Pros and Cons of Executable Neural Networks for Deeply Embedded Systems

Matheus F. Ferraz

Embedded Software Systems Group

Hamburg, 21. September 2023

Table of Contents

Introduction

- Edge devices often come with limited computational resources
  - RAM
  - Processing power
  - Energy

- Especially in IoT contexts
- Many applications require low-latency inference
- Among optimizations to improve performance:
  - Quantization
  - Pruning
  - Specialized hardware accelerators
  - Ahead-of-Time Compilation

Source: jeferrb/Pixabay

Introduction

Ahead-of-Time Compilers

- MicroTVM
- XLA
- Glow

- All offer graph optimization, operator fusion, quantization and hardware specific optimizations
- All under heavy development

µtvm

GLOW

Approach - Executable Data Structures (EDs)

- Traversal optimization to reduce the time it takes to navigate through a data structure
- Uses node-specific code:
  - each node stores both data and instructions of how to traverse the structure starting from itself
- Eliminates intermediate functions calls and lookup operations, resulting in improved efficiency

3.2.3 Executable Data Structures

The executable data structures method reduces the traversal time of data structures that are frequently traversed in a preferred way. It works by storing node-specific traversal code along with the data in each node, making the data structure *self-traversing*.

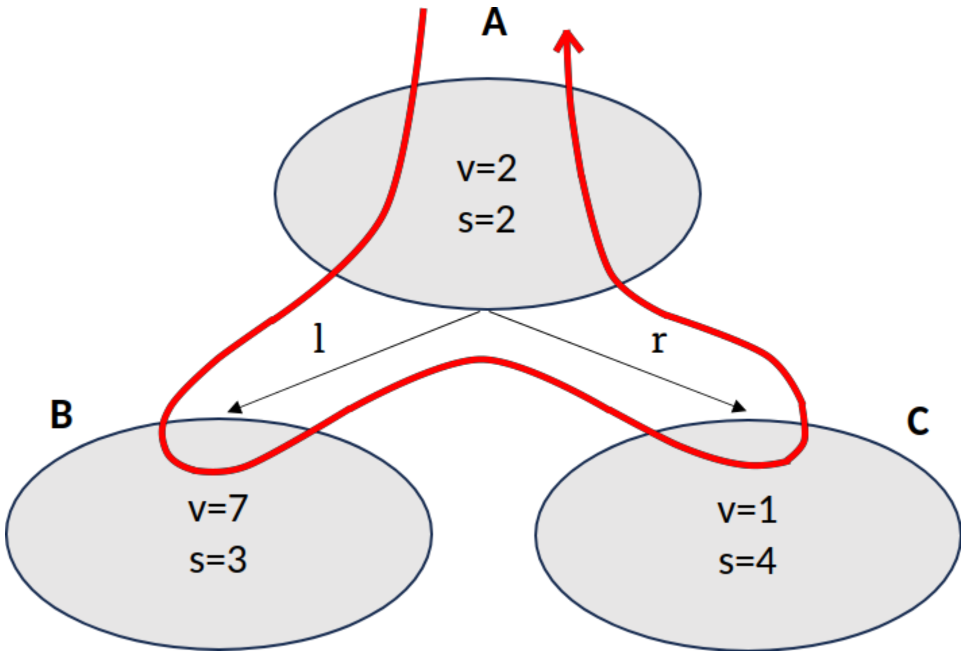
Consider an active job queue managed by a simple round-robin scheduler. Each element in the queue contains two short sequences of code: `stopjob` and `startjob`. The `stopjob` saves the registers and branches into the next job's `startjob` routine (in the next element in queue). The `startjob` restores the new job's registers, installs the address of its own `stopjob` in the timer interrupt vector table, and resumes processing.

An interrupt causing a context switch will execute the current program's `stopjob`, which saves the current state and branches directly into the next job's `startjob`. Note that the scheduler has been taken out of the loop. It is the queue itself that does the context switch, with a critical path on the order of ten machine instructions. The scheduler intervenes only to insert and delete elements from the queue.

Original definition of Executable Data Structures found in A. Massalin (1992)

Approach - Executable Data Structures (EDs)

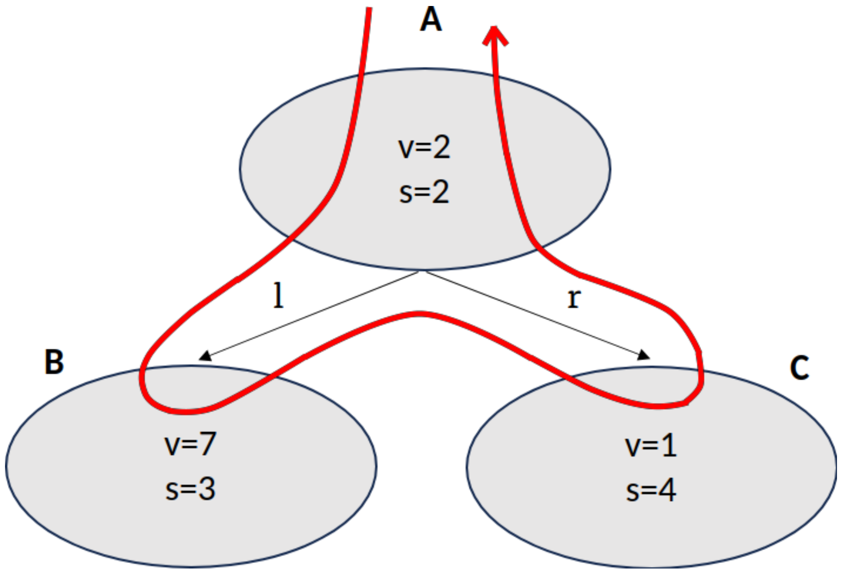
A basic EDs example




```
struct node {
  int v; s; node *l, *r;
  int msum(int f) {
    return v*f+(l?l->msum(f+s):0)+
               (r?r->msum(f+s):0);
} };
```

Approach - Executable Data Structures (EDs)

A basic EDs example




```
struct node {
  int v; s; node *l, *r;
  int msum(int f) {
    return v*f+(l?l->msum(f+s):0)+
               (r?r->msum(f+s):0);
} };
```

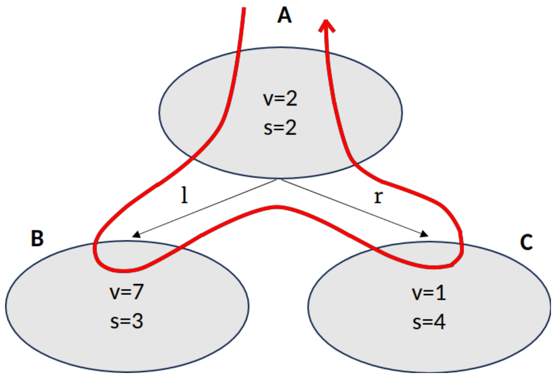
```
struct C {
   static const int v = 1;
   static const int s = 4;
   static int msum(int f) { return v * f; }
};
struct B {
   static const int v = 7;
   static const int s = 3;
   static int msum(int f) { return v * f; }
};
struct A {
   static const int v = 2;
   static const int s = 2;
   static int msum(int f) {
      return v * f + B::msum(f + s) + C::msum(f + s);
   }
};
```

Approach - Executable Data Structures (EDs)

A basic EDs example

- The compiler is able to use strategies to optimize the final code to:

A::msum(F) = v * F + B::msum(F + s) + C::msum(F + s)
A::msum(F) = 2 * F + B::msum(F + 2) + C::msum(F + 2)
A::msum(F) = 2 * F + (7 * F + 14) + (F + 2)
A::msum(F) = 10 * F + 16




```
struct C {
    static const int v = 1;
    static const int s = 4;
    static int msum(int f) { return v * f; }
};
struct B {
    static const int v = 7;
    static const int s = 3;
    static int msum(int f) { return v * f; }
};
struct A {
    static const int v = 2;
    static const int s = 2;
    static int msum(int f) {
        return v * f + B::msum(f + s) + C::msum(f + s);
    }
};
```

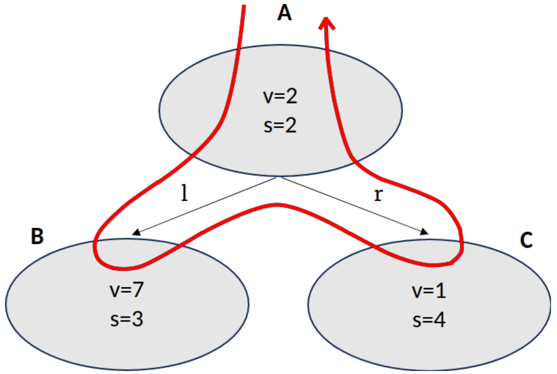
Approach - Executable Data Structures (EDs)

A basic EDs example

- The compiler is able to use strategies to optimize the final code to:

$$A::msum(F) = v * F + B::msum(F + s) + C::msum(F + s)$$
$$A::msum(F) = \mathbf{2} * F + B::msum(F + \mathbf{2}) + C::msum(F + \mathbf{2})$$
$$A::msum(F) = \mathbf{2} * F + (\mathbf{7} * F + \mathbf{14}) + (F + \mathbf{2})$$
$$A::msum(F) = \mathbf{10} * F + \mathbf{16}$$

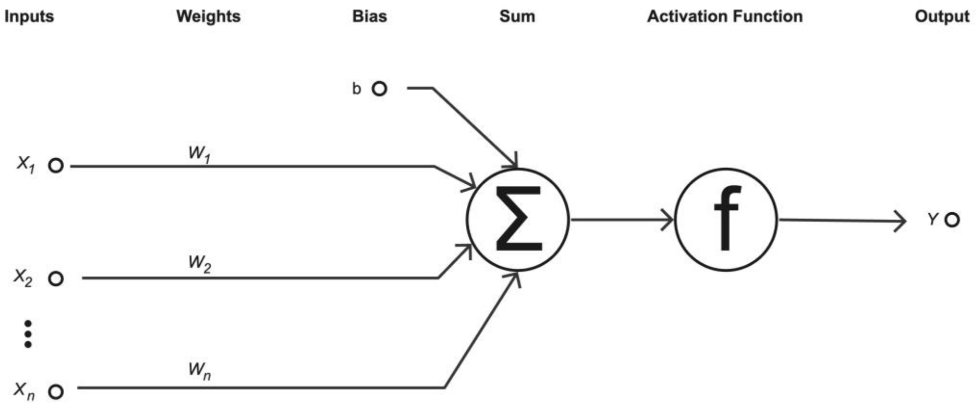
```
struct C {
    static const int v = 1;
    static const int s = 4;
    static int msum(int f) { return v * f; }
};
struct B {
    static const int v = 7;
    static const int s = 3;
    static int msum(int f) { return v * f; }
};
struct A {
    static const int v = 2;
    static const int s = 2;
    static int msum(int f) {
        return v * f + B::msum(f + s) + C::msum(f + s);
    }
};
```



compiler

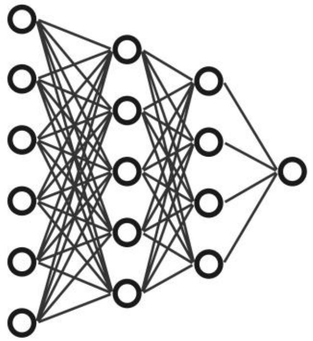
```
int A::msum(int f) {
    return 10*f+16;
} };
```

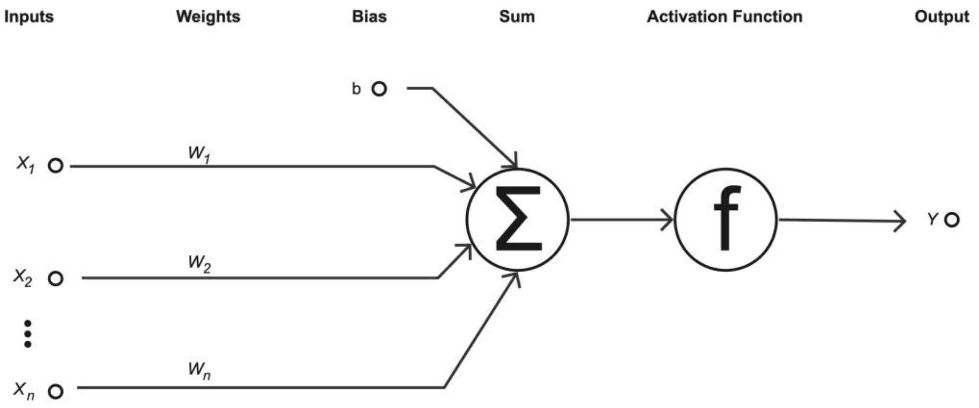
Approach - Executable Neural Network (ExecNN)



$$y = f(w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n + b)$$

Approach - Executable Neural Network (ExecNN)





$$y = f(w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n + b)$$

```
for (int b = 0; b < batches; ++b) {
  for (int out_c = 0; out_c < output_depth; ++out_c) {
    AccumScalar acc = 0;

    for (int d = 0; d < accum_depth; ++d) {
      int32_t input_val = input_data[b * accum_depth + d];
      int32_t filter_val = filter_data[out_c * accum_depth + d];
      acc += (filter_val + filter_offset) * input_val;
    }

    if (bias_data) {
      acc += bias_data[out_c];
    }

    int32_t acc_scaled = MultiplyByQuantizedMultiplier(
      acc,
      output_multiplier,
      output_shift
    );
    acc_scaled = std::max(acc_scaled, output_activation_min);
    acc_scaled = std::min(acc_scaled, output_activation_max);
    output_data[out_c + output_depth * b] =
static_cast<int16_t>(acc_scaled);
  }
}
```

Approach - Executable Neural Network (ExecNN)

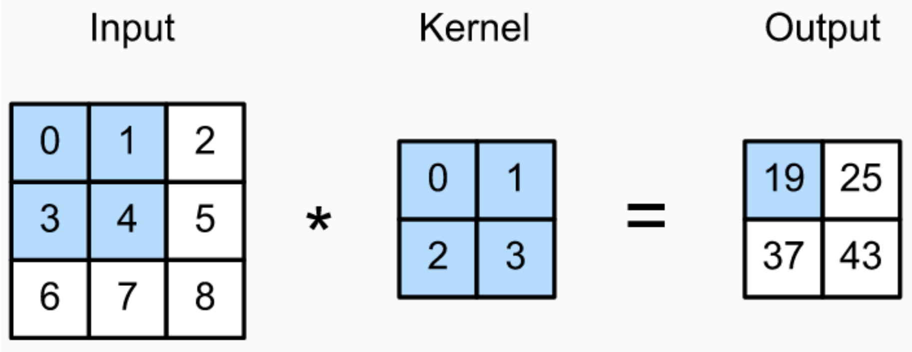
```
for (int b = 0; b < batches; ++b) {
  for (int out_c = 0; out_c < output_depth; ++out_c) {
    AccumScalar acc = 0;

    for (int d = 0; d < accum_depth; ++d) {
      int32_t input_val = input_data[b * accum_depth + d];
      int32_t filter_val = filter_data[out_c * accum_depth + d];
      acc += (filter_val + filter_offset) * input_val;
    }

    if (bias_data) {
      acc += bias_data[out_c];
    }

    int32_t acc_scaled = MultiplyByQuantizedMultiplier(
      acc,
      output_multiplier,
      output_shift
    );
    acc_scaled = std::max(acc_scaled, output_activation_min);
    acc_scaled = std::min(acc_scaled, output_activation_max);
    output_data[out_c + output_depth * b] =
static_cast<int16_t>(acc_scaled);
  }
}
```

Values expressed as constexpr, all of them are known at compile time

```
y[0] = x[0] * weights[1] + x[1] * weights[2] + x[2] * weights[3] + x[3] * weights[4] + bias[1];
y[1] = x[0] * weights[5] + x[1] * weights[6] + x[2] * weights[7] + x[3] * weights[8] + bias[2];
y[2] = x[0] * weights[9] + x[1] * weights[10] + x[2] * weights[11] + x[3] * weights[12] + bias[3];
y[3] = x[0] * weights[13] + x[1] * weights[14] + x[2] * weights[15] + x[3] * weights[16] + bias[4];
...
y[n] = x[0] * weights[m-3] + x[1] * weights[m-2] + x[2] * weights[m-1] + x[3] * weights[m] +
bias[n];
```

ExecNN

Approach - Executable Neural Network (ExecNN)




```
for (int batch = 0; batch < batches; ++batch) {
   for (int out_y = 0; out_y < output_height; ++out_y) {
      for (int out_x = 0; out_x < output_width; ++out_x) {
         for (int out_channel = 0; out_channel < output_depth;
++out_channel) {
            for (int filter_y = 0; filter_y < filter_height; ++filter_y) {
               for (int filter_x = 0; filter_x < filter_width; ++filter_x) {
... }}}}}}
```

```
int8_t convKernel_0_0(int8_t *input, uint16_t x, uint16_t y){
   int32_t acc = 0;
   acc+= weights[0] * input[(0 + x) * $width + 0 + y];
   acc+= weights[1] * input[(0 + x) * $width + 1 + y];
   acc+= weights[2] * input[(0 + x) * $width + 2 + y];
   acc+= weights[3] * input[(1 + x) * $width + 0 + y];
   acc+= weights[4] * input[(1 + x) * $width + 1 + y];
   acc+= weights[5] * input[(1 + x) * $width + 2 + y];
   acc+= weights[6] * input[(2 + x) * $width + 0 + y];
   acc+= weights[7] * input[(2 + x) * $width + 1 + y];
   acc+= weights[8] * input[(2 + x) * $width + 2 + y];
   acc+= bias[0];
   ...
}


void inference_0(int8_t *x){
   output[0]=convKernel_0_0((&x[0]),0,0);
   output[1]=convKernel_0_0((&x[0]),0,1);
   output[2]=convKernel_0_0((&x[0]),1,0);
   output[3]=convKernel_0_0((&x[0]),1,1);
}
```

ExecNN

Evaluation

Load Model → **Generate C++ Code** → **Compile** → **Run**

- Receives a TensorFlow Lite model as input

Evaluation

Load Model → **Generate C++ Code** → **Compile** → **Run**

- A Python script uses the loaded model to create inference code
- Quantizes models parameters (int8)
- NHWC
- Output:
  - Model.h -> header with weights and biases
  - Inference.cpp -> source file with inference functions

Evaluation

Load Model → **Generate C++ Code** → **Compile** → **Run**

- Built with –O3 flags for maximum optimization
- Outputs an elf file

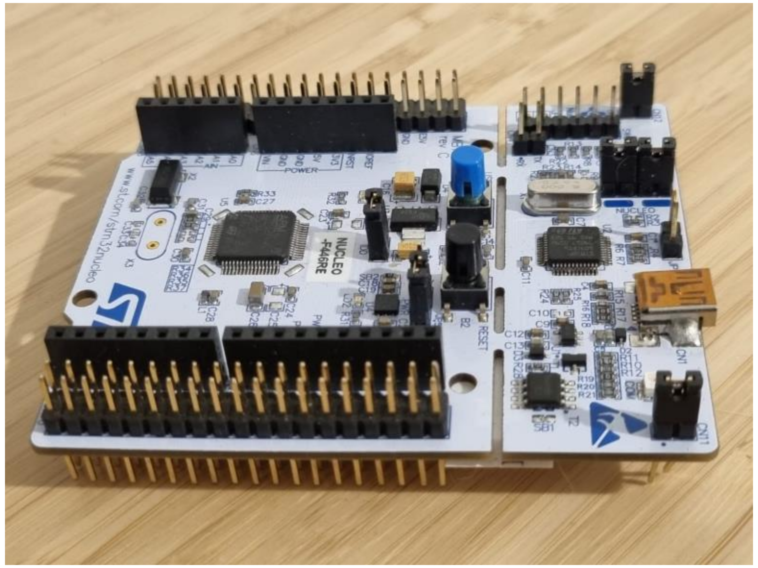
Evaluation

Load Model → **Generate C++ Code** → **Compile** → **Run**

- Deploys on target device
- Main loop performs hundreds of inferences
- Monitor the time taken to perform the task

Evaluation

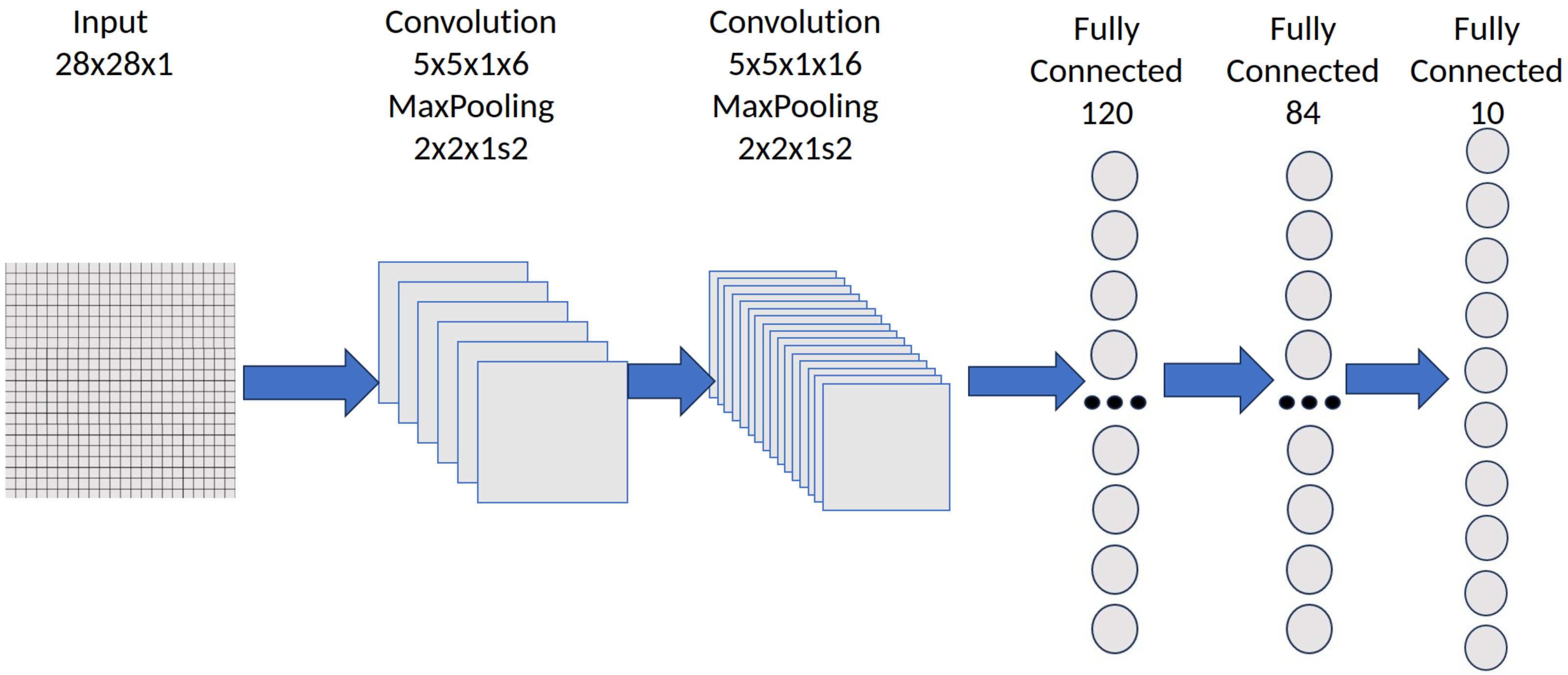
- X86 i7-1185
- STM32F446RE MCU
  - ARM Cortex-M4
  - 128 kB RAM
  - 512 kB Flash
  - 180 MHz



Evaluation

- Lenet-5

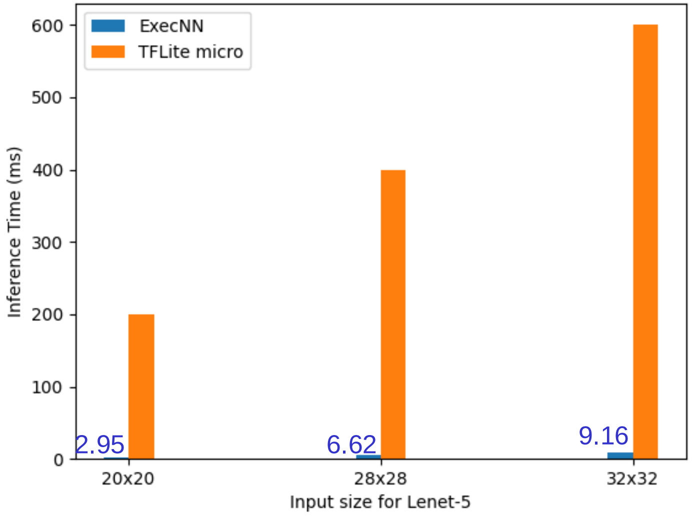
- Supported layers:
  - Fully connected
  - Conv2D
  - Maxpool



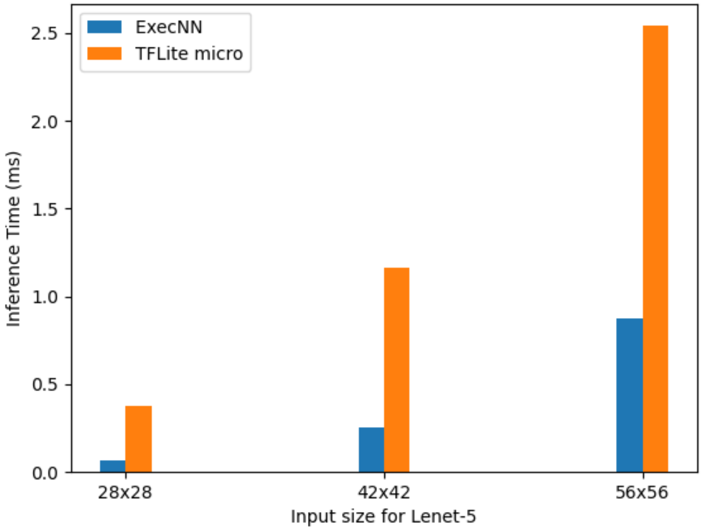
Evaluation

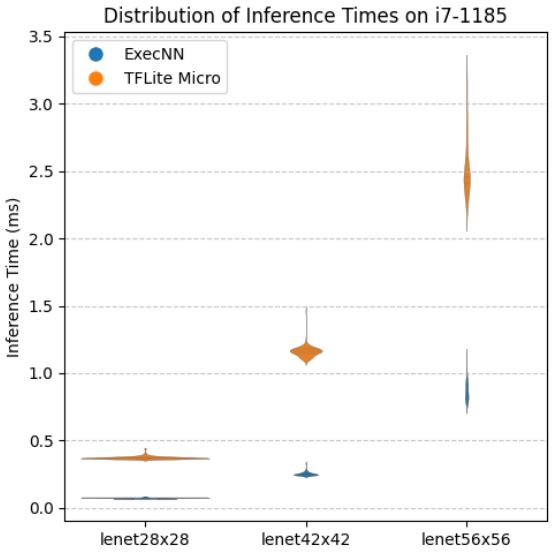
Inference time

- On STM32



- On Intel i7-1185

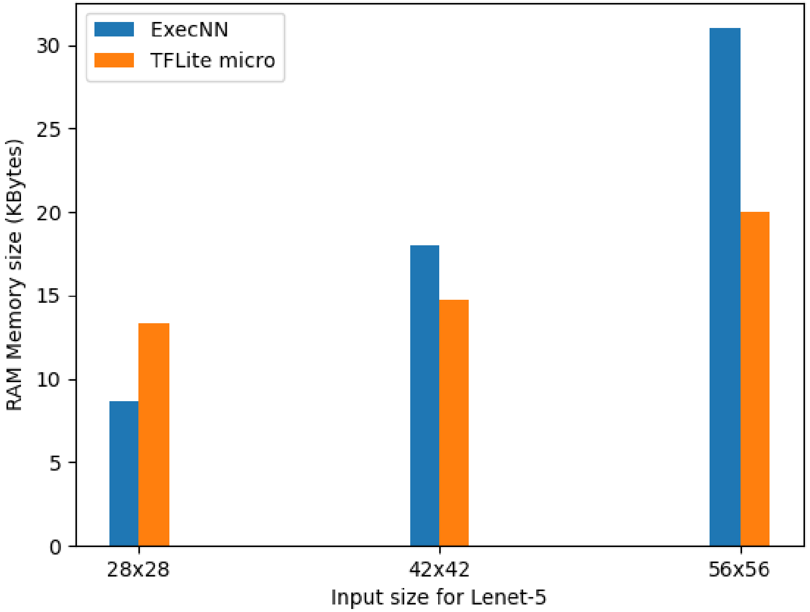




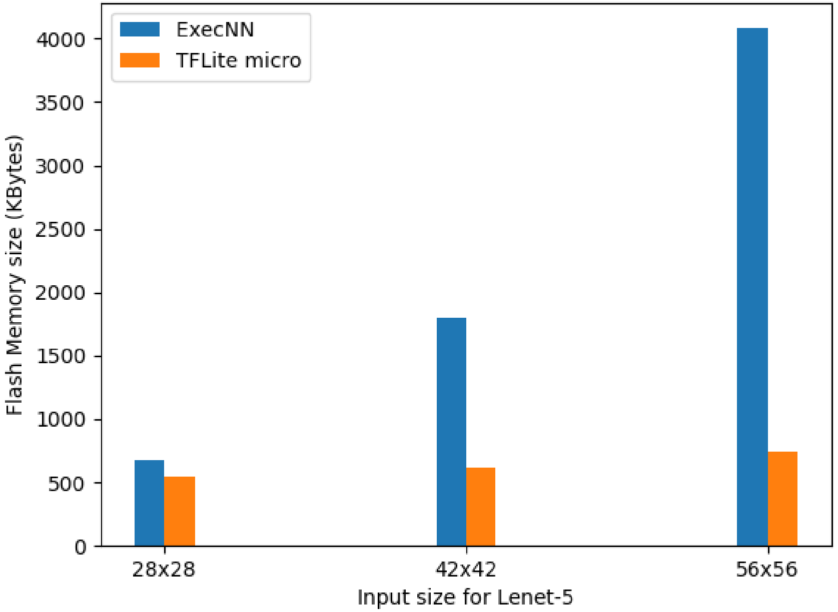
Evaluation

Binary size

- Data/BSS sections

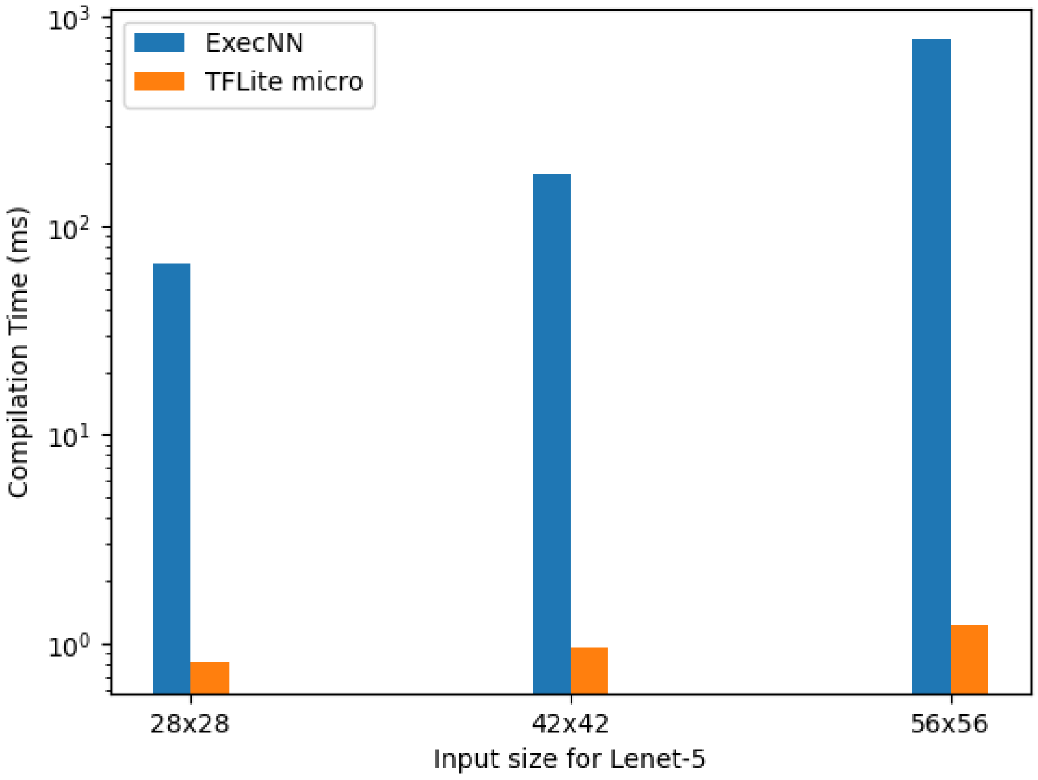


- Text section



Evaluation

Compilation time



	Compile Time (s)		
	LeNet-5 28x28x1	LeNet-5 42x42x1	LeNet-5 56x56x1
ExecNN	66	178	778
TFLite Micro	0.81	0.97	1.23

Conclusions

- Significant reduction in inference time (up to 70 times) compared to a state-of-the-art interpreter
- Could prove beneficial for MCUs and energy-efficient devices
- Exponential increase in memory size (flash) and compile time

Future Work

- Compare with other frameworks capable of AoT compilation (microTVM)
- Adding support for other architectures
- Implement common optimizations(g.e. Pruning)
- Profiling which types of layers most benefit from this approach

Thank you for your attention