16
0.588 s
Ground Plane
0.462 s
Titan X

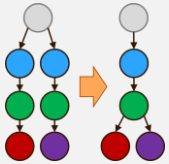
Summarized: What can you do to improve the run time?



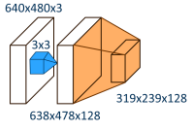
More operations on GPU



Remove redundancies



Specify exact tensor shapes



Use float 16 precision

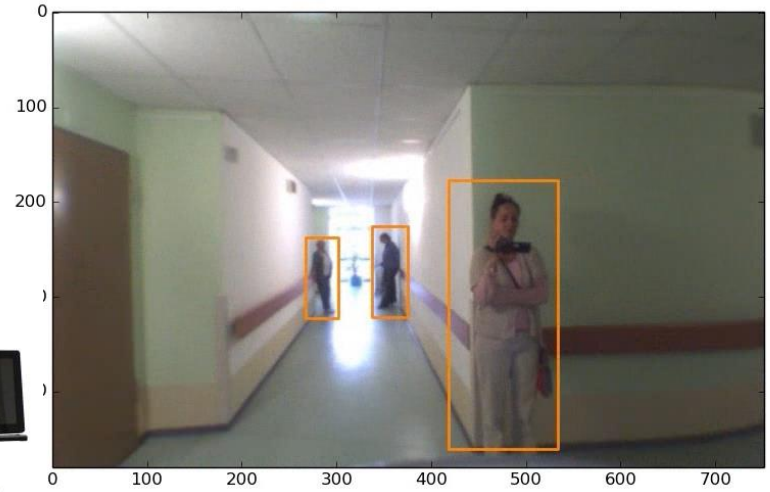
Avoid needless computations

Performance

How does the detector perform on a robot?

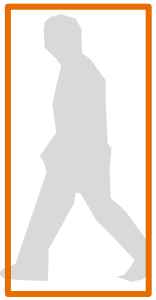
Performance: video example

Front camera



Back camera

Detections

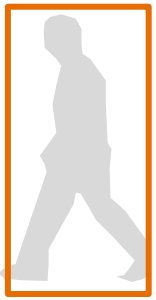


Performance: video example

Front camera

Back camera

Detections

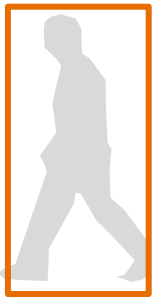


Performance: video example

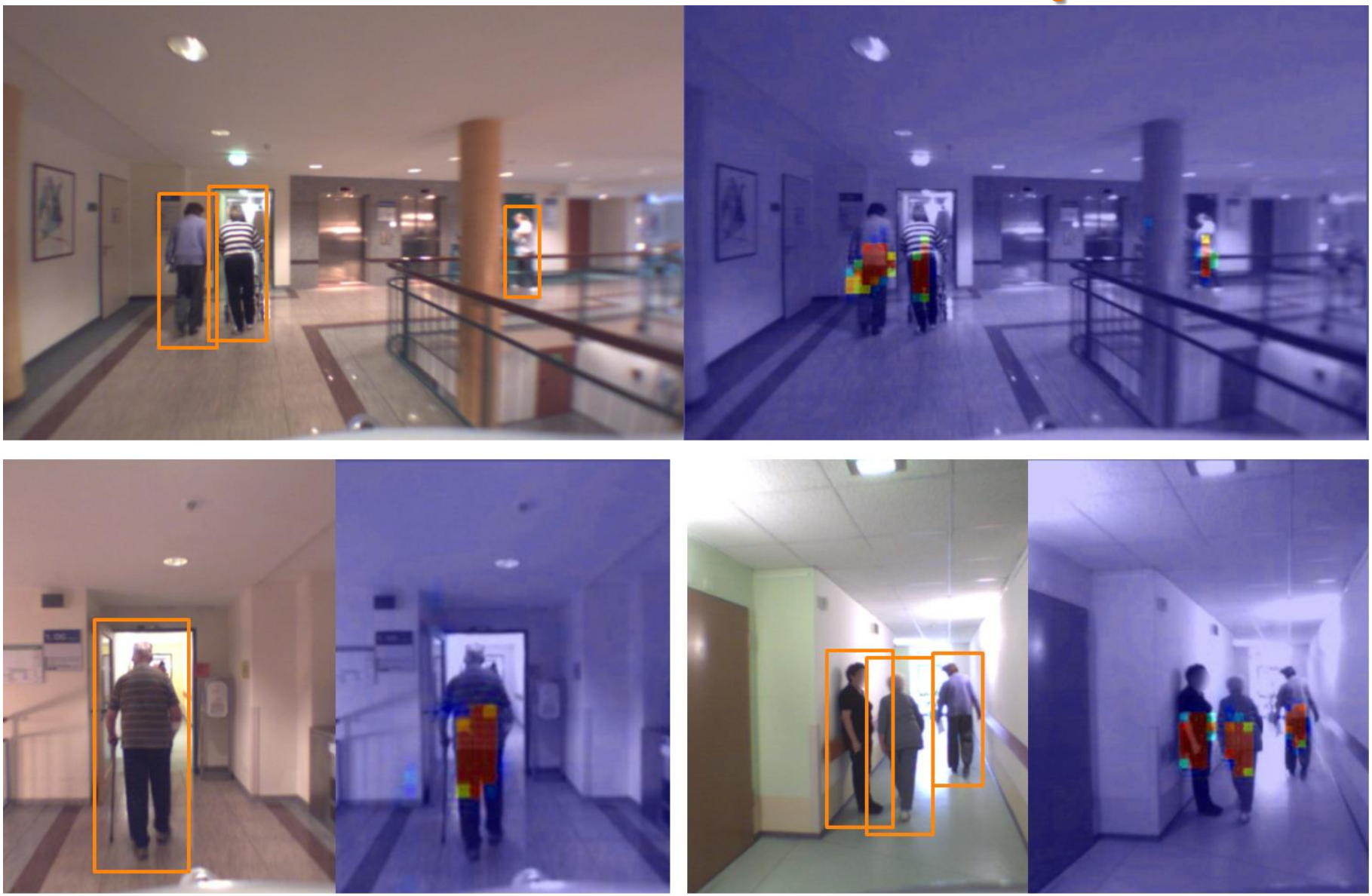
Front camera

Back camera

Detections



Performance: visual examples



Performance in numbers

(Robotic dataset)

	Miss rate (@0.1 fppi)	Speed (fps)
YOLO	38.6%	12
Faster R-CNN	32.1%	2-3
Part-based HOG	27.3%	1.5-2
Proposed	11.5%	1.6

Conclusion

Conclusion

Speeding up Deep Neural Networks on the Jetson TX1

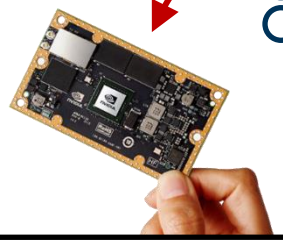
67% Process all on GPU, optimize graph

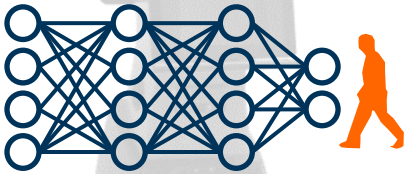
25% Use float16 instead of float32

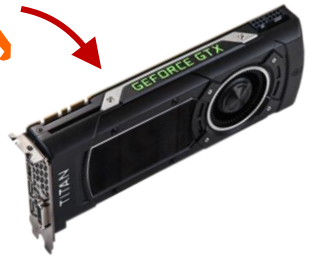
75% Use assumptions for less computations

0.588 s

Jetson TX1







0.462 s

Titan X