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](https://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

NVidia DLSS

No DLSS
High computational intensity

Denoising
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

Python

ONNX

Unity Editor

Runtime

Keras

PyTorch

TensorFlow
Barracuda pipeline

Python

- Keras
- TensorFlow
- PyTorch
- ...

...
Barracuda pipeline

Python

Keras
TensorFlow
PyTorch
...
Barracuda pipeline

Python

Keras
TensorFlow
PyTorch
...

Unity Editor import

ONNX

Barracuda IR & Optimizations
Barracuda pipeline

Python

Keras
TensorFlow
PyTorch
...

Unity Editor import

ONNX

Barracuda IR & Optimizations

Barracuda runtime

Barracuda Worker
Barracuda pipeline

Keras
TensorFlow
PyTorch
...

Python
Unity Editor import

ONNX
Barracuda IR & Optimizations

Barracuda runtime

Barracuda Worker

Backends
Unity Burst
HPC#
Unity Compute
Unity Pixel
DirectML

SSE
AVX
NEON
DXIL
GLSL
METAL
SPIR-V

Unity Editor
import Barracuda runtime

Python

import Barracuda runtime

SSE
AVX
NEON
DXIL
GLSL
METAL
SPIR-V
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

- Input
- MatMul
- Convolution+Relu
- Reduce
- Output
- Constants
Graph simplification

Input
- MatMul
- MatMul
- Convolution
- Transpose
- Relu
- Reduce
- Transpose
- Output

Constants

MatMul

Output
- Reduce
- Output

Reduce

Convolution+Relu

MatMul

Input

Constants
Graph simplification

Graph Simplification win

- EfficientNet: 1.13%
- InceptionNet: 3.02%
- YoloV4: 6.06%
- ResNet: 6.09%
- DenseNet: 8.52%
- GoogleNet: 10.77%
- MobileNet: 26.89%
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

\[ C = A \times B \]
Typically, convolution are implemented via the im2col algorithm + a MatMul.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>10</th>
<th>11</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td>18</td>
</tr>
</tbody>
</table>

2x2 kernel
Optimization: online

CPU - Convolution

im2col algorithm:

Input tensor: \([H \times W, C]\)

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:

Input tensor: \([H \times W, C]\)
Optimization: online

CPU - Convolution

im2col algorithm:

Input tensor: \([H \times W, C]\)
Optimization: online

CPU - Convolution

im2col algorithm:

Input tensor: \([H \times W, C]\)
Optimization: online

CPU - Convolution

im2col algorithm:

\[
\begin{array}{cccc}
1 & 2 & 4 & 5 \\
2 & 3 & 5 & 6 \\
4 & 5 & 7 & 8 \\
5 & 6 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccc}
10 & 11 & 12 \\
11 & 12 & 14 & 15 \\
13 & 14 & 16 & 16 \\
14 & 15 & 17 & 18 \\
\end{array}
\]

MatMul

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 \\
7 & 7 & 7 & 7 \\
8 & 8 & 8 & 8 \\
\end{array}
\]

\[
\begin{array}{cccc}
KxK\*C \\
KxK \times C \\
KxK \times C \\
\end{array}
\]

\[
\begin{array}{cccc}
F \\
\end{array}
\]
Optimization: online

CPU - Convolution

KxK kernel
Optimizations - online

CPU - Convolutions

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

  We implement convolution as a KxK matrix multiplications which reduces memory consumption by KxK 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

\[
\begin{pmatrix}
10 & 11 & 12 \\
13 & 14 & 15 \\
16 & 17 & 18 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
10 & 11 & 12 \\
13 & 14 & 15 \\
16 & 17 & 18 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
4 & 4 \\
8 & 8 \\
\end{pmatrix}
\]
Optimizations: online

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

NCHW [1,4,1,2]

NCHWC2 [1,2,1,2,2]

NHWC [1,1,2,4]
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
Style transfer

Initial exploration and plan

Initial model profiling
- Upsample: 2.8%
- Activations: 6.7%
- Broadcasts: 13.6%
- Instance: 20.7%

Convolutions: 56.2%
Style transfer

Book of dead with style transfer early tests
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
- Upsample: 7.9%
- Broadcasts: 4.5%
- Instance: 27.7%
- Convolutions: 59.9%
Style transfer

With temporal reprojection on PS4Pro
Thanks for listening!

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

<table>
<thead>
<tr>
<th>The Barracuda team</th>
<th>The Grenoble Style transfer team</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexandre Ribard</td>
<td>Kenneth Vanhoey</td>
</tr>
<tr>
<td>Aurimas Petrovas</td>
<td>Thomas Deliot</td>
</tr>
<tr>
<td>Tracy Sharpe</td>
<td>Adele Saint-Denis</td>
</tr>
<tr>
<td>Mantas Puida</td>
<td></td>
</tr>
<tr>
<td>Renaldas Zioma</td>
<td></td>
</tr>
<tr>
<td>Florent Guinier</td>
<td></td>
</tr>
</tbody>
</table>
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

```python
# network
net = ...

# Input to the model
x = torch.randn(1, 3, 256, 256)

# Export the model
torch.onnx.export(net,
                  x,
                  "example.onnx",
                  export_params=True,
                  opset_version=9,
                  do_constant_folding=False,
                  input_names=['X'],
                  output_names=['Y'])
```

# model being run
# model input (or a tuple for multiple inputs)
# where to save the model (can be a file or file-like object)
# store the trained parameter weights inside the model file
# the ONNX version to export the model to
# whether to execute constant folding for optimization
# the model's input names
# the model's output names
ONNX

—  From TensorFlow

First export tf model to .pb

```python
# network
net = ...

# Export the model
tf.saved_model.save(net, "saved_model")
# or

tf.train.write_graph(sess.graph_def, directory, 
'saved_model.pb', as_text=False)
```

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

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

— From Keras

First you need keras2onnx (pip install keras2onnx)

Then it is quite similar to the pytorch exporter

```python
# network
net = ...

# convert model to ONNX
onnx_model = keras2onnx.convert_keras(net,
                                        name="example",
                                        target_opset=9,
                                        channel_first_inputs=None)

# keras model
# the converted ONNX model internal name
# the ONNX version to export the model to
# which inputs to transpose from NHWC to NCHW

onnx.save_model(onnx_model, "example.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.