Dustin Heither, Maximilian Achenbach and Robert Kagan

Optimizing Deep Learning Performance:

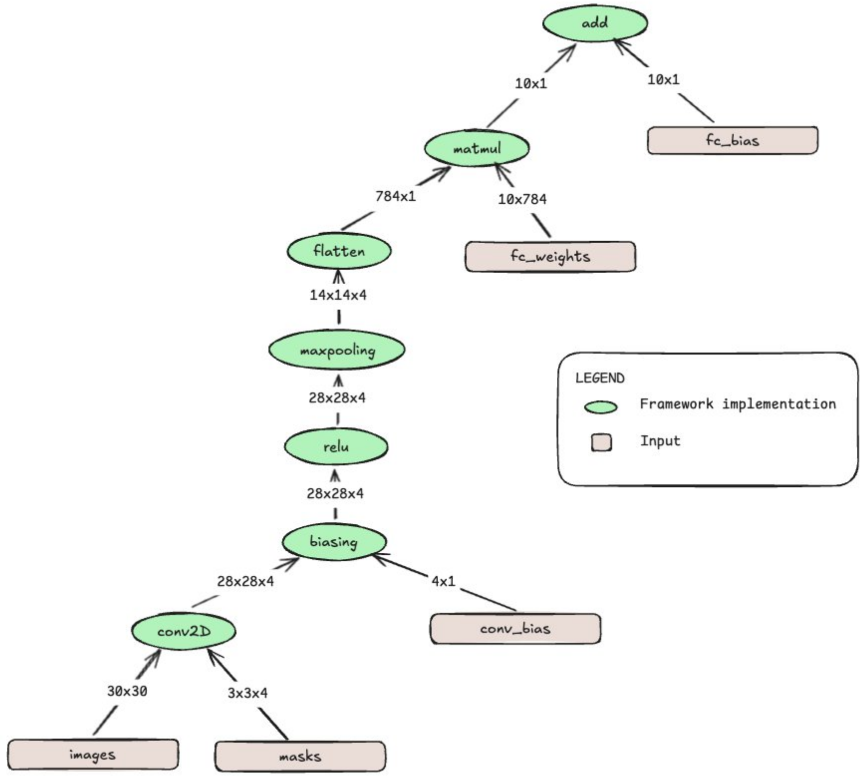
A Hybrid CPU-GPU Framework with Multithreading, SIMD, and Evaluation of Efficiency Metrics

Outline

1. Presentation

TensorFlow

1 / 4



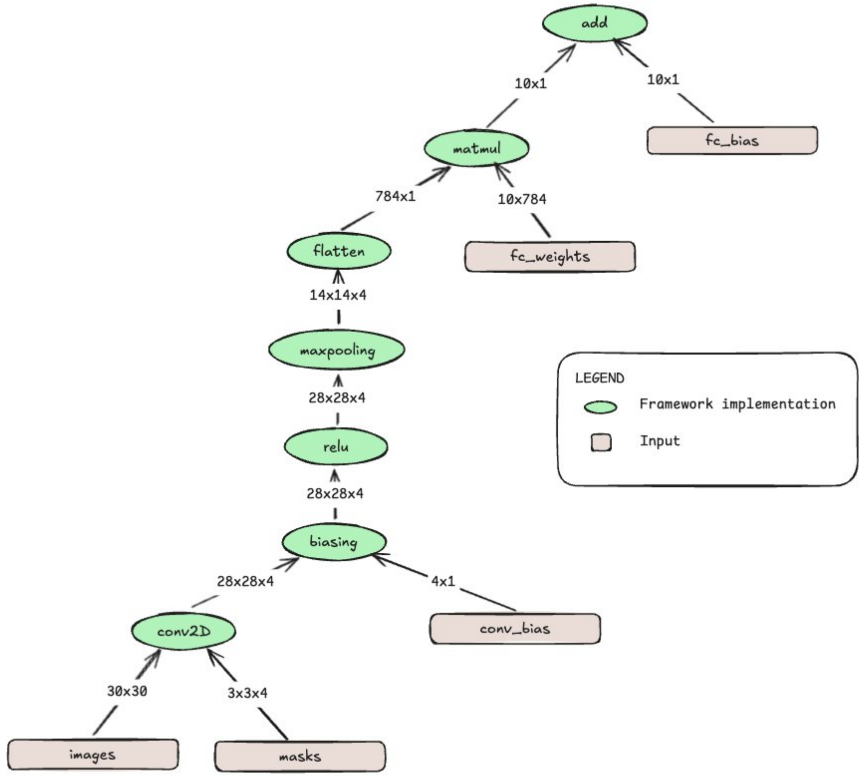
Provided slides

```
#!/usr/bin/env python
import tensorflow # type: ignore
import tensorflow_datasets as tfds # type: ignore
import numpy as np # type: ignore
import os

# https://www.tensorflow.org/datasets/keras_example
tf = tensorflow.compat.v1

# disable eager execution
tf.disable_eager_execution()

# download dataset and store reference in variable
(train_ds, test_ds), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

# normalizes images
def normalize_img(image, label):
    """Normalizes images: `uint8` -> `float32`."""
    return tf.cast(image, tf.float32) / 255.0, label

# tfds provide images of type 'tf.uint8', while the model expects 'tf.float32', therefore, you need to normalize images
train_ds = train_ds.map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)

# reshape datasets to 28 x 28 x 1 pixels (height x width x color channels)
train_ds = train_ds.map(lambda image, label: (tf.reshape(image, [28, 28, 1]), label))

# pad images with 1 row/column of pixels on each side for 3 x 3 filter (border handling)
train_ds = train_ds.map(lambda image, label: (tf.pad(image, [[1, 1], [1, 1], [0, 0]], 'CONSTANT'), label))

# cache the modified data in memory
train_ds = train_ds.cache()

# shuffling and dividing in batches
shuffle_size = 60000
batch_size = 128
train_ds = train_ds.shuffle(shuffle_size).batch(batch_size)

# define iterator over batches of data
data_iterator = tf.data.Iterator.from_structure(tf.data.get_output_types(train_ds), tf.data.get_output_shapes(train_ds))

# define graph operation which initializes the iterator with the dataset
train_init_op = data_iterator.make_initializer(train_ds)

# define graph operation which gets the next batch of the iterator over the dataset
next_data_batch = data_iterator.get_next()
...
```

Dataset iterator

TensorFlow

2 / 4



Provided slides

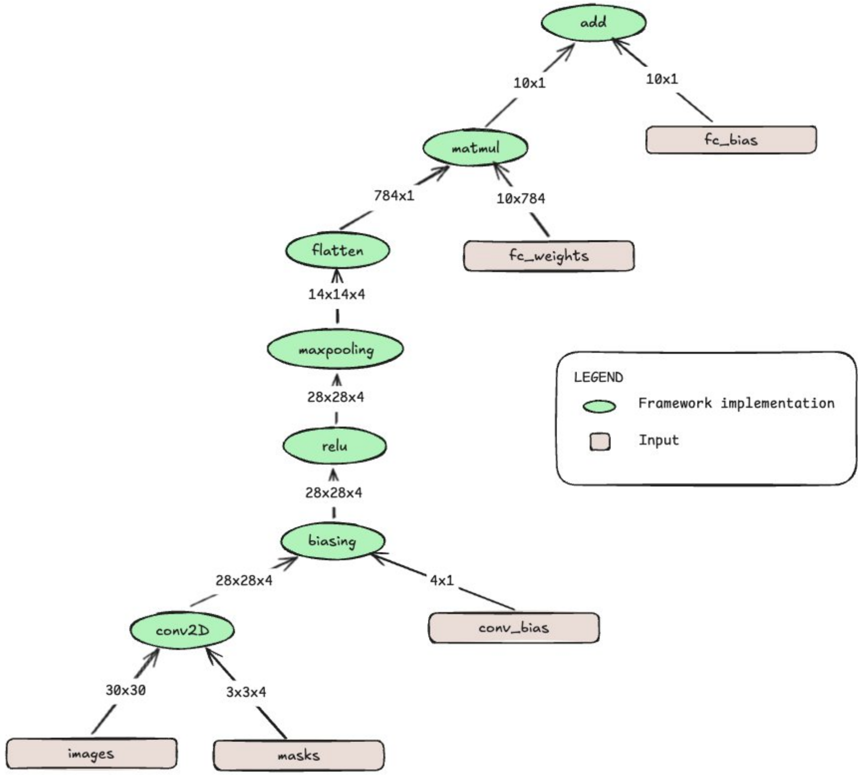
→

```
...
# define initialization parameters
mu = 0
sigma = 0.1

# build graph with one convolutional layer (with 4 masks) and one fully connected layer
images = tf.placeholder(tf.float32, shape=(None, 30, 30, 1), name='images')
masks = tf.Variable(tf.random.truncated_normal(shape=(3, 3, 1, 4), mean=mu, stddev=sigma))
fc_weights = tf.Variable(tf.random.truncated_normal(shape=(14 * 14 * 4, 10), mean=mu, stddev=sigma))
conv_bias = tf.Variable(tf.zeros(4), name="conv_bias")
fc_bias = tf.Variable(tf.zeros(10), name='fc_bias')

input_layer = images
convolution = tf.nn.conv2d(input_layer, masks, strides=[1, 1, 1, 1], padding='VALID', name='conv2D')
convolution = tf.add(convolution, conv_bias, name='biasing')
convolution = tf.nn.relu(convolution, name='ReLU')
pooling = tf.nn.max_pool2d(convolution, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')
hidden_layer = tf.keras.layers.Flatten()(pooling)
output_layer = tf.add(tf.matmul(hidden_layer, fc_weights, name='matmul'), fc_bias, name='add')

# feed the correct labels into the net
labels = tf.placeholder(tf.int32, (None), name='labels')

# highest value is the guess of the network
network_prediction = tf.argmax(output_layer, axis=1, output_type=tf.int32)

# return 0.0 if net prediction is wrong, 1.0 if true
is_correct_prediction = tf.equal(network_prediction, labels)

# percentage of correct predictions is the mean of the batch
accuracy_op = tf.reduce_mean(tf.cast(is_correct_prediction, tf.float32))

# initialize all variables before evaluating the graph
session = tf.Session()
session.run(tf.global_variables_initializer())

# define the loss graph
onehot_labels = tf.one_hot(labels, 10)
loss_op = tf.losses.softmax_cross_entropy(onehot_labels=onehot_labels, logits=output_layer)

# define training parameters
learning_rate = 0.01

# define the training graph
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
train_op = optimizer.minimize(loss_op)
...
```

Graph

TensorFlow

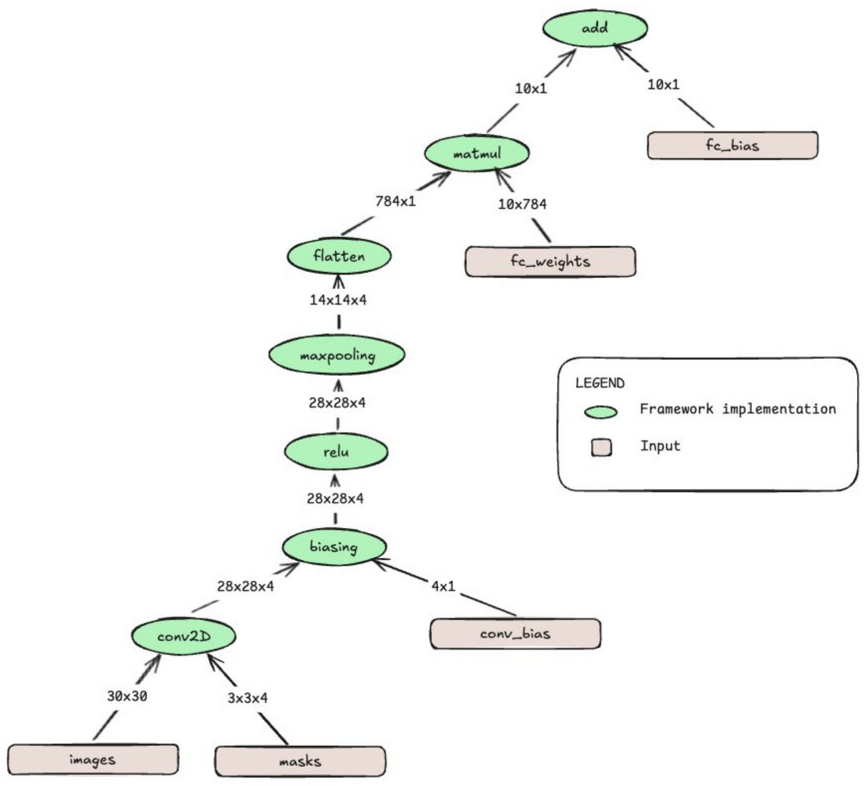
3 / 4



Provided slides

```
...
epochs = 100

# train the weights by looping repeatedly over all the data (and shuffling in between)
for i in range(epochs):
    session.run(train_init_op)
    accuracy = 0
    loss = 0
    while True:
        try:
            data_batch = session.run(next_data_batch)
            image_batch = data_batch[0]
            label_batch = data_batch[1]
            session.run(train_op, feed_dict={images:image_batch, labels:label_batch})
            accuracy = session.run(accuracy_op, feed_dict={images:image_batch, labels:label_batch})
            loss = session.run(loss_op, feed_dict={images:image_batch, labels:label_batch})
        except tf.errors.OutOfRangeError:
            break
    print(f"Epoch {i} done: accuracy {accuracy * 100:.2f}%, loss {loss * 100:.2f}%")
...
```

Training



Provided slides

```
...
# ensure the directory exists
try:
    os.mkdir("./data")
except:
    pass


# save output
with open("./data/conv_bias.txt", "w") as f:
    # first two lines are the shape
    np.savetxt(f, conv_bias.shape, fmt='%f')
    f.write("\n")
    np.savetxt(f, conv_bias.eval(session=session), fmt='%f')
...
```

Saving weights

Input/output

Matrix

```
3.000000
3.000000

0.707285 0.641241 -0.163969
0.302579 1.043134 0.891180
-0.876062 0.260266 1.070686
```

./data/masks_0.txt

→

```c
#ifndef MATRIX_H
  #define MATRIX_H

    typedef struct matrix {
        int x;
        int y;
        float** m;
    } matrix;

    void print_matrix(matrix* a);
    matrix** flip_kernels(matrix** a, int len);
    matrix* transpose(matrix* a);
    matrix* malloc_matrix(int x, int y);
    void free_matrix(matrix* a);
    void free_matrix_ptr(matrix** a, int len);
#endif
```

./h/matrix.h

Input/output

IO

```c
#ifndef MATRIX_H
  #define MATRIX_H

  typedef struct matrix {
    int x;
    int y;
    float** m;
  } matrix;

  void print_matrix(matrix* a);
  matrix** flip_kernels(matrix** a, int len);
  matrix* transpose(matrix* a);
  matrix* malloc_matrix(int x, int y);
  void free_matrix(matrix* a);
  void free_matrix_ptr(matrix** a, int len);
#endif
```

./h/matrix.h

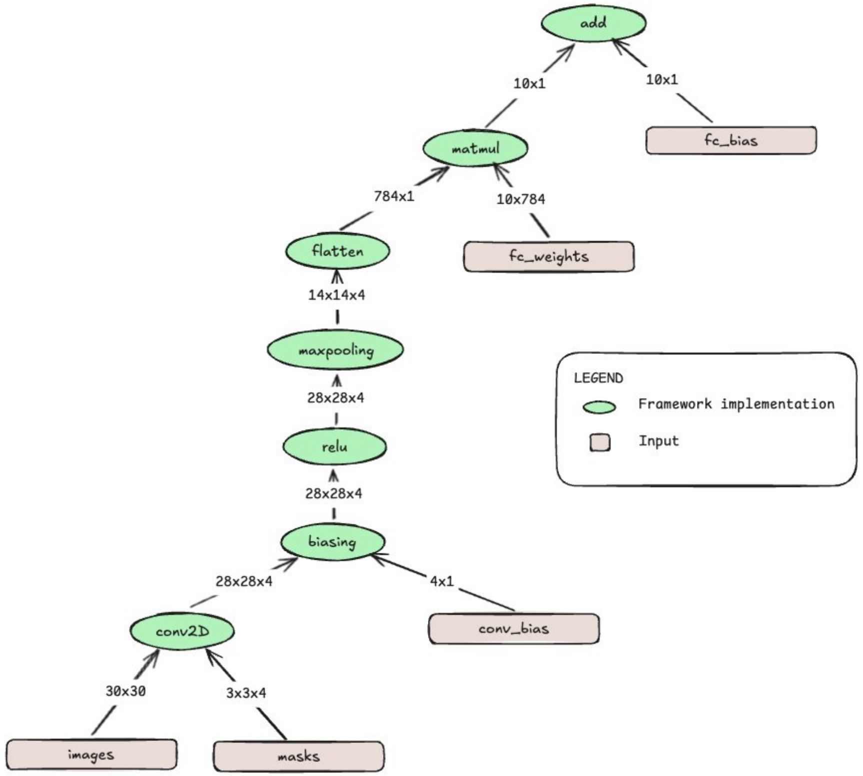
→

```c
#ifndef IO_H
  #define IO_H

  #include "matrix.h"

  typedef struct io {
    matrix* conv_bias;
    matrix* fc_bias;
    matrix* fc_weights;
    int image_len;
    matrix** image;
    int* label;
    int masks_len;
    matrix** masks;
  } io;

  matrix* io_to_matrix(char* a);
  io* malloc_io();
  void free_io(io* a);
#endif
```

./h/io.h

C implementation

Tensorflow



TensorFlow

```c
#ifndef TF_H
  #define TF_H

  #include "matrix.h"

  int max(matrix* a);
  matrix* add(matrix* a, matrix* b);
  matrix* matmul(matrix* a, matrix* b);
  matrix* flatten(matrix** a, int len);
  matrix** maxpool(matrix** a, int len);
  matrix** hyperbolic_tangent(matrix** a, int len);
  matrix** relu(matrix** a, int len);
  matrix** biasing(matrix** a, int len, matrix* b);
  matrix** conv2d(matrix* a, matrix** b, int len);
#endif
```

./h/tf.h

Hardware

Overview

CPU	Release dates	TDP (W)
AMD Ryzen 7 3800XT	7. Juli 2020	105
Apple M3 Pro 11-Core	30. Oktober 2023	27
Intel Core i7 1065G7	1. Juni 2019	15

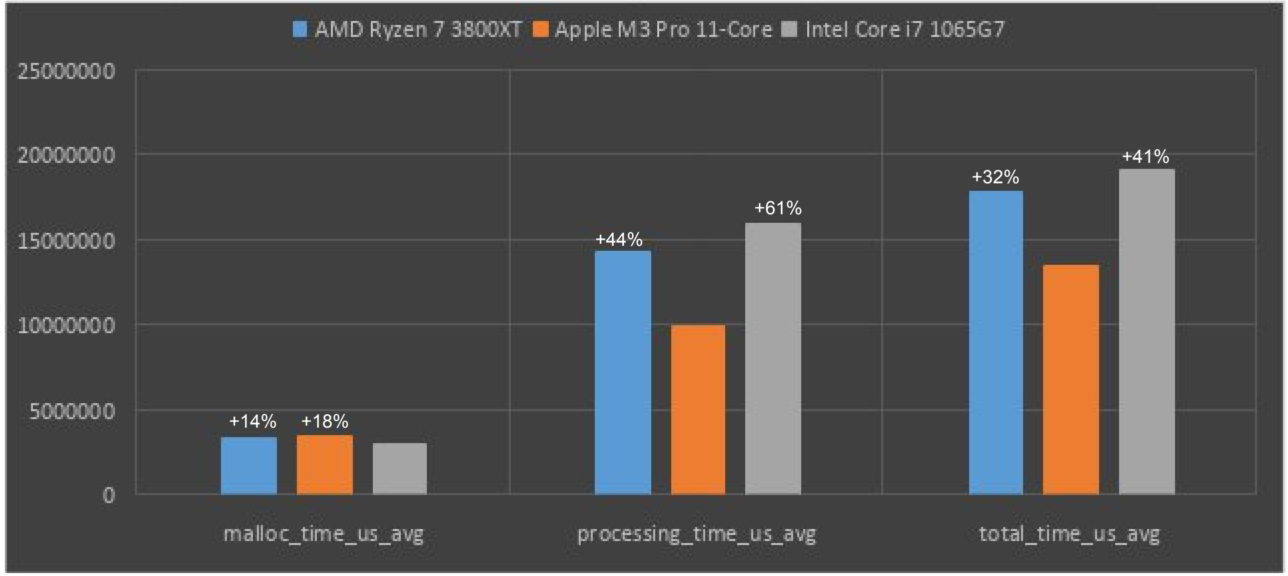
Benchmark

Overview

- Batch size: 1
- Epochs: 60000
- **malloc_time_us_avg**
  - In microseconds
  - Averaged over 10 runs
  - Read in files
  - Create matrix structs
  - Create io struct
- **processing_time_us_avg**
  - In microseconds
  - Averaged over 10 runs
  - Calculate the results of all epochs
- **total_time_us_avg**
  - In microseconds
  - Averaged over 10 runs
  - Total time with garbage collection

Benchmark

Performance

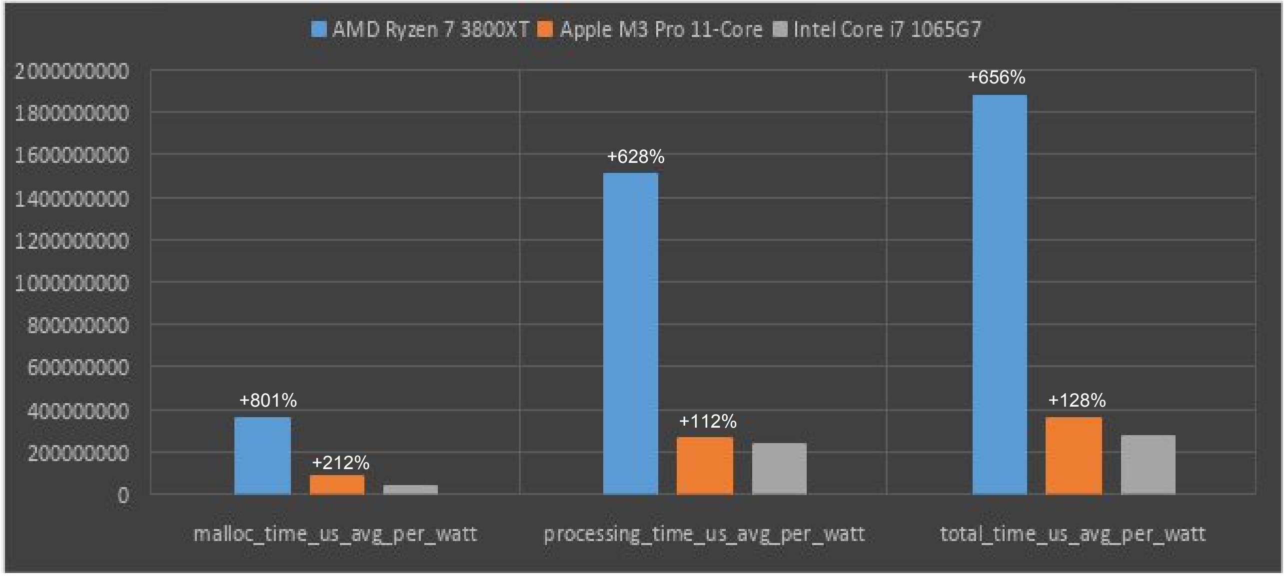


Performance in microseconds (averaged over 10 test runs)

*lower is better

Benchmark

Performance per watt



Performance per watt in microseconds (averaged over 10 test runs)

*lower is better

Outlook

Work in progress

- (Apple M3 Pro NPU)
- Multithreading
  - Implemented, but needs a lot of testing
- SIMD
  - Arm Neon
  - Quantization
  - SSE vs. AVX2 vs. AVX-512
  - ICX vs. GCC
    - Both finally work, but Intel oneAPI needs testing
- CUDA tuning
  - NVCC finally works