10th EDITION EUROPEAN CONGRESS

ERTS 2020

PROFILING AND OPTIMIZATION OF DEEP NEURAL NETWORKS FOR EMBEDDED AUTOMOTIVE APPLICATIONS

Loïc CORDONE, Eric PERRAUD and Jean-Marc GABRIEL
Renault Software Labs, Toulouse and Sophia-Antipolis

INTRODUCTION

- Deep Neural Networks (DNNs) now have excellent accuracy

⇒ Car manufacturers consider using DNNs for their applications

- Ease of development thanks to DL frameworks and state-of-the-art models
- But their integration on embedded systems represents an industrial challenge:
  - High constraint on latency
  - On low-cost hardware with limited computing power, memory and power consumption

Objectives:

1. Assess the inference latency and determine where an optimization effort should focus
2. Compile and optimize the model for a fast and lightweight inference on the target hardware

SCOPE OF STUDY

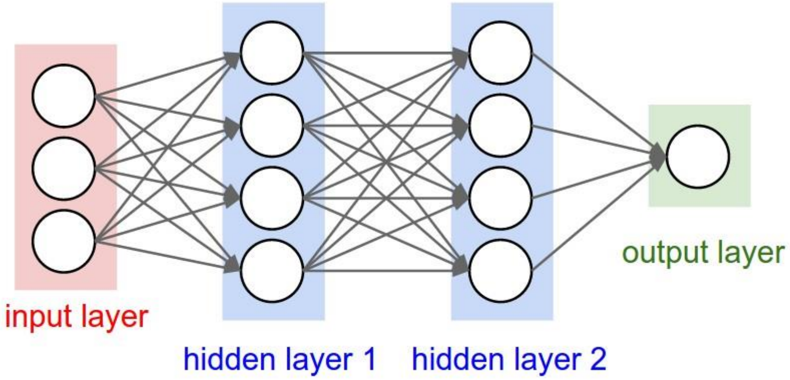
- Variety of embedded solutions: multicore CPU (ARM, Intel), FPGAs, embedded GPU

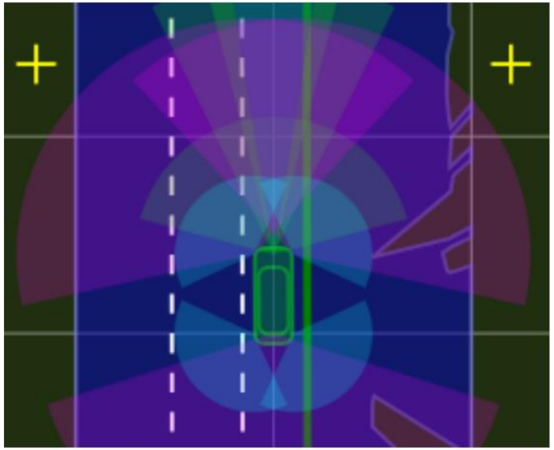
$\Rightarrow$ Still unclear which hardware architecture will be preferred for embedded DNNs

- Our approach is **hardware-independent**
- We considered 3 representative classes of embedded neural networks:
  - Fully-Connected Neural Networks (FC-DNN), used for a variety of small functions
  - Convolutional Neural Networks (CNN), used in a multitude of computer vision applications
  - Recurrent Neural Networks (RNN), for problems involving time series

STEERING WHEEL ANGLE PREDICTION FC-DNN

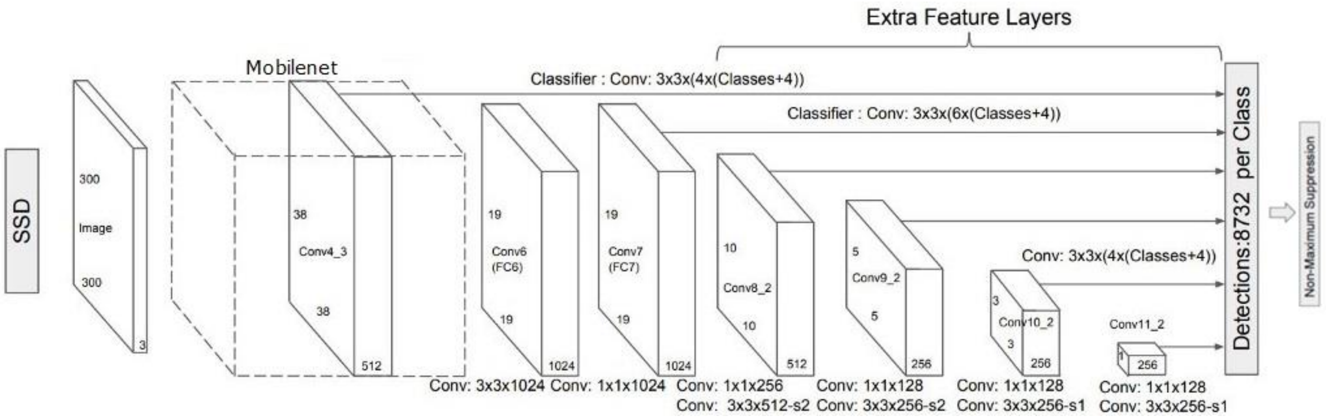
Fully-connected DNN: 13-128-128-1

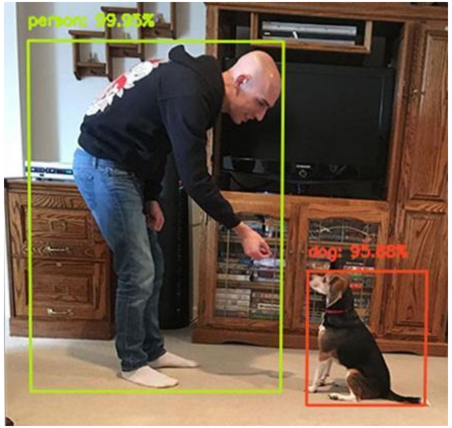




Trained internally with Renault data

OBJECT DETECTION CNN: MOBILENET+SSD

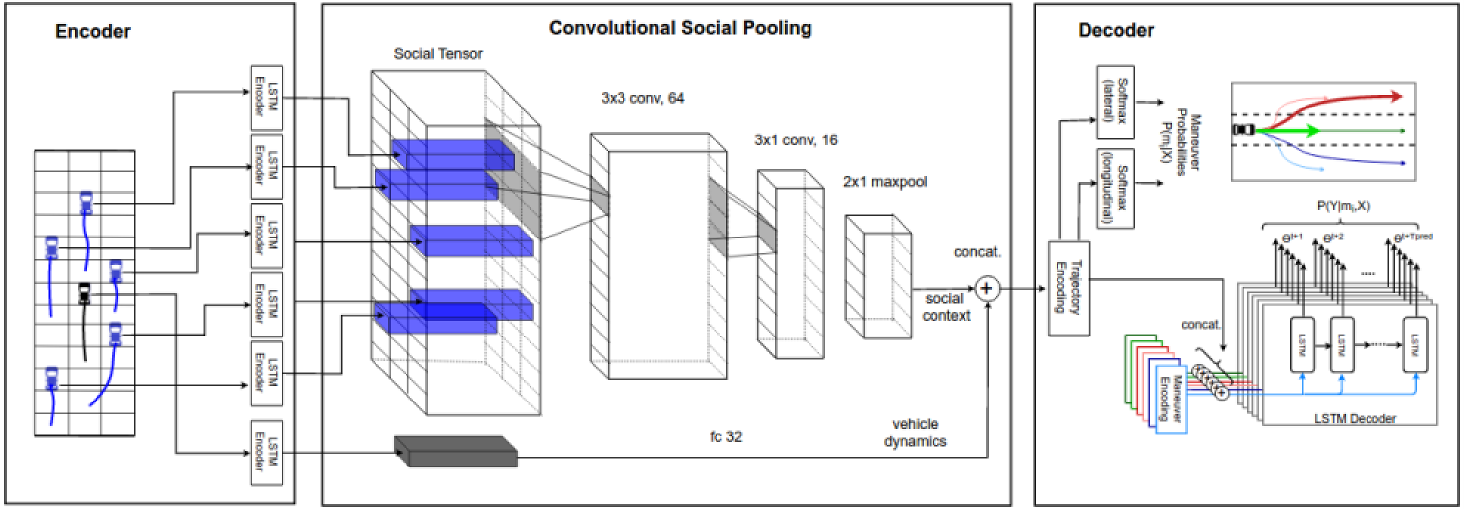




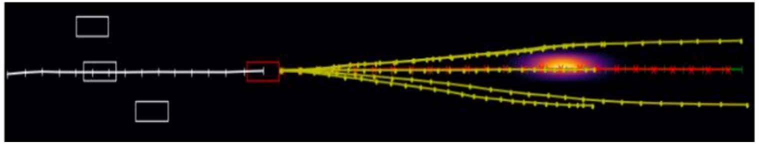
"MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", Howard et al. (2017)

TRAJECTORY PREDICTION RNN: CS-LSTM

Inputs: Position histories of the vehicle and up to 38 neighboring vehicles during the last 3 seconds



Ouputs: For each maneuver, trajectory prediction over the next 5 seconds



"Convolutional Social Pooling for Vehicle Trajectory Prediction", N. Deo, M. Trivedi (2018)

PROFILING AND DEEP LEARNING PROFILERS

Profiling: measuring the space or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls

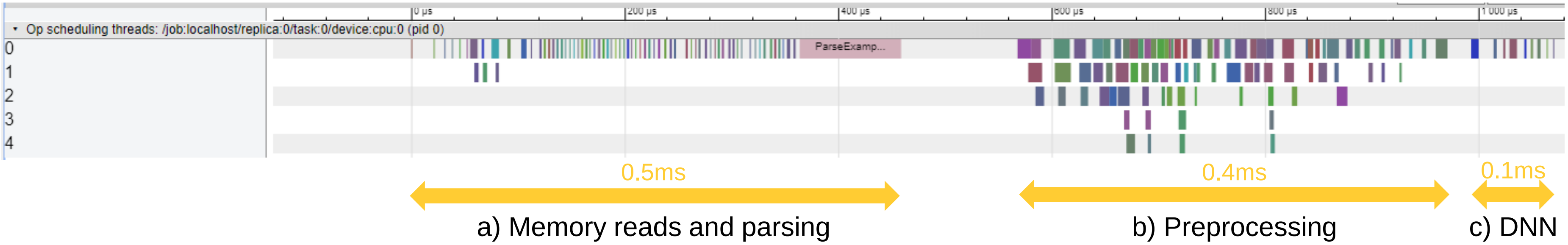
- Most models are trained and executed in frameworks

⇒ High-level profiling: inference time, frequency and duration of the framework function calls

These measures will be gathered with the profilers integrated in each deep learning frameworks

PROFILING RESULTS FOR THE FC-DNN

Profiling of the **13-128-128-1 network** with TensorFlow Profiler:

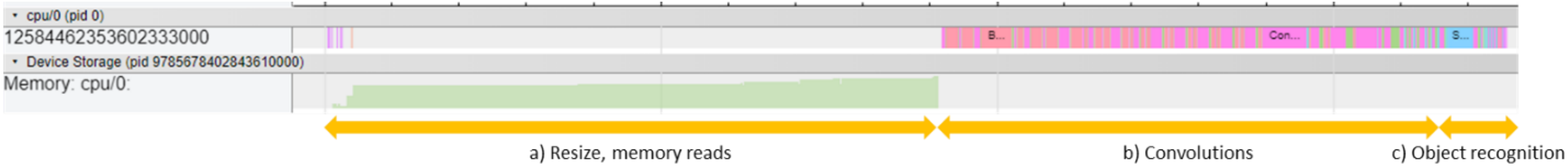


- Inference time on CPU: **1ms**

- Network traversal represents less than 10% of the inference time
- The inference optimization should focus on the **data ingestion/preprocessing pipeline**

PROFILING RESULTS FOR THE OBJECT RECOGNITION CNN

Profiling of the **MobileNet+SSD** CNN with MX-Net Profiler:



- Inference time on CPU: **60ms** (16 FPS) ; on GPU: **12ms** (83 FPS)

- Convolutions represent more than 60% of the inference time
- …and are not parallelized over the multiple CPU cores

- State-of-the-art model, not easily retrainable

PROFILING RESULTS FOR THE TRAJECTORY PREDICITION RNN

Profiling of the **CS-LSTM** RNN with PyTorch Profiler (top 5 operations):

Operation name	CPU total time (ms)	CPU total %	Number of calls
addmm	27.3ms	45.8%	335
sigmoid	6.2ms	10.3%	498
tanh	5.9ms	9.9%	338
mul	3.8ms	6.4%	515
add	3.7ms	6.3%	349

- Inference time on CPU: **36ms**

- Lot of diverse operations, matrix multiplications add up to 60% of CPU total time
- Activation functions represent 20% of inference time => look for alternatives

PROFILING CONCLUSIONS

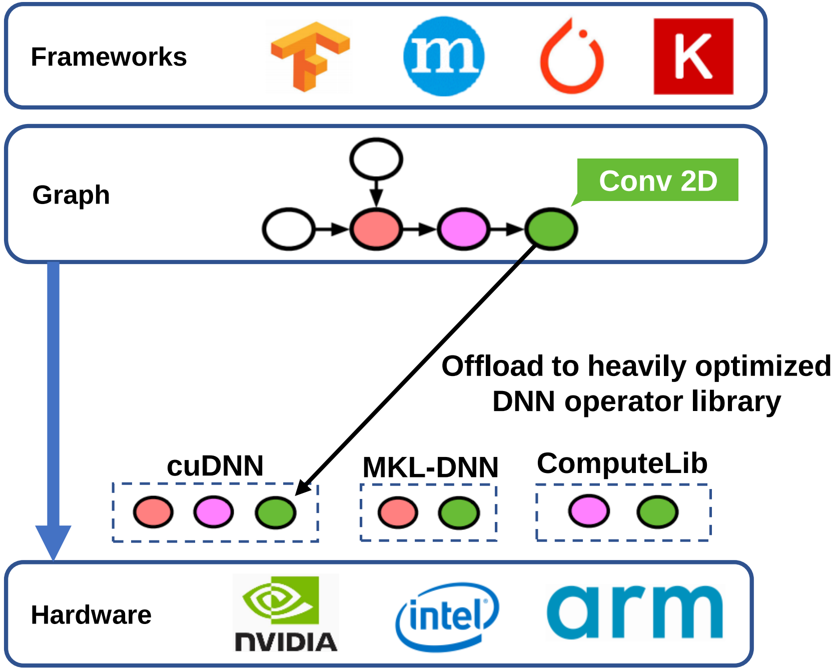
- Depending on the model, the focus shall be put on:
  - Data ingestion (FC-DNN), outside the model
  - Changing the way a specific operation is performed (parallelize convolutions in CNN)
  - Modify the network to reduce its inference time

Now that the bottlenecks are identified, can we do something about it?

DIFFERENT LEVELS OF OPTIMIZATION

Optimization possible at 3 levels:

- **Model**: pruning, quantization
- **Graph**: graph simplification, operation fusion
- **Operation (DNN)**: tiling, parallelization



DEEP LEARNING COMPILERS

- DNNs are simple programs
- DNN compilation for inference: optimized result for target hardware
- Strong trend among AI companies

tvm

  - **Compilation for CPU, GPU, FPGA, ASIC**
  - **Support of all major Deep Learning frameworks**
  - **Automatic optimization for a target hardware**

OPTIMIZATIONS DEFINITION WITH TVM

$A^T B$ operation

Description

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
    lambda y, x:
    t.sum(A[k, y] * B[k, x], axis=k))
```

Default schedule

```
s = tvm.create_schedule(C.op)
```

generated in x86, CUDA…

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

CPU schedule

```
yo, xo, yi, xi = s[C].title(y, x, ty, tx)
s[C].reorder(yo, xo, k, yi, xi)
```

generated in x86

```
for yo in range(1024 / ty):
  for xo in range(1024 / tx):
    C[yo*ty:yo*ty+ty][xo*tx:xo*tx+tx] = 0
    for k in range(1024):
      for yi in range(ty):
        for xi in range(tx):
          C[yo*ty+yi][xo*tx+xi] +=
             A[k][yo*ty+yi] * B[k][xo*tx+xi]
```

GPU schedule

```
yo,xo,ko,yi,xi,ki = s[C].title(y,x,k,8,8,8)
s[C].tensorize(yi, intrin.gemm8x8)
```

generated in CUDA

```
for yo in range(128):
  for xo in range(128):
    intrin.fill_zero(C[yo*8:yo*8+8][xo*8:xo*8+8])
    for ko in range(128):
      intrin.fused_gemm8x8_add(
        C[yo*8:yo*8+8][xo*8:xo*8+8],
        A[ko*8:ko*8+8][yo*8:yo*8+8],
        B[ko*8:ko*8+8][xo*8:xo*8+8])
```

written code

equivalent generated pseudo-code

AUTOTVM: AUTOMATIC OPTIMIZATION FOR A TARGET HARDWARE

$A^T B$ operation

Description

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
    lambda y, x:
    t.sum(A[k, y] * B[k, x], axis=k))
```

CPU schedule

```
yo, xo, yi, xi = s[C].title(y, x, ty, tx)
s[C].reorder(yo, xo, k, yi, xi)
```

generated

in x86

```
for yo in range(1024 / ty):
  for xo in range(1024 / tx):
    C[yo*ty:yo*ty+ty][xo*tx:xo*tx+tx] = 0
    for k in range(1024):
      for yi in range(ty):
        for xi in range(tx):
          C[yo*ty+yi][xo*tx+xi] +=
             A[k][yo*ty+yi] * B[k][xo*tx+xi]
```

AutoTVM

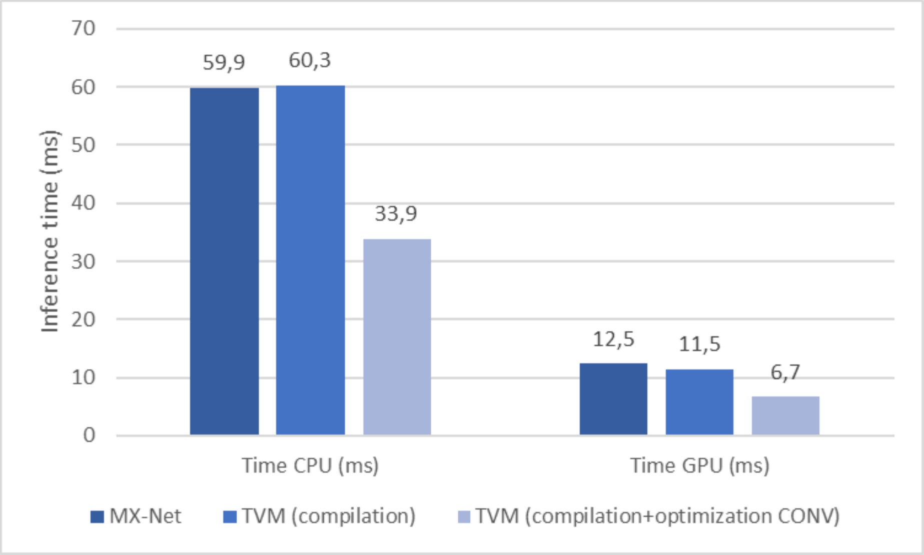
- tx, ty ∈ [1, 2, 4, 8, 16, 32, etc.]
- For each operation, search the best combination of parameters

written code

equivalent generated pseudo-code

OPTIMIZATION RESULTS FOR THE OBJECT RECOGNITION CNN

Compilation and optimization of **28 convolutions** on Intel Core i7 (8 coeurs, 3GHz) and NVIDIA RTX 2060



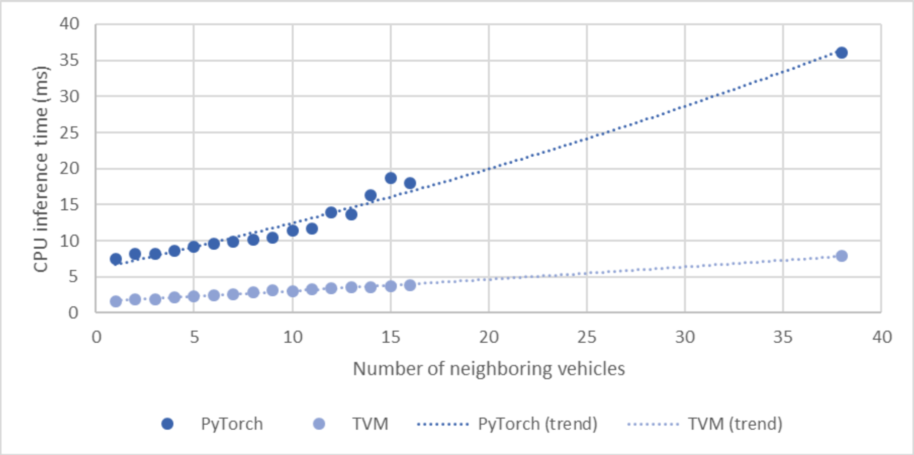
Divided by 2

OPTIMIZATION RESULTS FOR THE TRAJECTORY PREDICTION RNN

Compilation and optimization of the **2 * n_vehicles FC layers** on Intel Xeon E5-2690 v2 (10 cores, 3GHz)

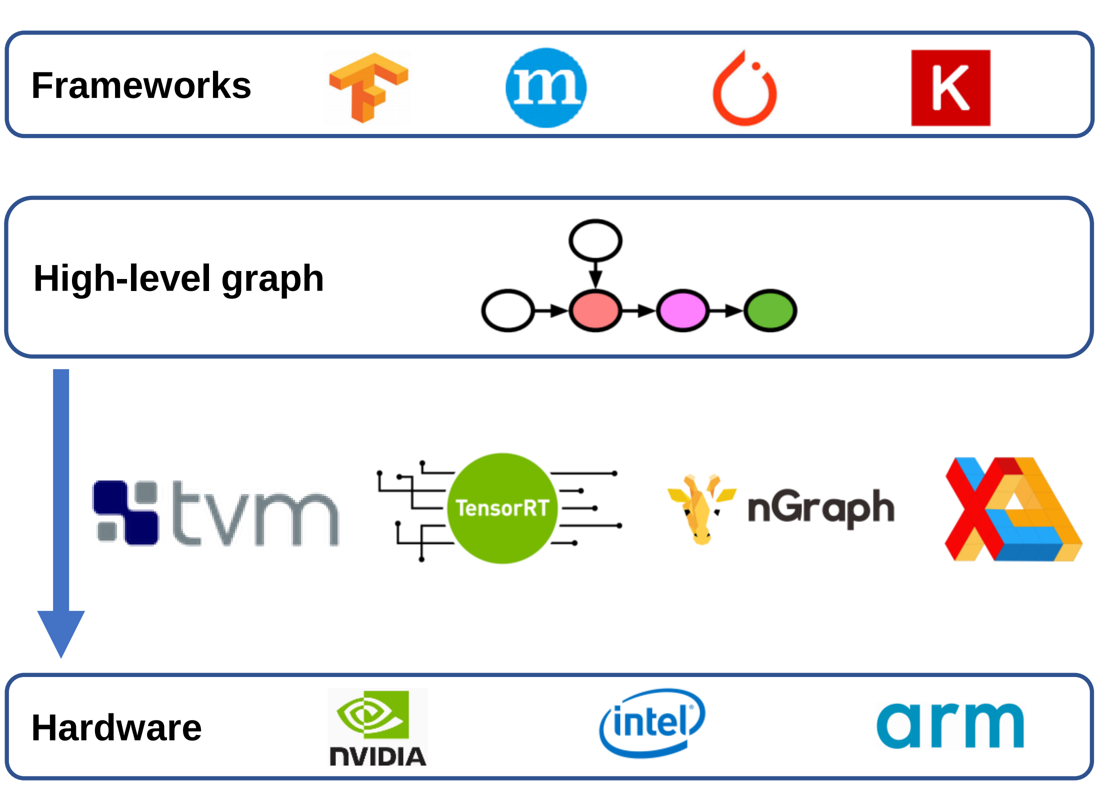
Situation	PyTorch	TVM	Tuned TVM
EGO+6V	9,5 ms	2,5 ms	2,4 ms
EGO+16V	18,1 ms	3,9 ms	3,8 ms
EGO+38V	36,1 ms	7,9 ms	7,8 ms

Divided by 4



- Compilation (graph optimization) **more important** than auto-tuning, due to the variety of operations

CONCLUSIONS



DNN profiling

- Model conception issues
- Identify bottlenecks

DNN optimization

- Best optimization
- Fast and lightweight inference
- Complete separation between the DNN design and its porting on embedded systems
- Embedding on new hardware (FPGAs)

THANK YOU

THANK YOU

OPTIMIZATION RESULTS FOR THE OBJECT RECOGNITION CNN

CPU inference, w/o optimizations : **16 FPS**

CPU inference, w/ optimizations : **26 FPS**

60% more FPS
or half the inference time, **for the same computations**

BONUS

FRAMEWORK MODEL IMPORT IN TVM AND COMPILATION

```
print("Load the model...")
inf_json = os.path.join(data_dir, model_name + "-symbol.json")
print("mxnet.sym.load: " + inf_json)
sym = mxnet.sym.load(inf_json)

checkpoint = os.path.join(data_dir, model_name)
print("load_checkpoint: " + checkpoint)
_, arg_params, aux_params = load_checkpoint(checkpoint, 0)

mod, params = relay.frontend.from_mxnet(sym, {"data": dshape},
                                        arg_params=arg_params, aux_params=aux_params)
net = mod[mod.entry_func]


print("Compile...")
with relay.build_config(opt_level=3):
    graph, lib, params = relay.build_module.build(
        net, target=target, params=params)
```

llvm, cuda, arm

For each operation, load its default schedule for the target, then optimize the graph

BONUS

AUTO-TUNING

```
print("Load model...")
net, params, input_shape = get_network(model_name, batch_size)

print("Extract tasks...")
tasks = autotvm.task.extract_from_program(
    net,
    target=target,
    params=params,
    ops=(relay.op.nn.conv2d,))

print("Tuning kernels...")
for i, tsk in enumerate(tasks):
    prefix = "[Task %2d/%2d] " % (i+1, len(tasks))

    # converting conv2d tasks to conv2d_NCHWc tasks
    op_name = tsk.workload[0]
    if op_name == 'conv2d':
        func_create = 'topi_x86_conv2d_NCHWc'
    elif op_name == 'depthwise_conv2d_nchw':
        func_create = 'topi_x86_depthwise_conv2d_NCHWc_from_nchw'
```

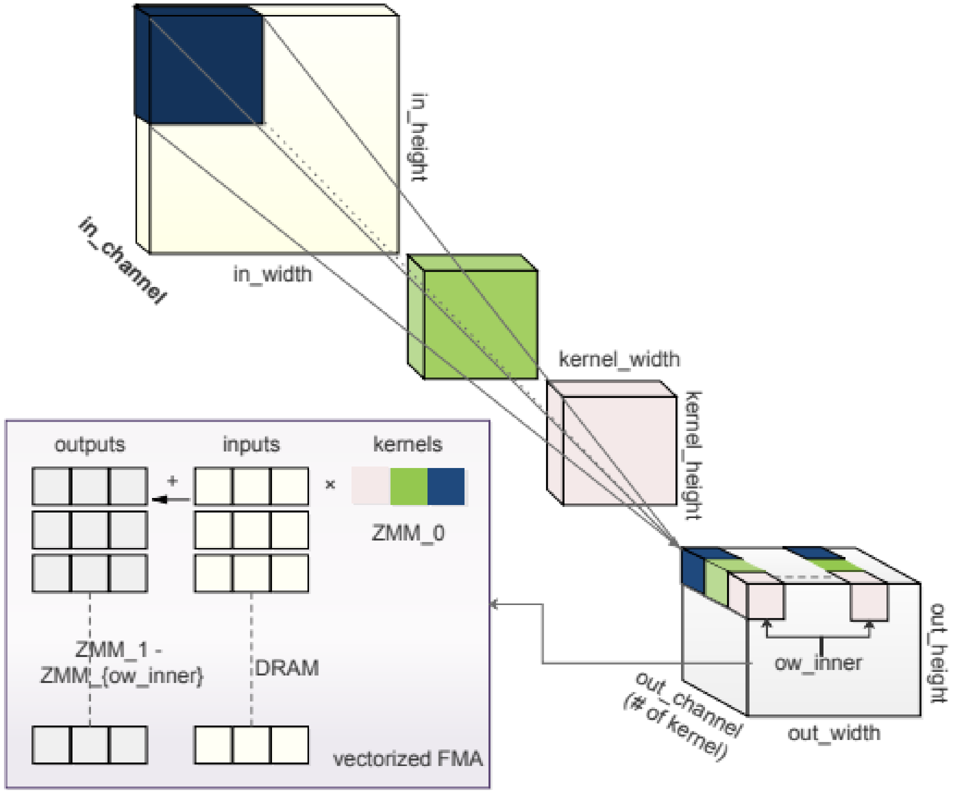
```
if tuner == 'xgb' or tuner == 'xgb-rank':
    tuner_obj = XGBTuner(task, loss_type='rank')
elif tuner == 'ga':
    tuner_obj = GATuner(task, pop_size=50)
elif tuner == 'random':
    tuner_obj = RandomTuner(task)
elif tuner == 'gridsearch':
    tuner_obj = GridSearchTuner(task)

n_trial=len(task.config_space)
tuner_obj.tune(n_trial=n_trial,
               early_stopping=early_stopping,
               measure_option=measure_option,
               callbacks=[
                   autotvm.callback.progress_bar(n_trial, prefix=prefix),
                   autotvm.callback.log_to_file(log_filename)])
```

BONUS

COMPILATION AFTER AUTO-TUNING

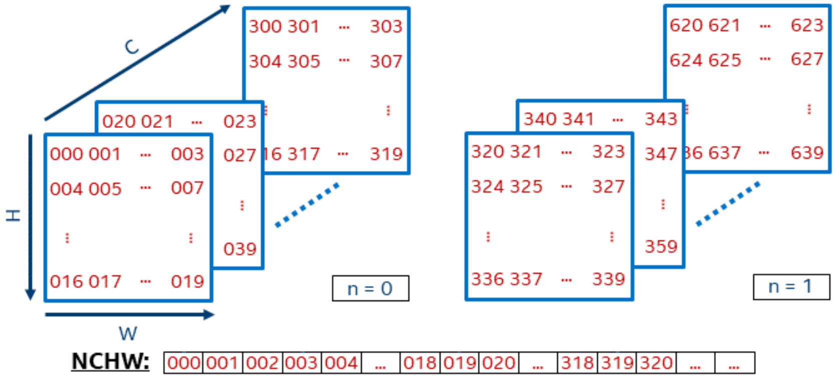
```
print("Load the model...")
inf_json = os.path.join(data_dir, model_name + "-symbol.json")
print("mxnet.sym.load: " + inf_json)
sym = mxnet.sym.load(inf_json)

checkpoint = os.path.join(data_dir, model_name)
print("load_checkpoint: " + checkpoint)
_, arg_params, aux_params = load_checkpoint(checkpoint, 0)

mod, params = relay.frontend.from_mxnet(sym, {"data": dshape},
                                        arg_params=arg_params, aux_params=aux_params)
net = mod[mod.entry_func]

with autotvm.apply_history_best(log_file):
    print("Compile...")
    with relay.build_config(opt_level=3):
        graph, lib, params = relay.build_module.build(
            net, target=target, params=params)
```

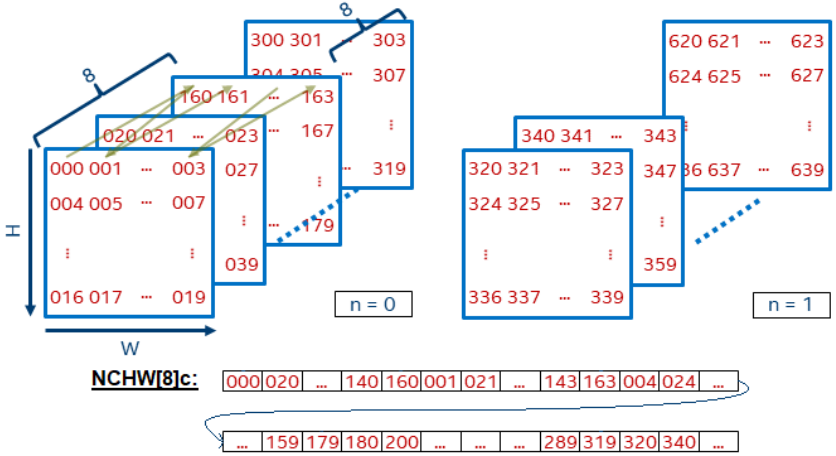
CONVOLUTION OPTIMIZATION ON CPU



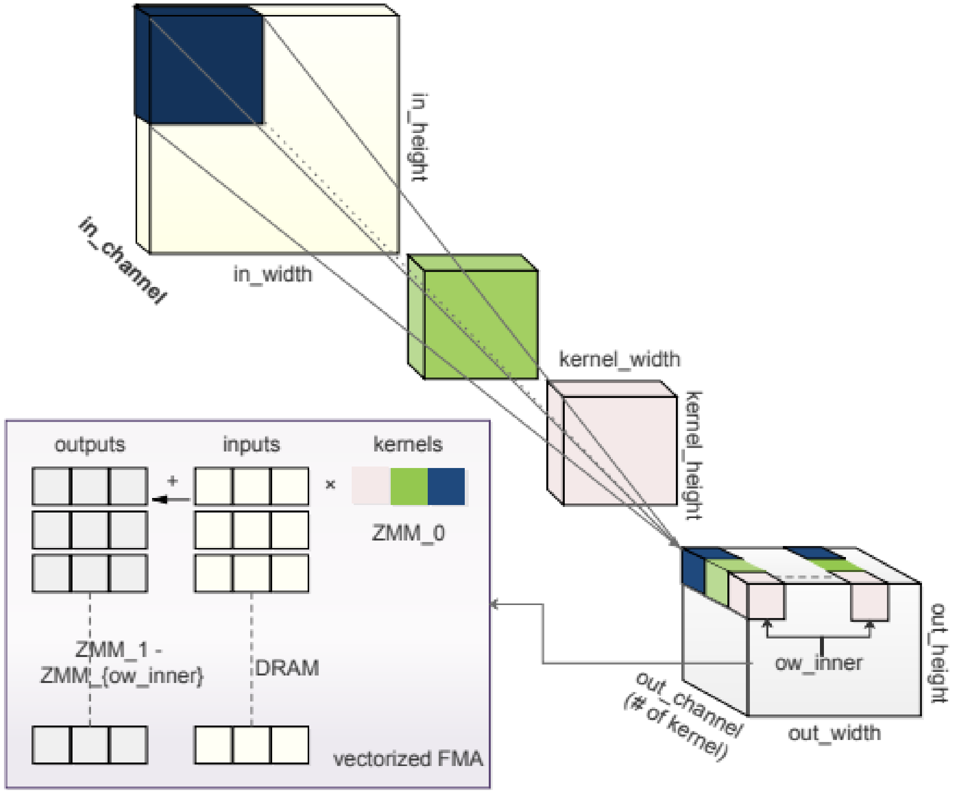
CONVOLUTION OPTIMIZATION ON CPU: DATA LAYOUT

N : batch size
C : channels number
H : feature map height
W : feature map width





CONVOLUTION OPTIMIZATION ON CPU: DATA LAYOUT



Algorithm 1 CONV operation algorithm via FMA

PARAM: $x > 0$ s.t. *in_channel* mod $x = 0$
PARAM: $y > 0$ s.t. *out_channel* mod $y = 0$
PARAM: $reg\_n > 0$ s.t. *out_width* mod $reg\_n = 0$
PARAM: $unroll\_ker \in \{True, False\}$
INPUT: *IFMAP* in NCHW[x]c
INPUT: *KERNEL* in KCRS[x]c[y]k
OUTPUT: *OFMAP* in NCHW[y]c
for each disjoint chunk of *OFMAP* **do** ▷ parallel
  for ow.outer:= $0 \rightarrow out\_width/\boxed{reg\_n}$ **do**
    Initialize V_REG_1 to $V\_REG_{reg\_n}$ by $\vec{0}$
    for ic.outer:= $0 \rightarrow in\_channel/\boxed{x}$ **do**
      for each entry of *KERNEL* **do** ▷ (opt) $\boxed{\text{unroll}}$
        for ic.inner:= $0 \rightarrow x$ **do**
          $vload(KERNEL, V\_REG_0)$ ▷ $\boxed{\text{y}}$ floats
          for i:= $1 \rightarrow reg\_n + 1$ **do** ▷ unroll
            $vfmadd(IFMAP, V\_REG_0, V\_RE$
          end for
        end for
      end for
    end for
    for i:= $1 \rightarrow reg\_n + 1$ **do**
      $vstore(V\_REG_i, OFMAP)$
    end for
  end for
end for