

Unity Labs // Intel Labs

Machine Learning in Real-time

Unity Labs - Barracuda team

Florent Guinier

Unity Labs

Mission: Explore how real-time 3D (RT3D) will be created and played in the future.

Area of interest:

- RT3D authoring
- AI, deep learning
- Computer Visualization
- XR
- Storytelling

Barracuda

Lightweight inference library

Cross platform

CPU and GPU

Delivered as Unity package

Source is available on [github](#)

Why do we do it?

We believe the ML and RT3D communities are extremely powerful together!

Agenda

- Real-time ML/DL inference use cases for RT3D (9 mins)
- Barracuda pipeline (5 mins)
- Optimizations (15 mins)
- Practical example (8 mins)

Bonus slides: ONNX & ONNX Runtime

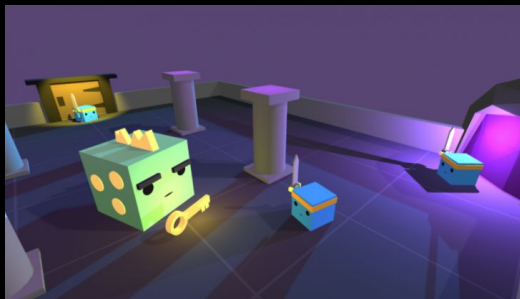
Real-time inference for RT3D

- Medium computational intensity
 - CPU
 - Complex architecture
 - Small input size
- High computational intensity
 - Better suited GPU
 - Convolution
 - Large input size

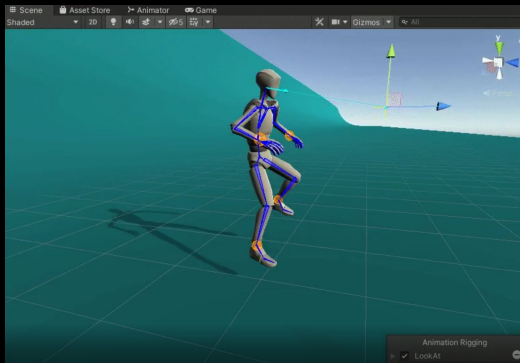
Real-time inference for RT3D

- **Medium computational intensity**

- Decision making / agent behavior

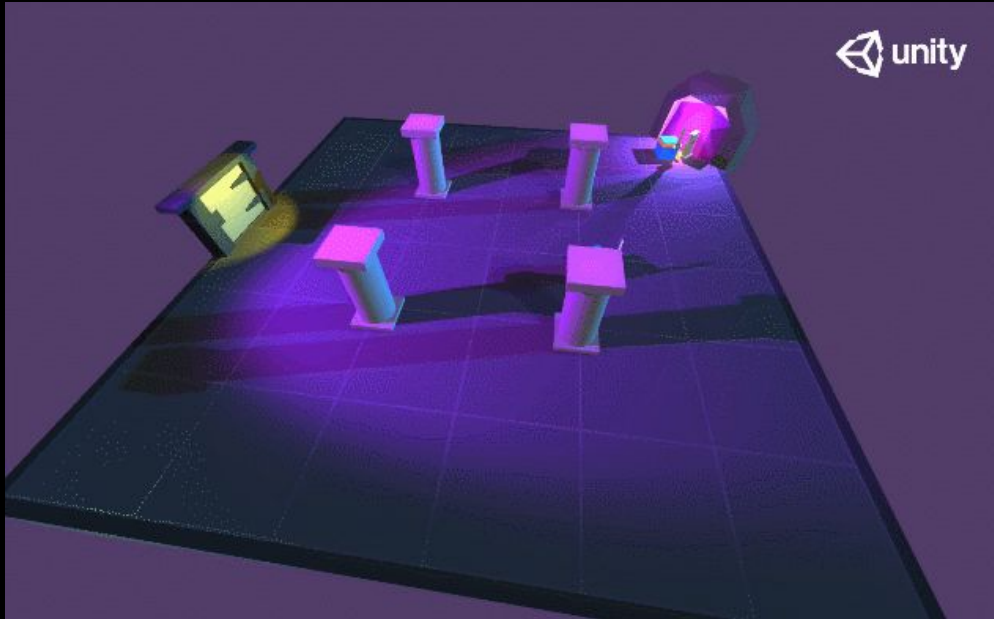


- Animation synthesis



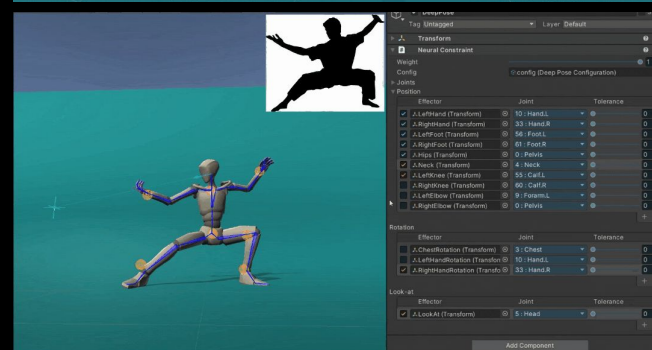
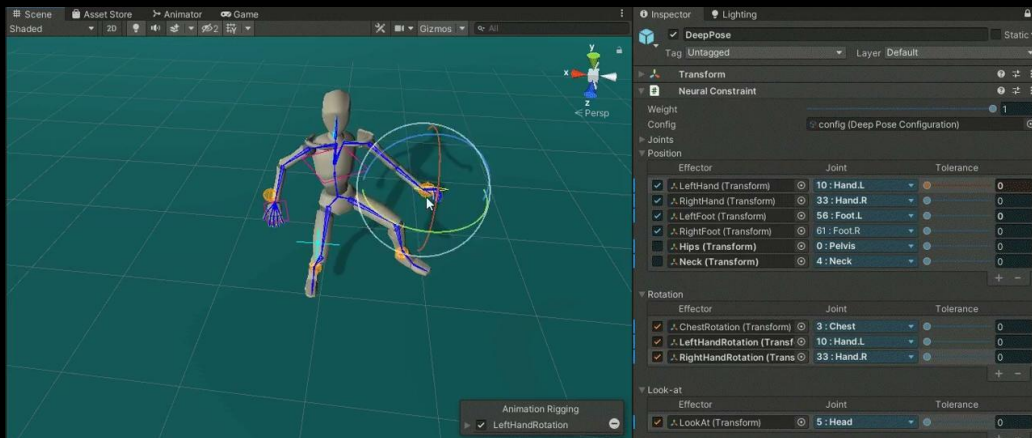
Medium computational intensity

Decision making / agent behavior



Medium computational intensity

Animation authoring



Real-time inference for RT3D

- **High computational intensity**
 - Super resolution
 - Style transfer
 - XR object detection, tracking & segmentation
 - XR pose estimation

High computational intensity

Super resolution



High computational intensity

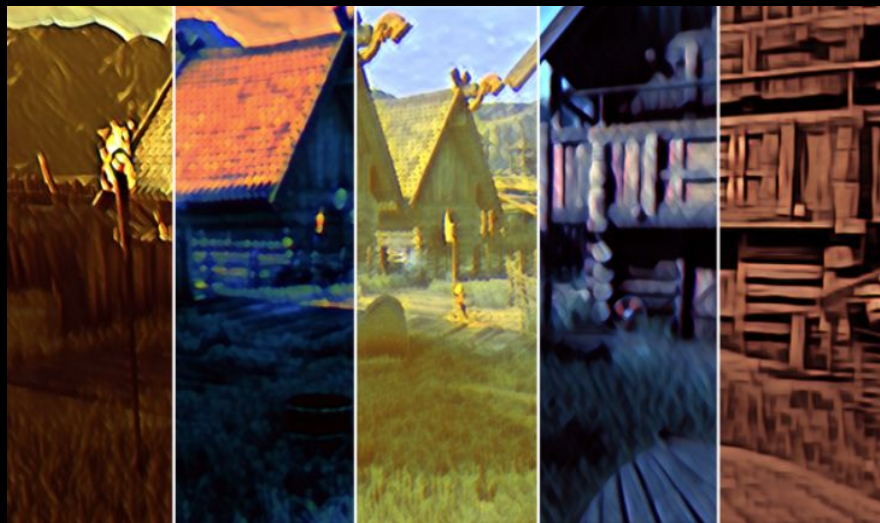
Denoising

32 samples per pixels - Raw



High computational intensity

Style transfer



High computational intensity

XR tracking & segmentation



High computational intensity

XR object detection / tracking



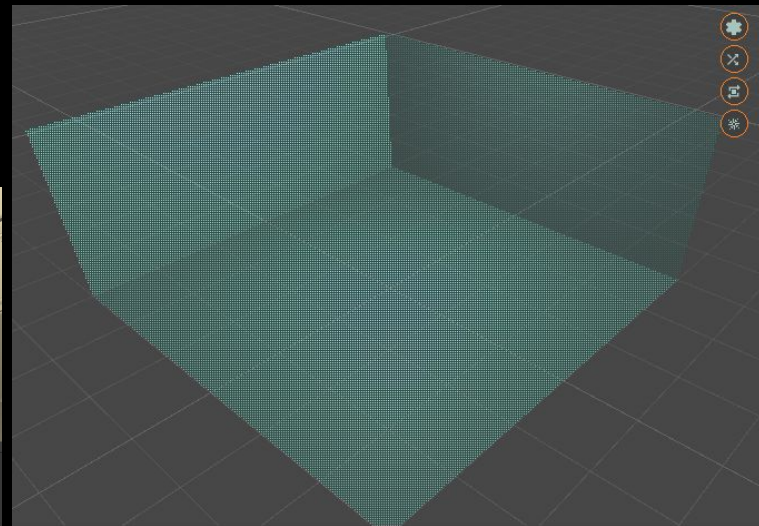
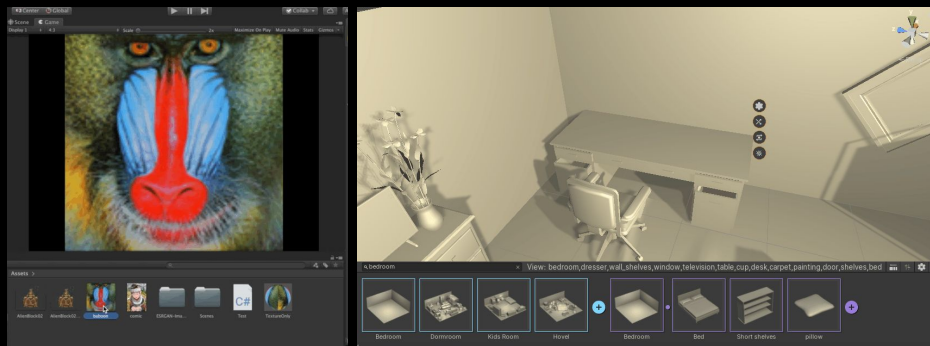
High computational intensity

XR object tracking



At loading or authoring time

- Texture upscaling/generation
- Baked lighting denoising
- Smart authoring
- And much more!

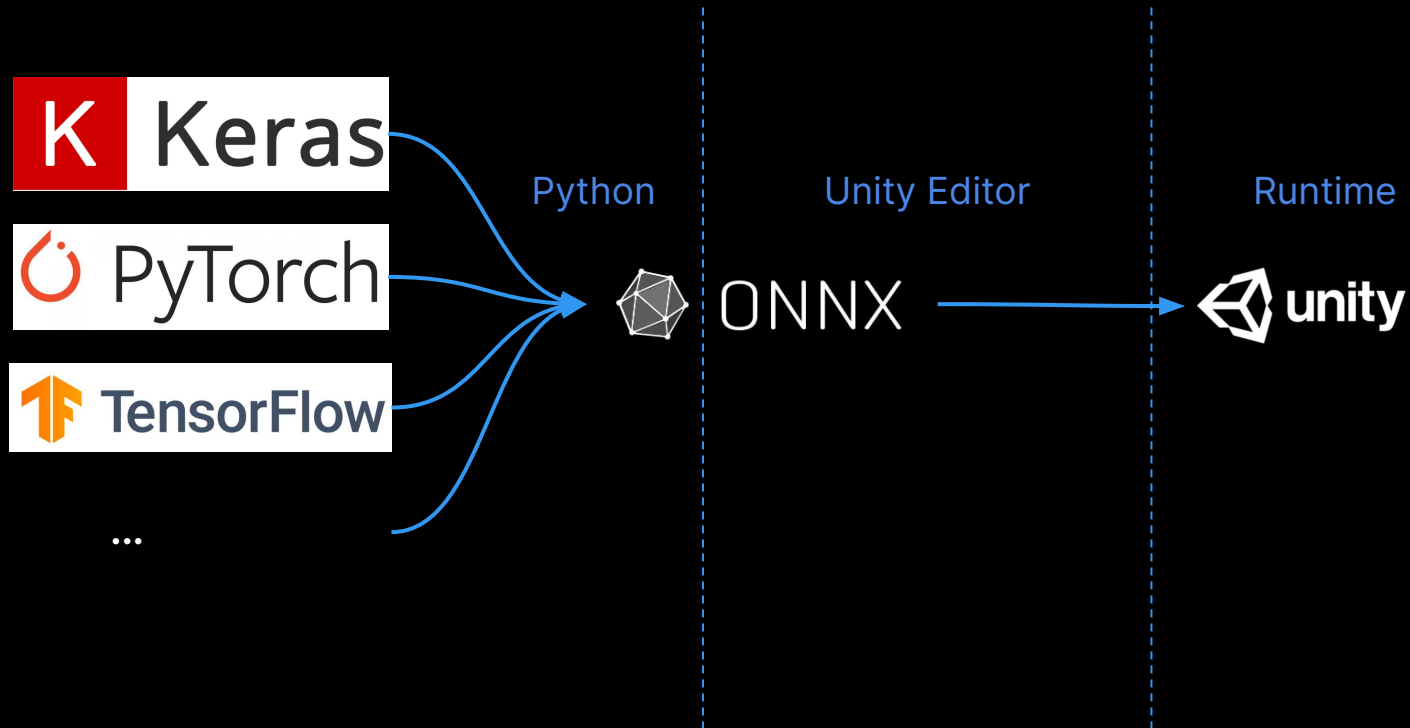


At loading or authoring time

Terrain Authoring



Barracuda pipeline



Barracuda pipeline

Python

Keras

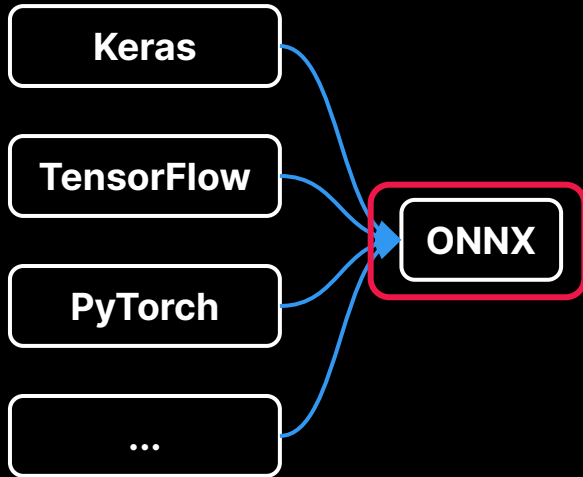
TensorFlow

PyTorch

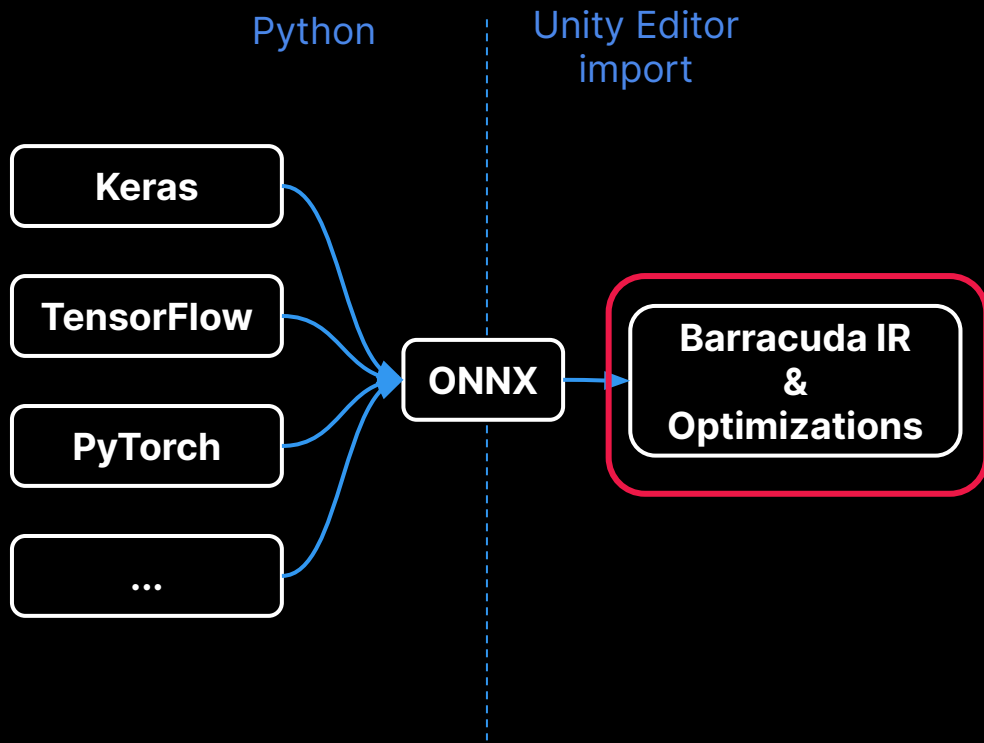
...

Barracuda pipeline

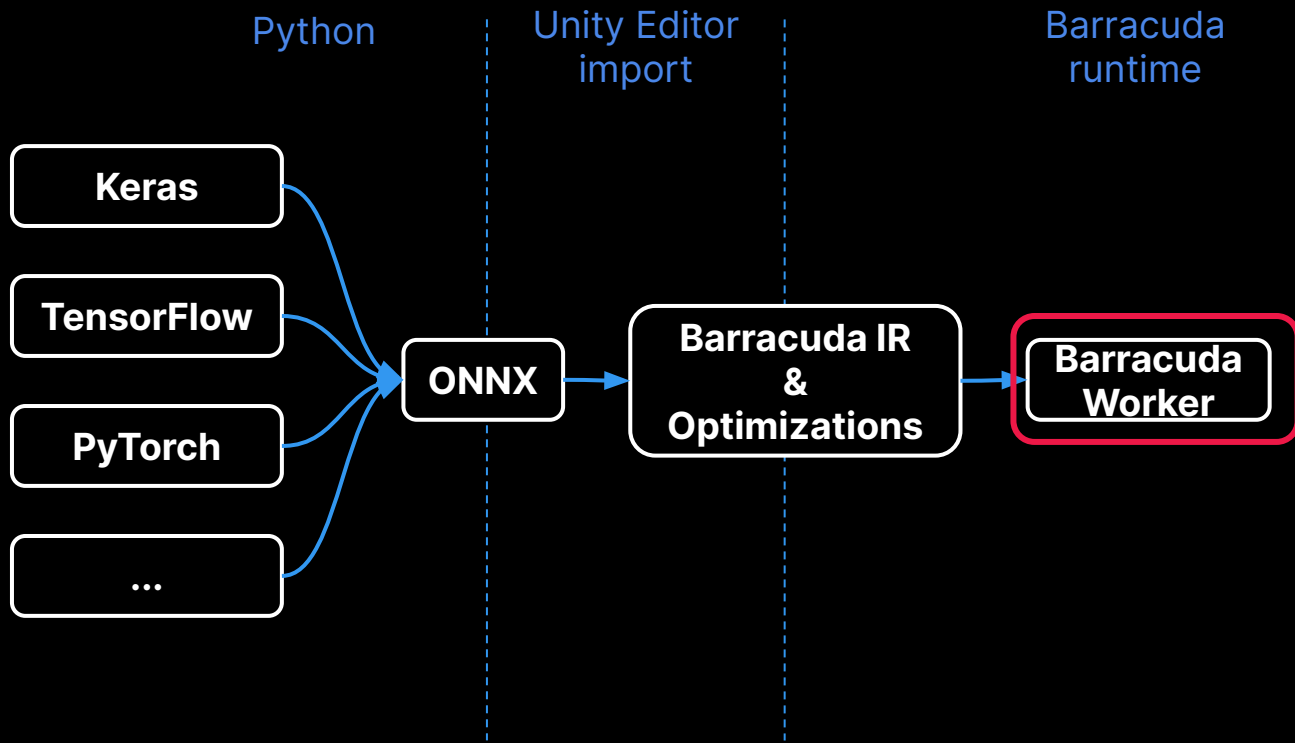
Python



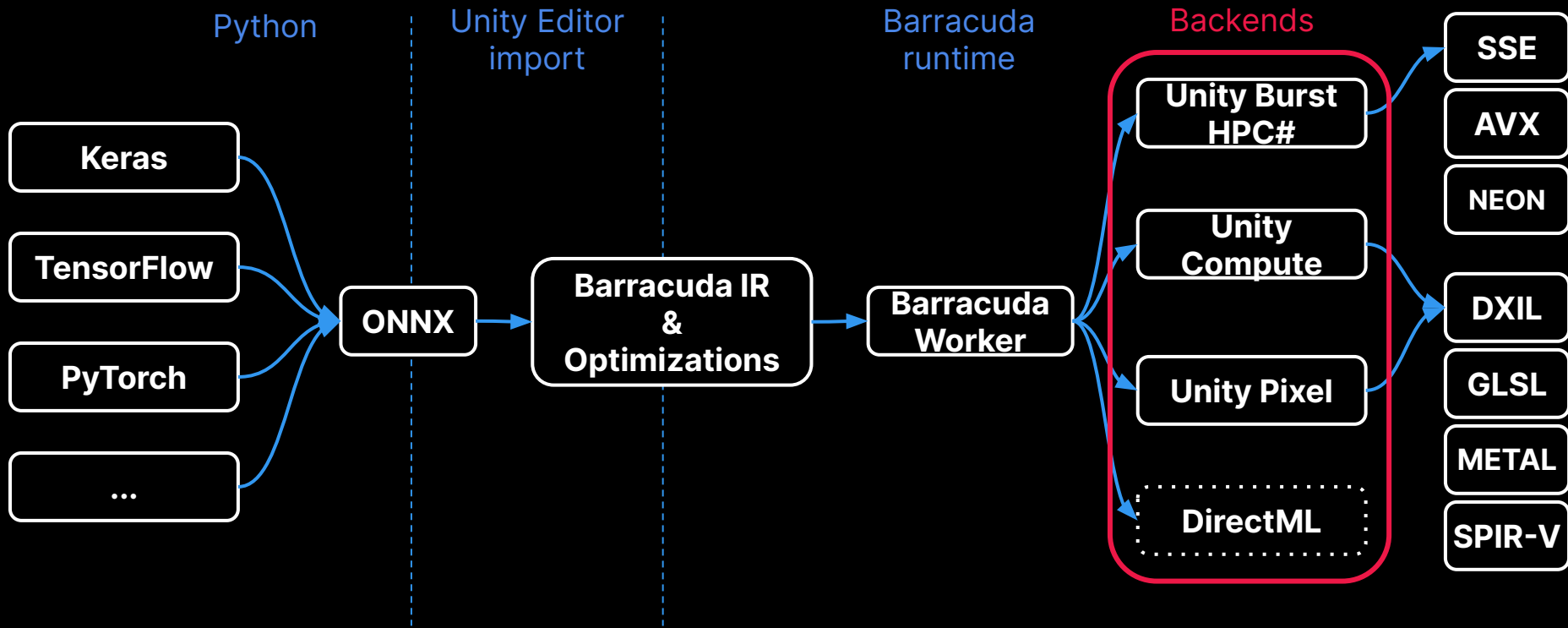
Barracuda pipeline



Barracuda pipeline



Barracuda pipeline

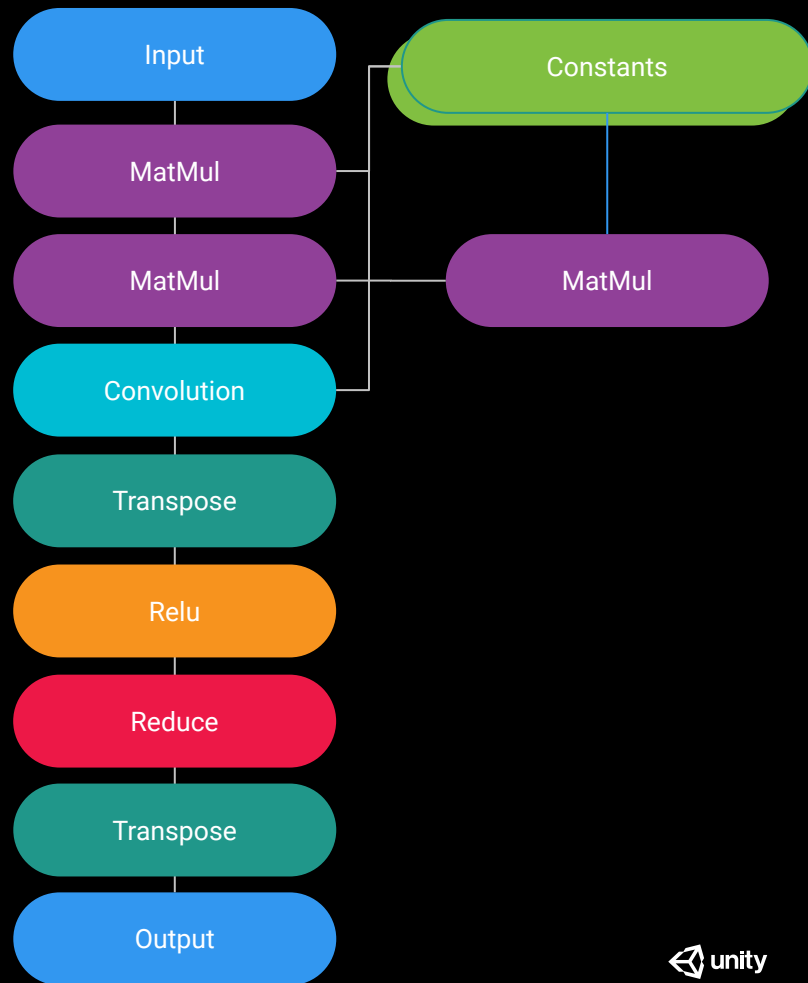


Optimizations

- Graph simplification/reordering
import time, backend agnostic
- Subgraph kernel/layout selection
Import time, backend specific
- Online
runtime, kernels implementation

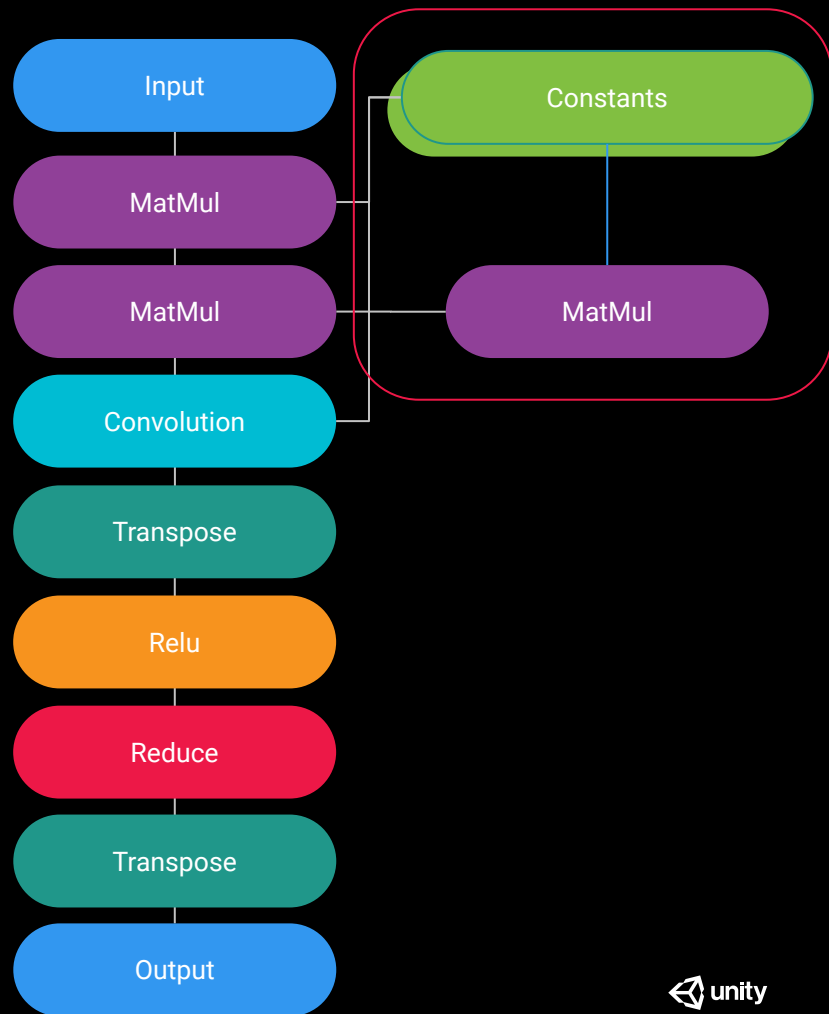
Graph simplification

- Fold constant sub-networks
- Fuse linear operations
- Remove Transpose ops
- Fuse activations



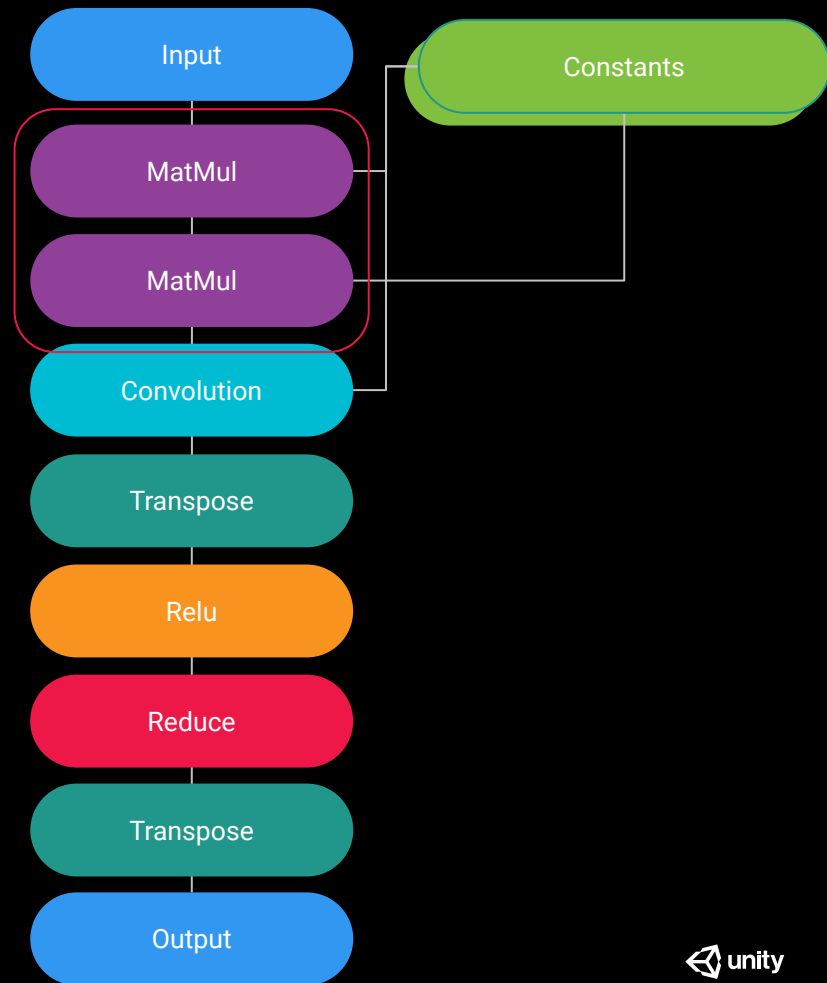
Graph simplification

- Fold constant sub-networks



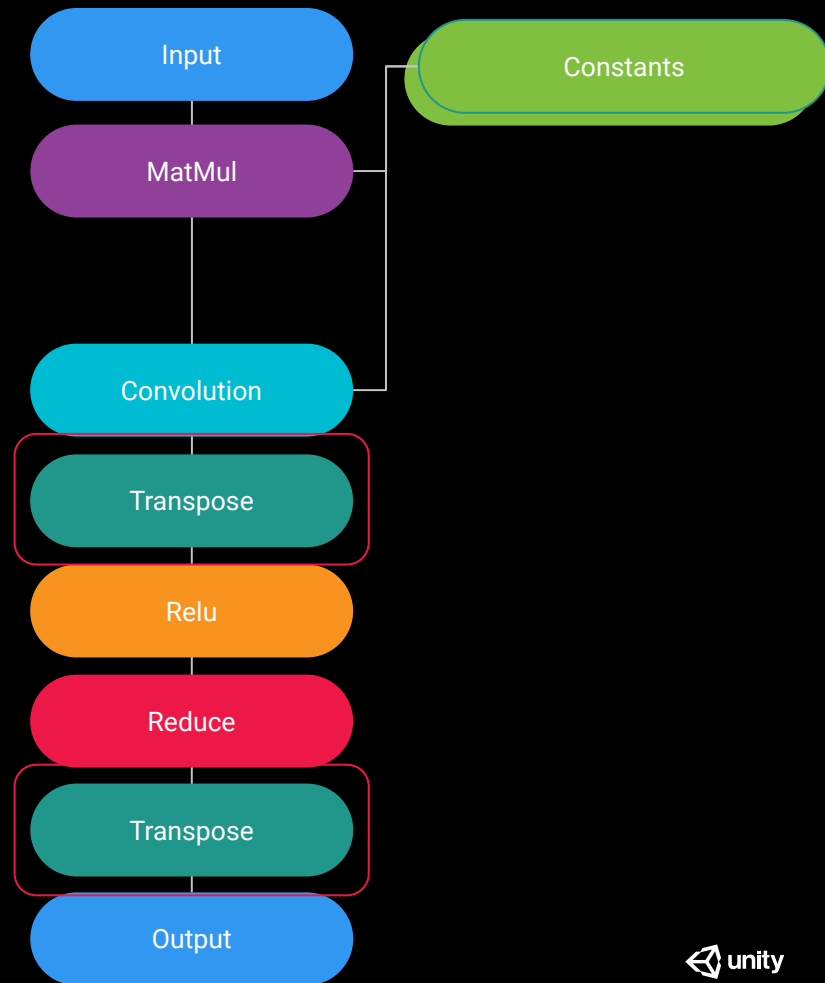
Graph simplification

- Fuse linear operations



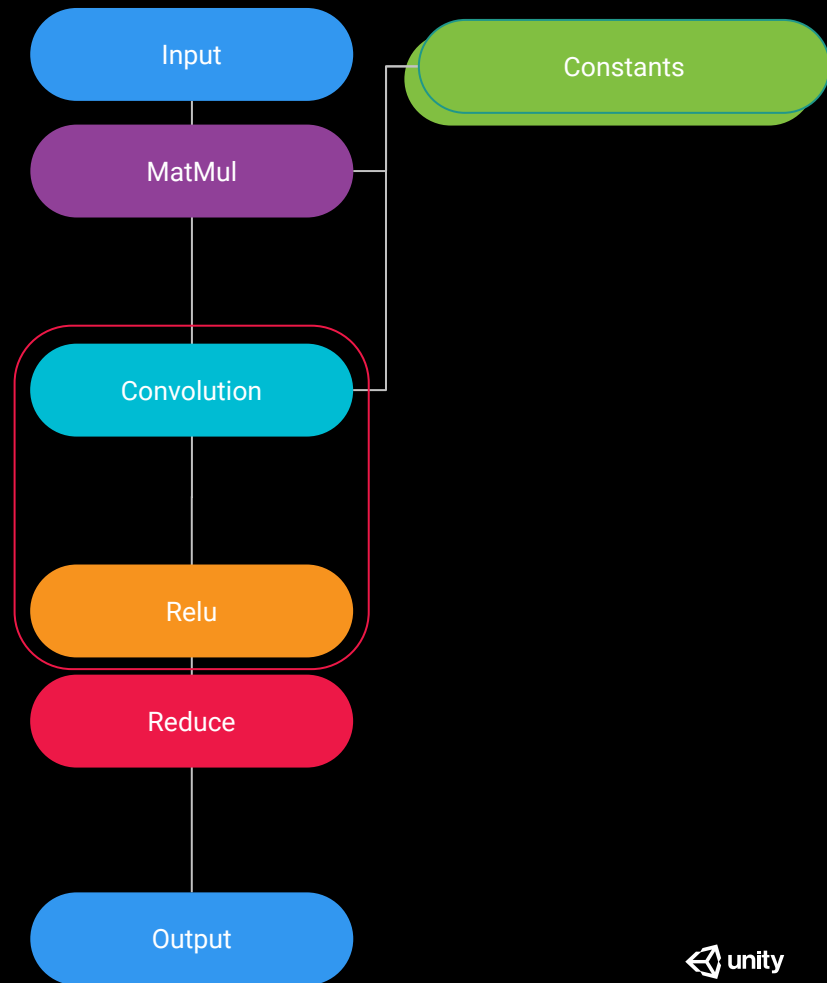
Graph simplification

- Remove Transpose ops



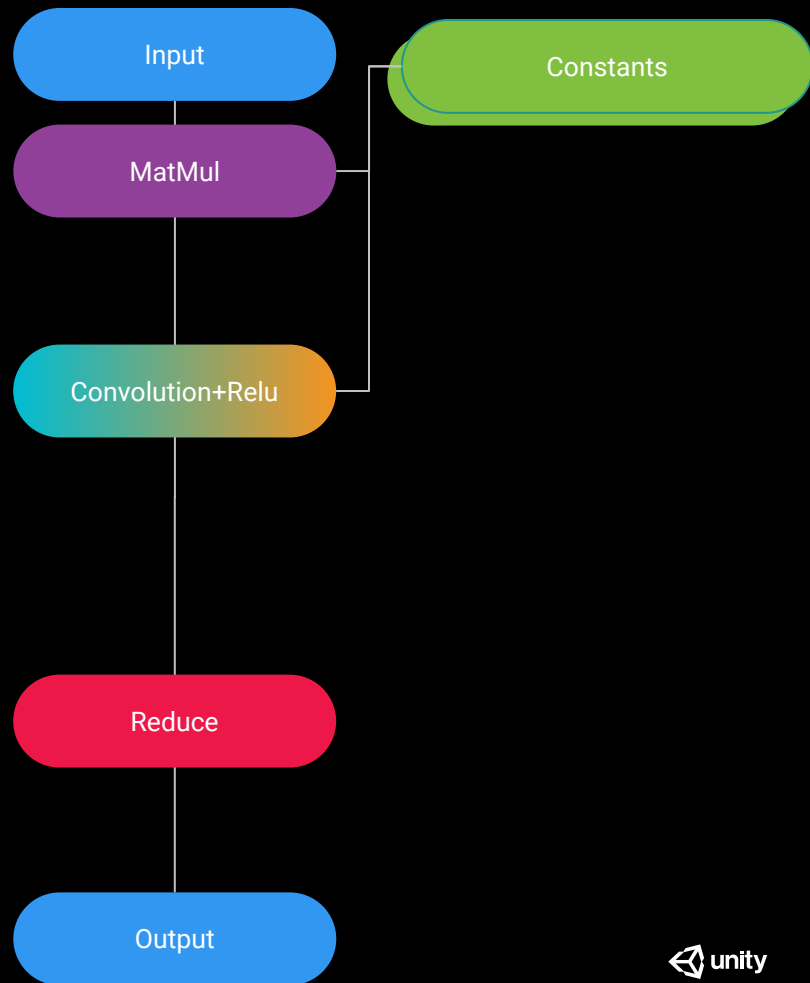
Graph simplification

- Fuse activations

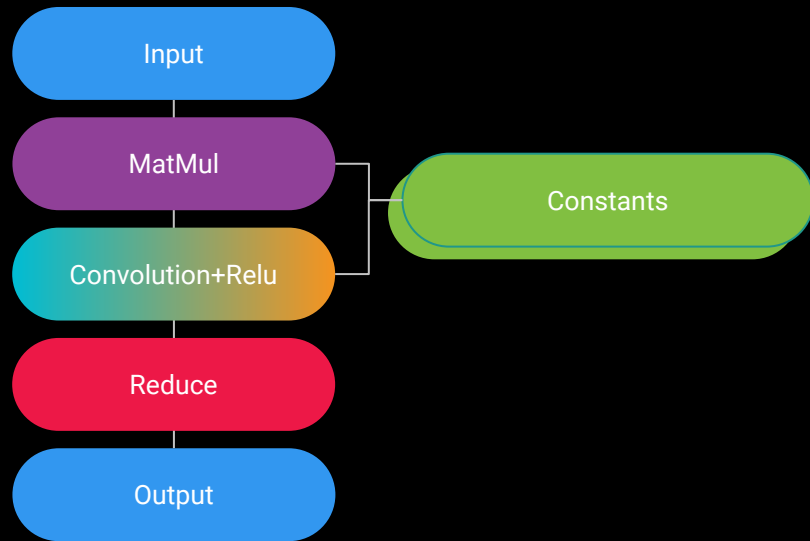


Graph simplification

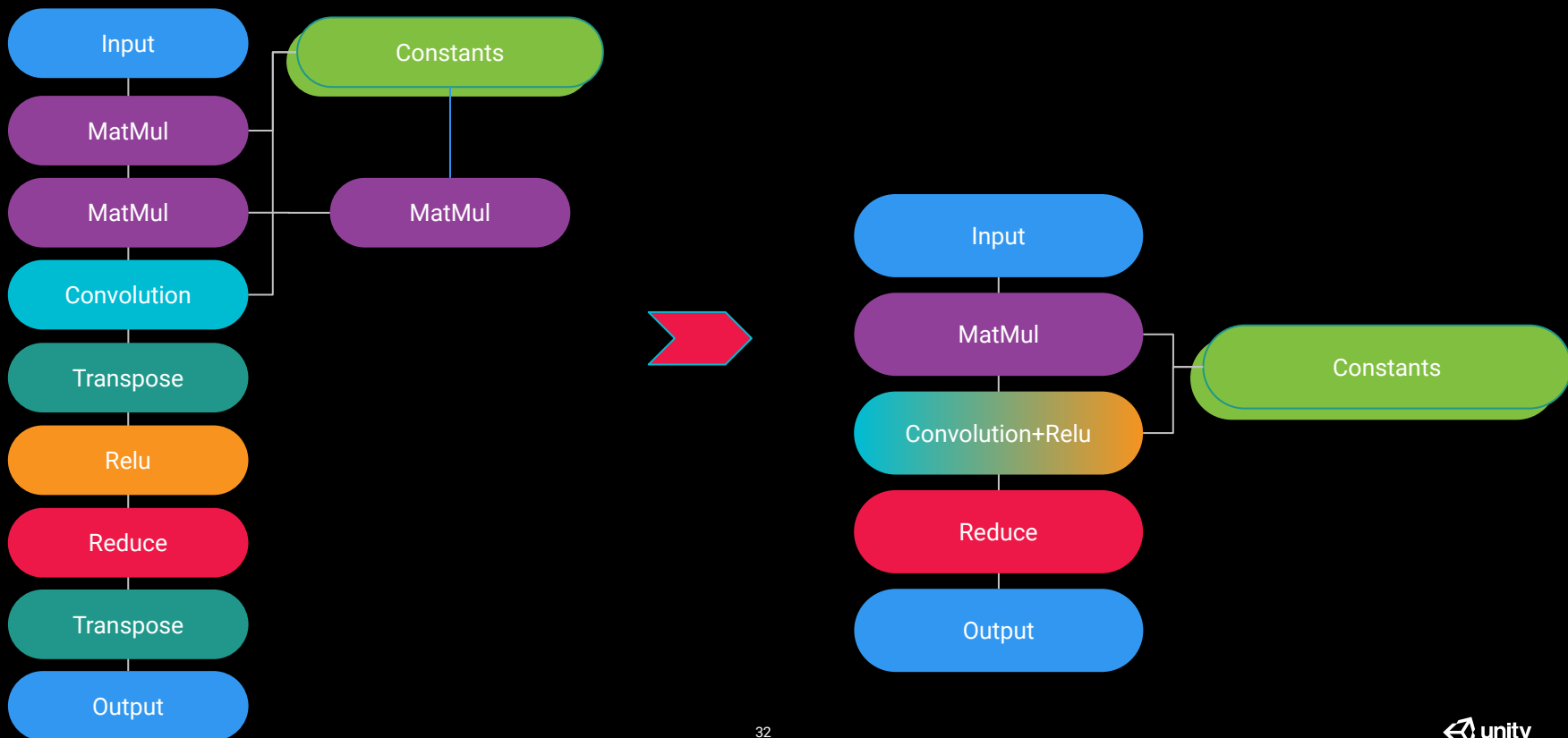
- Fuse activations



Graph simplification

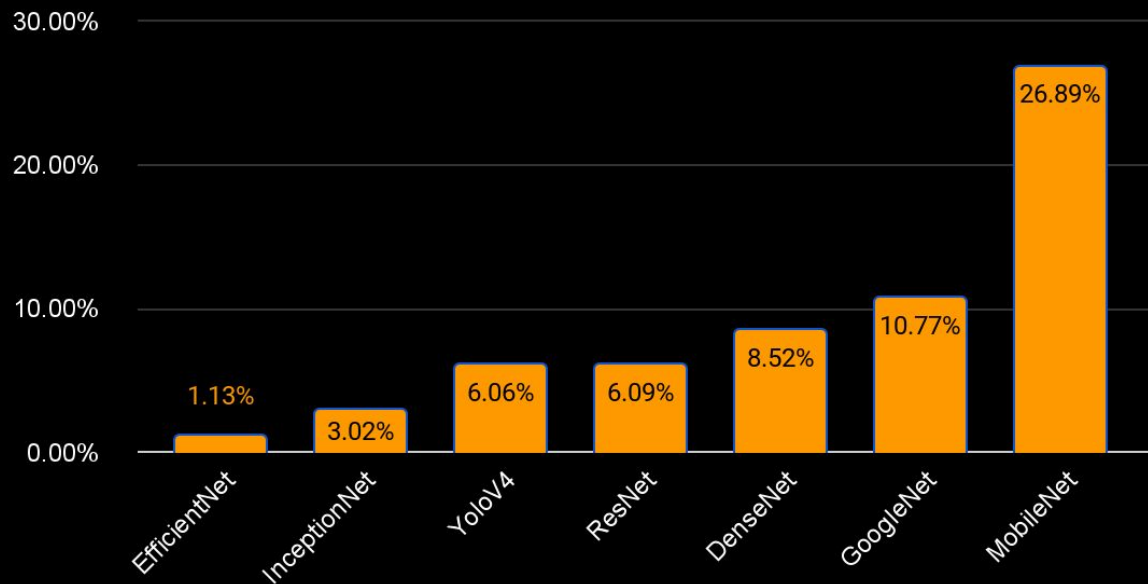


Graph simplification



Graph simplification

Graph Simplification win



Subgraph kernel/layout selection

We can select best the kernels in advance for given hardware and model.

- Reduce scheduling cost
- Allow to prebake temporary data structure

For best performance some kernel require specific memory layout.

- Up to Barracuda 3: internal memory layout can be select for graph.
- Upcoming: automatic subgraph memory layout per backend/hardware.

Optimizations : online

Convolution and Dense/MatMul are often responsible for most of the latency at inference.

- Deserve high amount of optimization love!
- Hardware and backend dependant.

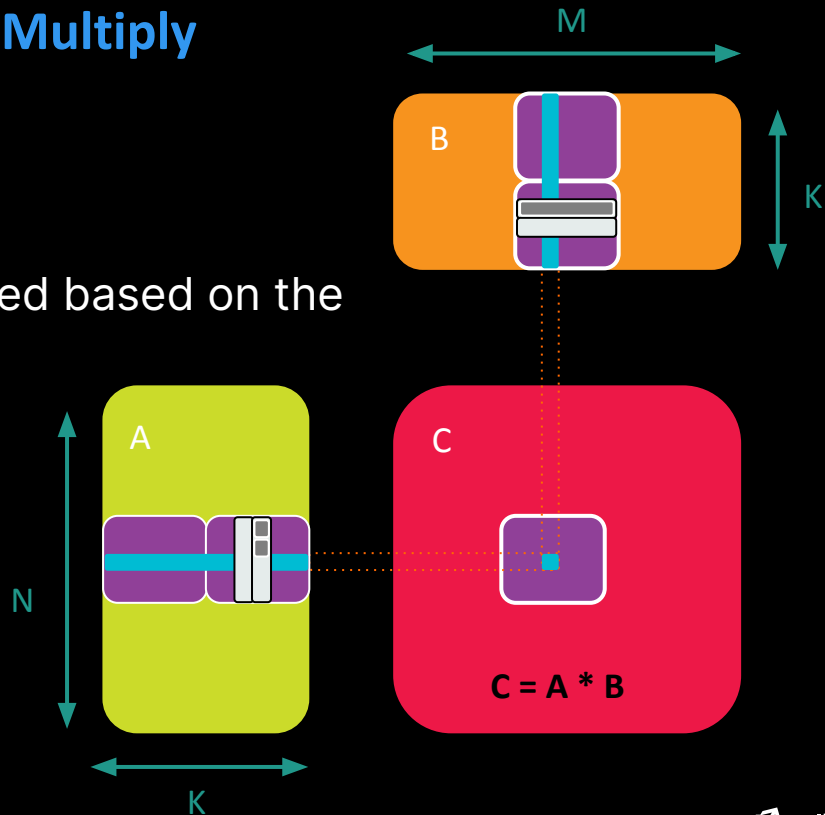


Optimization: online

CPU – Matrix Multiply

Parallel Block Matrix-Multiply

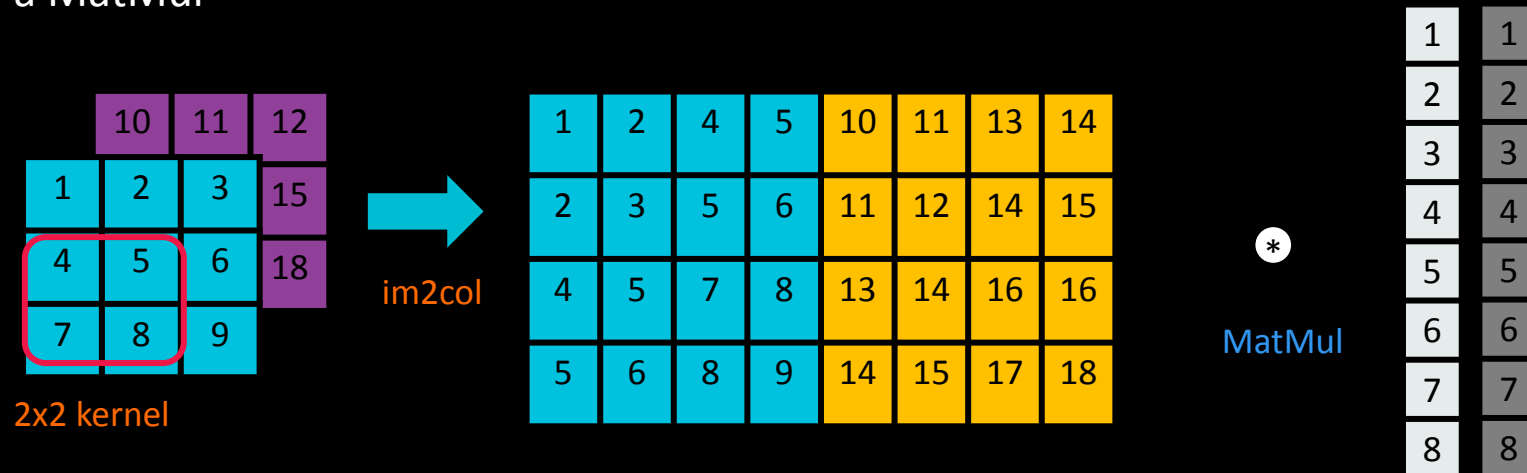
- Block size and inner loop are determined based on the architecture
- Parallelized on the leading dimension



Optimization: online

CPU - Convolution

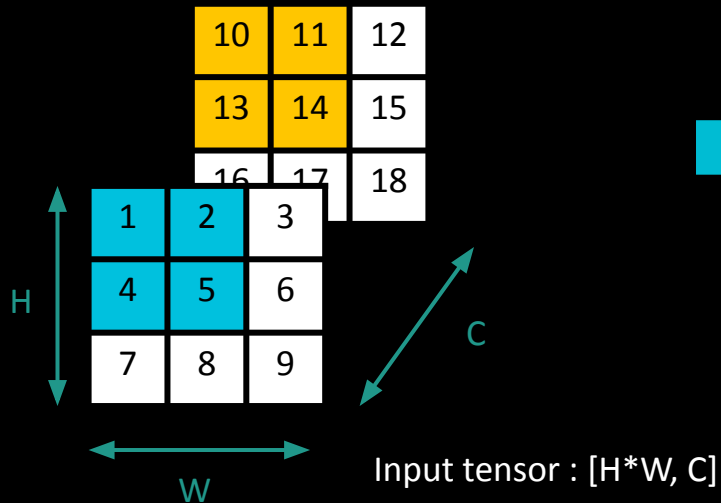
Typically, convolution are implemented via the im2col algorithm + a MatMul



Optimization: online

CPU - Convolution

im2col algorithm:

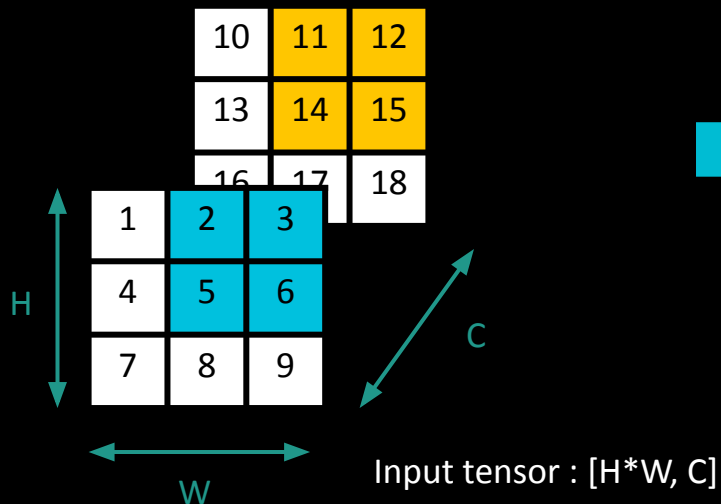


2x2 kernel is slid along the input image.
These values are flattened and concatenated to form the matrix on the right

Optimization: online

CPU - Convolution

im2col algorithm:

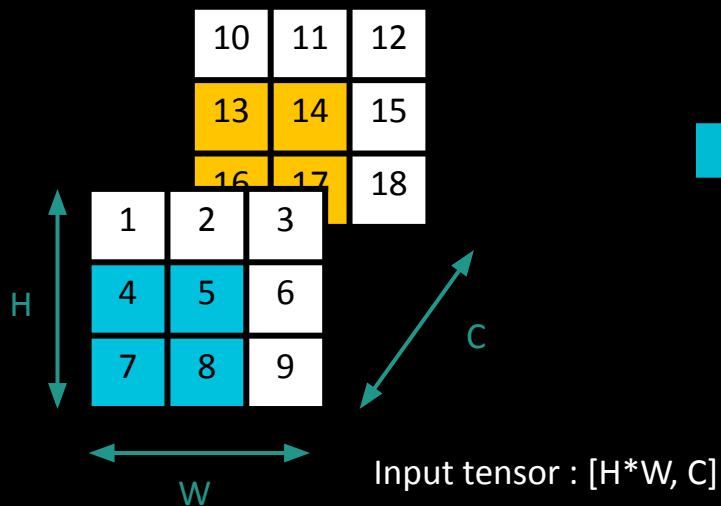


1	2	4	5	10	11	13	14
2	3	5	6	11	12	14	15

Optimization: online

CPU - Convolution

im2col algorithm:

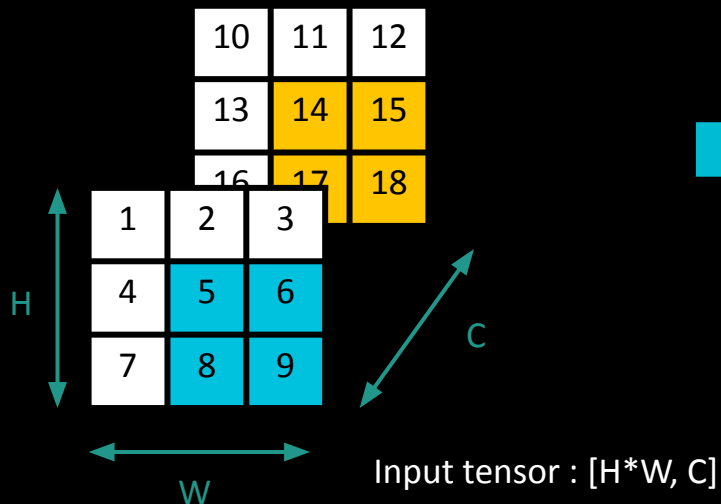


1	2	4	5	10	11	13	14
2	3	5	6	11	12	14	15
4	5	7	8	13	14	16	16

Optimization: online

CPU - Convolution

im2col algorithm:



1	2	4	5	10	11	13	14
2	3	5	6	11	12	14	15
4	5	7	8	13	14	16	16
5	6	8	9	14	15	17	18

Optimization: online

CPU - Convolution

im2col algorithm:

	10	11	12	
1	2	3	15	
4	5	6	18	
7	8	9		



im2col

1	2	4	5	10	11	13	14
2	3	5	6	11	12	14	15
4	5	7	8	13	14	16	16
5	6	8	9	14	15	17	18



$K \times K$



$K \times K * C$

42

MatMul

*

1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8



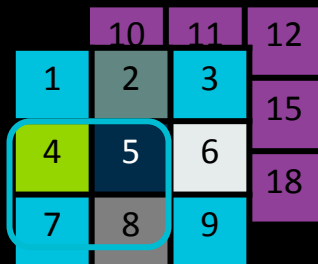
$K \times K * C$



F

Optimization: online

CPU - Convolution



KxK kernel

1	2	4	5	10	11	13	14
2	3	5	6	11	12	14	15
4	5	7	8	13	14	16	16
5	6	8	9	14	15	17	18

Optimizations - online

CPU - Convolutions

- We use a custom variation of the im2col algorithm:
 - Fast
 - **Very good peak memory**

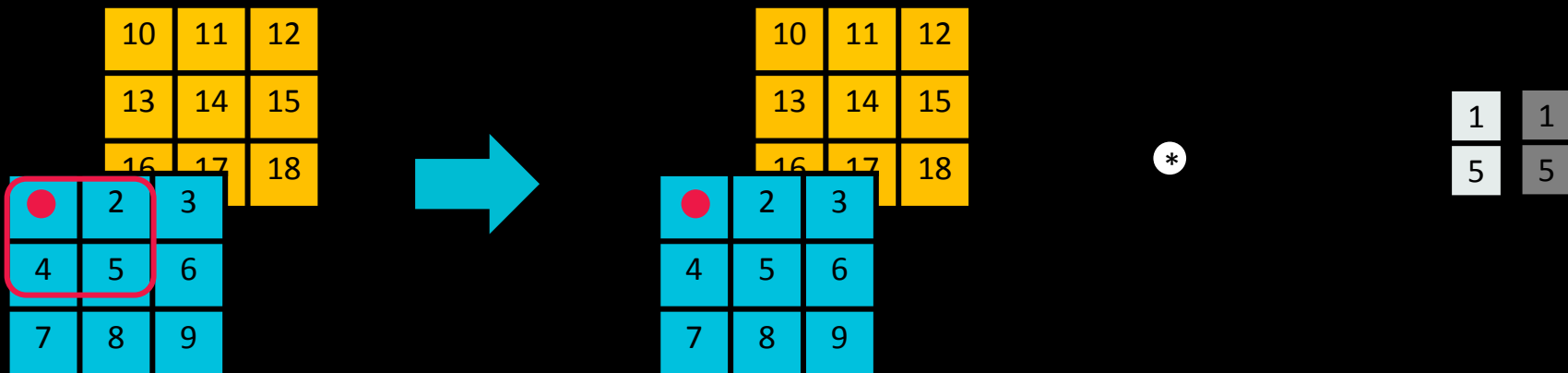
We implement convolution as a $K \times K$ matrix multiplications which reduces memory consumption by $K \times K$ times comparing to standard im2col algorithm.

Our approach trades fraction of performance for significant memory use

Optimization: online

CPU - Convolution

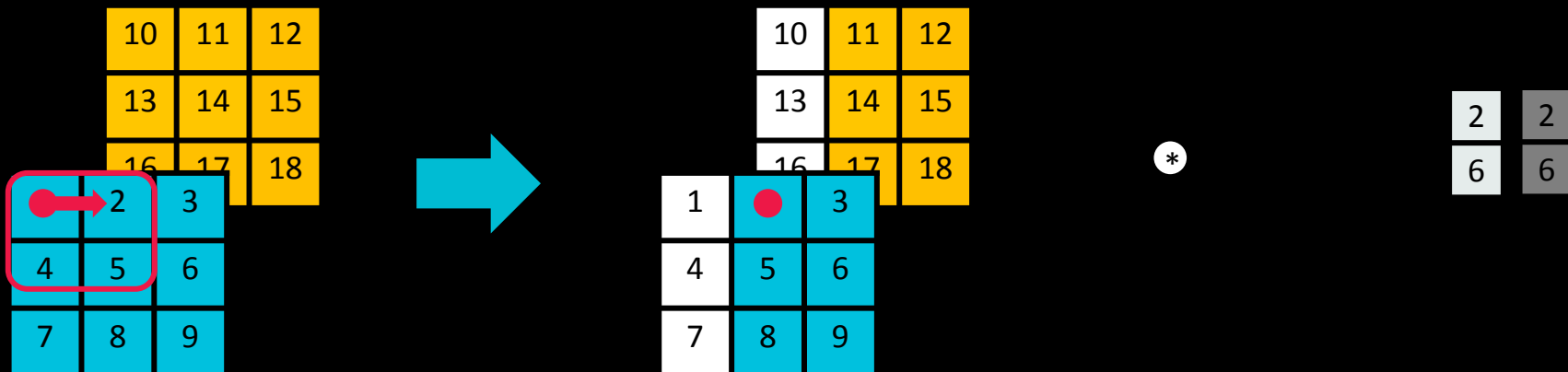
KxK independent matrix multiplication



Optimization: online

CPU - Convolution

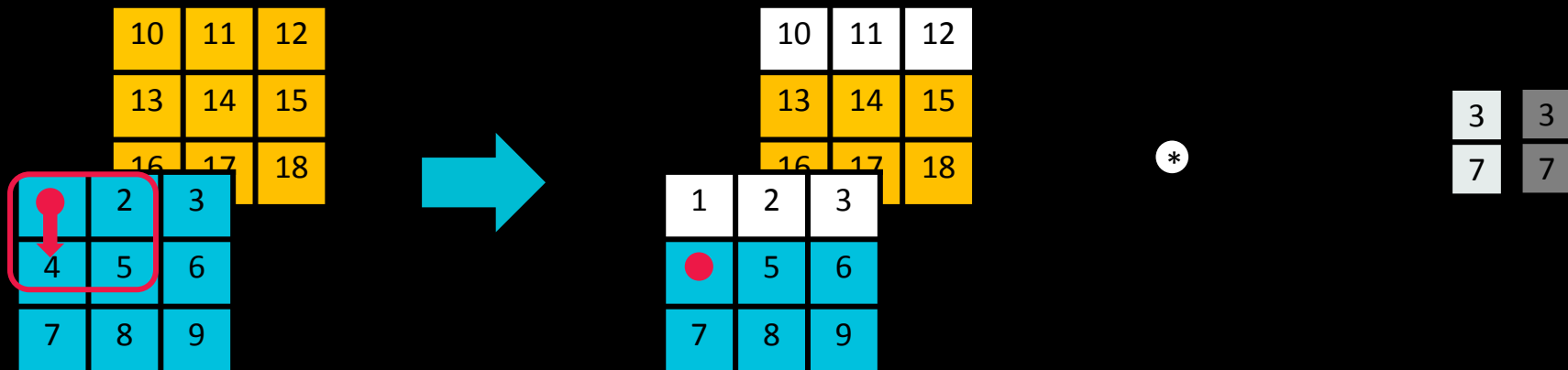
KxK independent matrix multiplication



Optimization: online

CPU - Convolution

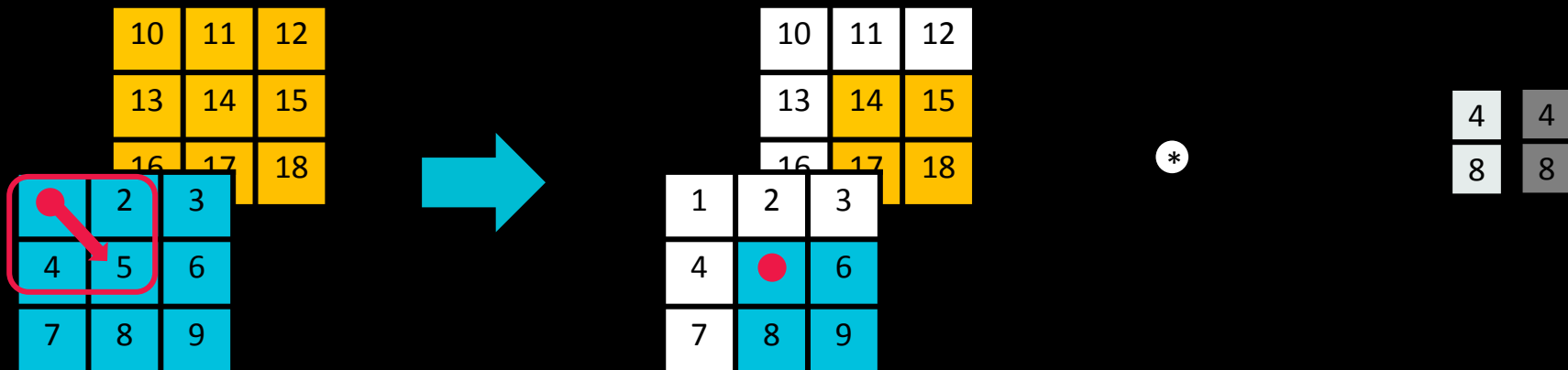
KxK independent matrix multiplication



Optimization: online

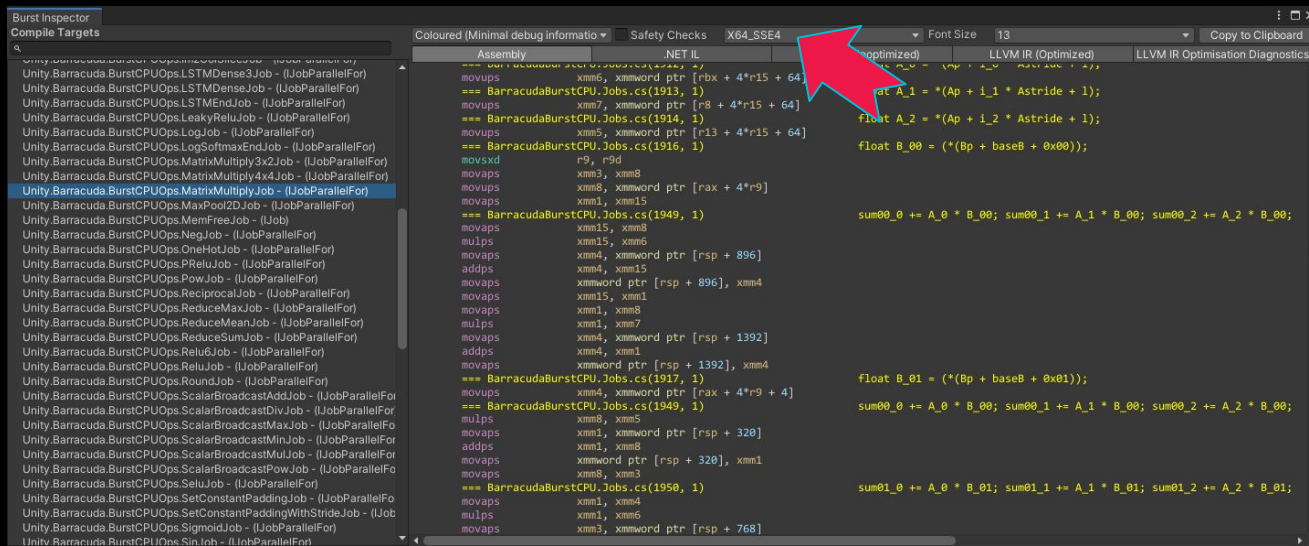
CPU - Convolution

KxK independent matrix multiplication



Optimizations : online

C# is compiled via Burst to highly optimized vectorized assembly code.



The screenshot shows the Burst Inspector interface with the following assembly code:

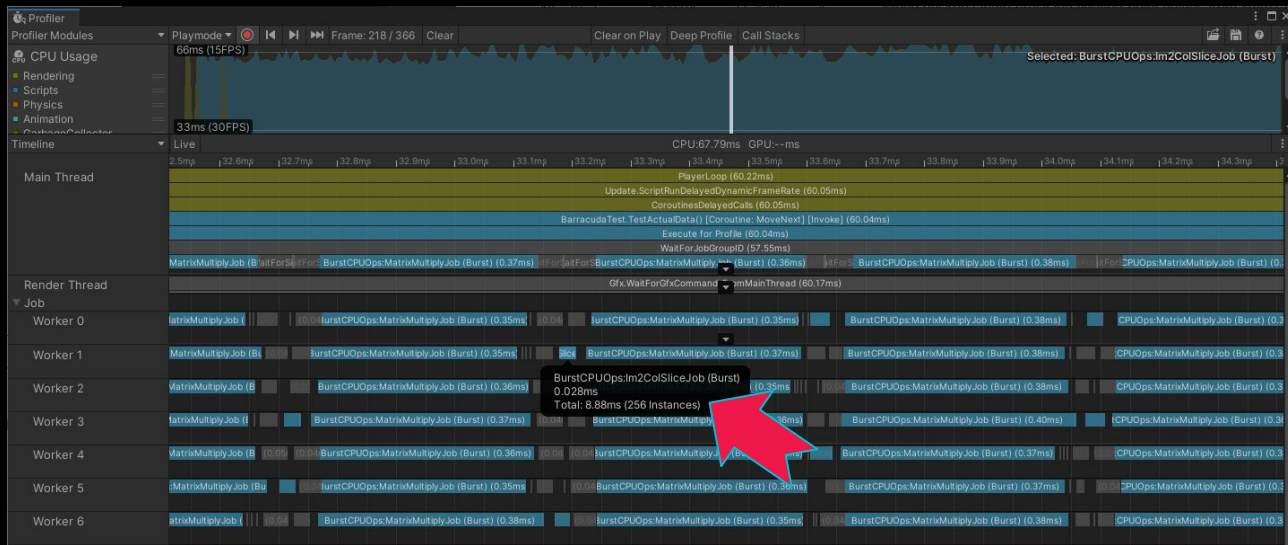
```
Assembly      NET IL
====
movups      xmm6, xmmword ptr [rbx + 4*r15 + 64]
== BarracudaBurstCPU.Jobs.cs(1913, 1)
movups      xmm7, xmmword ptr [r8 + 4*r15 + 64]
== BarracudaBurstCPU.Jobs.cs(1914, 1)
movups      xmm5, xmmword ptr [r13 + 4*r15 + 64]
== BarracudaBurstCPU.Jobs.cs(1916, 1)
movsxd     r9, r9d
movaps     xmm3, xmm8
movups     xmm8, xmmword ptr [rax + 4*r9]
movaps     xmm1, xmm15
== BarracudaBurstCPU.Jobs.cs(1949, 1)
movaps     xmm15, xmm8
mulps     xmm15, xmm6
movaps     xmm4, xmmword ptr [rsp + 896]
addps     xmm4, xmm15
movaps     xmmword ptr [rsp + 896], xmm4
movaps     xmm15, xmm1
movaps     xmm1, xmm8
mulps     xmm1, xmm7
movaps     xmm4, xmmword ptr [rsp + 1392]
addps     xmm4, xmm3
movaps     xmmword ptr [rsp + 1392], xmm4
== BarracudaBurstCPU.Jobs.cs(1917, 1)
movups     xmm4, xmmword ptr [rax + 4*r9 + 4]
== BarracudaBurstCPU.Jobs.cs(1949, 1)
mulps     xmm8, xmm5
movaps     xmm1, xmmword ptr [rsp + 328]
addps     xmm1, xmm3
movaps     xmmword ptr [rsp + 328], xmm1
movaps     xmm8, xmm3
== BarracudaBurstCPU.Jobs.cs(1950, 1)
movaps     xmm1, xmm4
mulps     xmm1, xmm6
movaps     xmm3, xmmword ptr [rsp + 768]
```

Corresponding LLVM IR (Optimized) code:

```
float A_1 = *(Ap + i_1 * Stride + 1);
float A_2 = *(Ap + i_2 * Stride + 1);
float B_00 = (*(Bp + baseB + 0x00));
sum00_0 += A_0 * B_00; sum00_1 += A_1 * B_00; sum00_2 += A_2 * B_00;
float B_01 = (*(Bp + baseB + 0x01));
sum00_0 += A_0 * B_01; sum00_1 += A_1 * B_01; sum00_2 += A_2 * B_01;
sum01_0 += A_0 * B_01; sum01_1 += A_1 * B_01; sum01_2 += A_2 * B_01;
```

Optimizations : online

CPU backend is by design heavily threaded (and thus asynchronous)



Optimizations : online

GPU - Convolution

- GPUs have awesome raw power, however they differ greatly:
 - On-chip memory VS DDR (dedicated VS mobile)
 - Scalar register? (dedicated VS mobile)
 - On-chip memory bandwidth VS FLOPS ratio
 - Number of threads to saturate GPU (and/or to hide latency efficiently)
 - ...
- This mean many [implementations](#), all of them carefully crafted for a specific purpose.

Optimizations : online

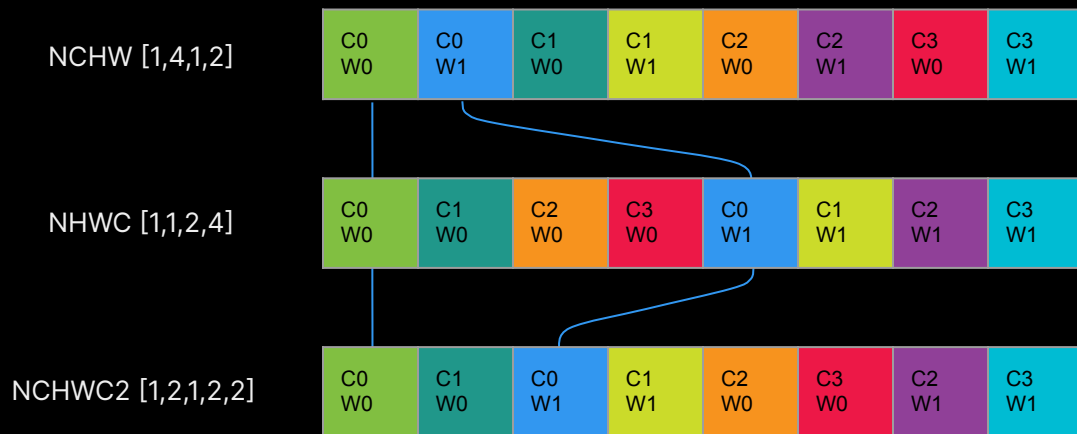
GPU - Tidbits

- Dedicated GPUs often have a warp size of 64 (or 32).
 - Map nicely to convolutions with multiple of 64 kernels, hence the popularity of those sizes.
- First/last convolution of the NN with large input and 3 or 4 channels?
 - Different algo + probably harder to reach great GPU utilization
- For 3×3 kernel winograd is a generally a win
 - For larger kernel size it is harder because of LDS constraint

Optimizations - online

Tensor Memory layout

Memory layout is critical for performance bound applications.



Optimizations - online

Tensor Memory layout

- HW/kernels combination have different preferred memory layouts
- Issues:
 - Memory shuffling around operator is suboptimal
 - Can't alter model weights as they are shared to all worker/backend
- Solution:
 - Subgraph meta-data defined by backend optimisation pass.
 - Reoptimize the graph around the added memory shuffling.


Practical example

Style transfer

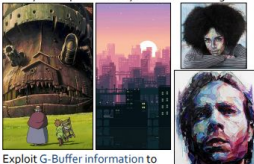
Goal: 30fps on desktop and console (PS4Pro)

Style transfer

Previous work : Research from Unity Labs Grenoble team

 **Multi-Stylization of Video-Games in Real-Time Guided by G-Buffer Information**
Adèle Saint-Denis^{1,2}, Kenneth Vanhoey¹, Thomas Deloit¹
University Paul Sabatier¹, Unity Technologies²

Problem
Render video-games in a given style.
The style is represented by one or more images.



Exploit G-Buffer information to give the artist more control on the stylization.
Challenges : real-time, temporal coherence

Motivation - Previous Work

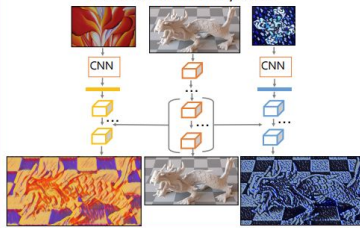
- Arbitrary style transfer [Shiomi et al. 2012]
- Training the CNN with temporal coherence [Huang et al. 2012]
- A meaningful representation of the style [Kulkarni et al. 2015]

• To draw a scene, artists exploit its luminance. [Fiser et al. 2016]

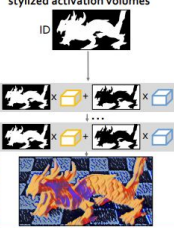
• Segmentation and interpolation of styles to stylize images [Kozlovskiy et al. 2015] [Zhang et al. 2015] [Li et al. 2017]

Our Approach


Pre-Trained Network for Stylization




Interpolation of the neural network's stylized activation volumes



Style Guidance Depth Normals Luminance



Result



Future Work

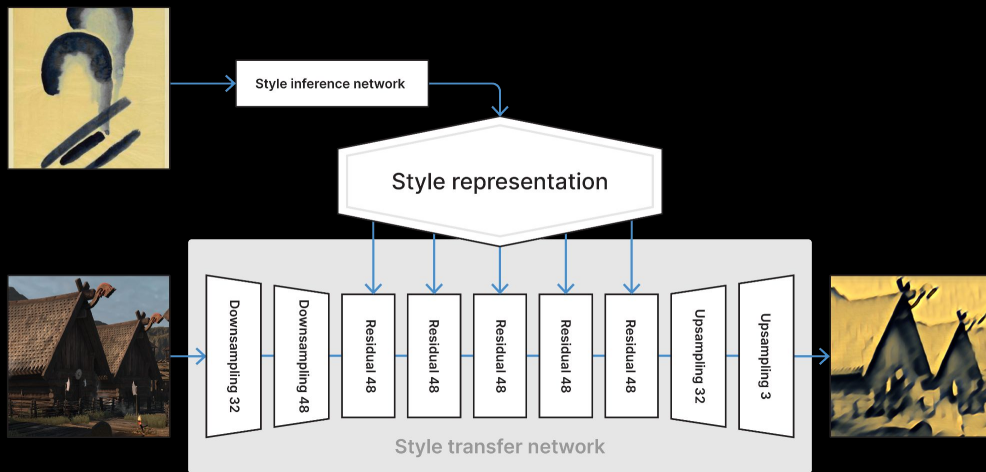
- Interpolation in a multidimensional space including luminance, depth, normals and the id of the scene's objects.
- Train a network with the interpolated styles.



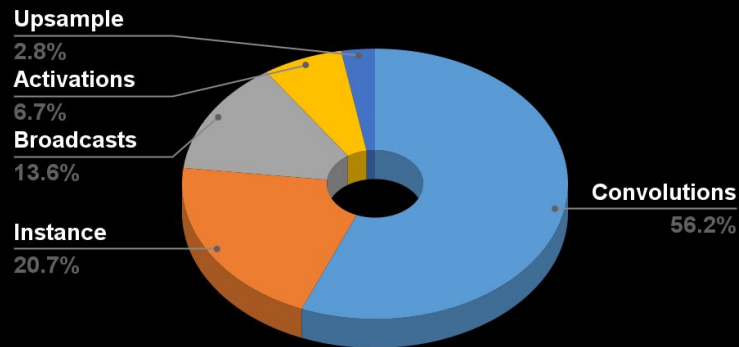
Rendered in Unity using Barracuda
https://720p.unity.com/

Style transfer

Initial exploration and plan



Initial model profiling



Style transfer

Book of dead with style transfer early tests



Style transfer

Some nice bugs/learning

- Models was hallucinating weird colors.
 - Model was trained with sRGB color space while we were feeding it in linear.
 - We converted to/from sRGB before/after the NN to avoid retraining it.

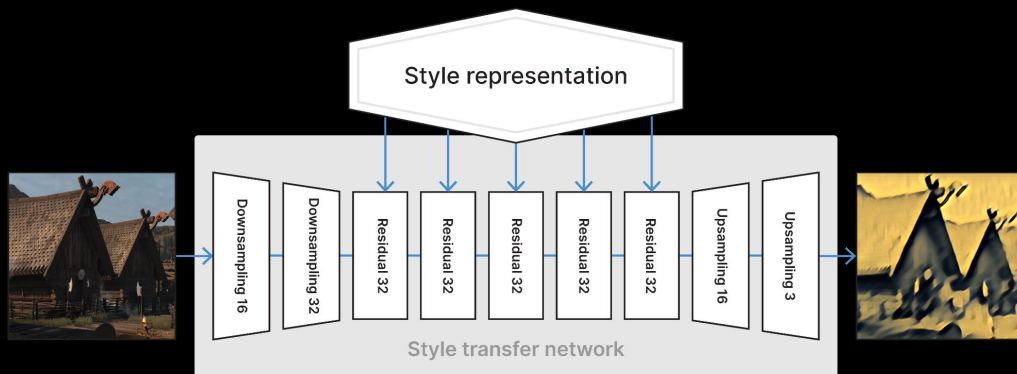
→ Check python texture import code!

- Initially, model was trained with point filtering Upsample creating artifacts.
 - Retraining would take too long.
 - We ended up forcing bilinear interpolation at inference while iterating.

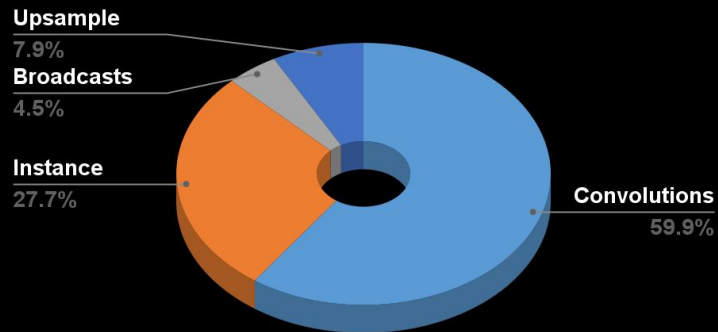
→ Try to uncouple iterations from NN training!

Style transfer

Final architecture

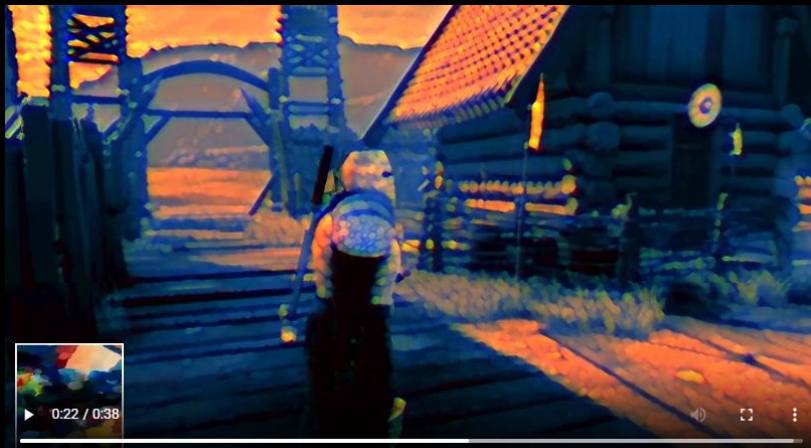
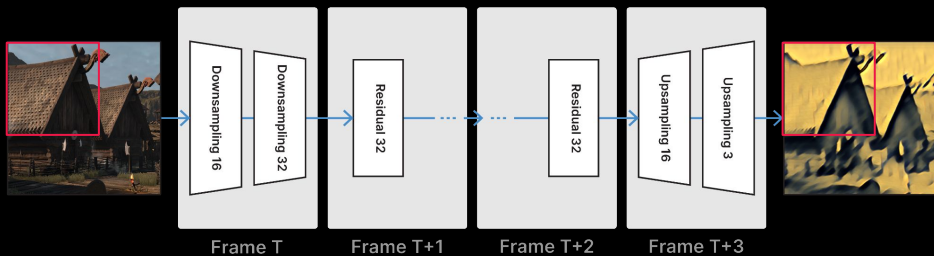


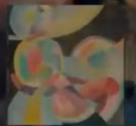
Final model profiling



Style transfer

With temporal reprojection on PS4Pro





Thanks for listening!

We hope the ML and RT3D communities will achieve
great things together!

Thanks to

The Barracuda team

- Alexandre Ribard
- Aurimas Petrovas
- Tracy Sharpe
- Mantas Puida
- Renaldas Zioma
- Florent Guinier

The Grenoble Style transfer team

- Kenneth Vanhoey
- Thomas Deliot
- Adele Saint-Denis

Bonus slides

ONNX

- Gaining traction inside of the ML/DL ecosystem
 - Easy to find exporters for the most popular frameworks
 - Well maintained and updated
- Easy to read and ingest into custom ML implementation
 - Encapsulates both network structure and weights in a single file

ONNX

— From pytorch

```
1 # network
2 net = ...
3
4 # Input to the model
5 x = torch.randn(1, 3, 256, 256)
6
7 # Export the model
8 torch.onnx.export(net,                # model being run
9                   x,                  # model input (or a tuple for multiple inputs)
10                  "example.onnx",      # where to save the model (can be a file or file-like object)
11                  export_params=True,  # store the trained parameter weights inside the model file
12                  opset_version=9,     # the ONNX version to export the model to
13                  do_constant_folding=True, # whether to execute constant folding for optimization
14                  input_names = ['X'],  # the model's input names
15                  output_names = ['Y']  # the model's output names
16                  )
17
```

ONNX

— From TensorFlow

First export tf model to .pb

```
1 # network
2 net = ...
3
4 # Export the model
5 tf.saved_model.save(net, "saved_model")
6 # or
7 tf.train.write_graph(self.sess.graph_def, directory,
8                       'saved_model.pb', as_text=False)
9
```

Then using `tf2onnx` (`pip install tf2onnx`) convert the .pb to ONNX

```
python -m tf2onnx.convert --graphdef model.pb
--inputs=input:0 --outputs=output:0 --output model.onnx
```


ONNX Runtime

- ONNX Runtime follow closely the ONNX specifications. We use it as a reference implementation for our integration tests.
- Support various execution context:
 - CPU
 - GPU (Cuda)
 - DirectML
 - and more!

Great to compare inference speed against our own implementations.