UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Artificial Intelligence and Data Engineering

Removing dynamic memory dependencies from tiny_dnn

Project Report

Martina Marino

Roberta Matrella

Academic Year: 2022/2023

Contents

1 Introduction

The goal of this project is to improve the tiny_dnn library by integrating the ETL library (Embedded Template Library) and removing its reliance on dynamic memory allocation. As lightweight deep learning library with just headers, 'tiny_dnn' is intended for embedded devices and situations with limited resources. Its dependence on dynamic memory allocation, however, may present difficulties in systems that have strict memory management guidelines. Our goal is to re-implement the parts of tiny_dnn that need static memory allocation by utilizing the ETL. The ETL offers a set of tools that make it easier to employ fixed-size containers, stable memory management, and predictable behavior. It is particularly made for embedded applications. In order to demonstrate that the changes made provided the expected improvement, the goal was to carry out synthesis with Vitis so that we could evaluate the improvements in terms of energy consumption, resource utilization and timing on IoT devices using Vivado[1].

2 Tiny_dnn library

Tiny_dnn[2] is a C++ deep learning library that has no dependencies and simply contains headers. It is intended for use in embedded systems and Internet of Things devices, among other real-world applications. Because tiny_dnn supports TBB threading and SSE/AVX vectorization, it can achieve 98.8% accuracy on the MNIST dataset in about 13 minutes of training on a Core i7-3520M, making it relatively fast even without a GPU. It is a header-only library that is portable and compatible with any platform that has C++14 support in its compiler. Additionally, tiny_dnn is a great option for anyone learning about neural networks because of its straightforward and approachable implementation.

3 ETL library

The Embedded Template Library (ETL[3]) is a reliable and flexible library created to offer fixed-size and deterministic options to the standard template library (STL) for embedded systems and similar resource-limited settings. The Embedded Template Library (ETL) has been specifically designed to overcome these challenges by utilizing fixed-size containers and deterministic memory management techniques. ETL containers allocate memory statically, either at compile-time or initialization, ensuring that memory usage remains constant and predictable throughout the program's execution. This approach eliminates fragmentation and guarantees that memory is always readily available when required. By steering clear of dynamic allocation, ETL ensures that operations on its containers maintain a consistent, predictable time complexity, which is essential for meeting real-time performance demands. ETL's methodology minimizes the likelihood of memory leaks and allocation errors, thereby streamlining the debugging process and enhancing the

overall stability of the system. The ETL encompasses a variety of fixed-size containers, including vectors, lists, queues, stacks, and maps, which mirror the interface of their STL equivalents but utilize pre-allocated memory pools for storing elements. Furthermore, ETL offers a range of algorithms and utilities that complement these containers, facilitating the development of intricate applications without the need for dynamic memory allocation.

4 Vitis

Xilinx offers Vitis[4] as a complete development platform to facilitate the creation and implementation of applications on FPGA devices. This platform encompasses a variety of tools and libraries that cater to the needs of both software and hardware developers, enabling them to optimize and accelerate their projects. Within the Vitis platform, there are specialized components such as Vitis AI, which focuses on machine learning inference, Vitis HLS for converting C/C++ code into hardware descriptions, and Vitis Accelerated Libraries that provide acceleration solutions for specific domains.

Vitis is an essential tool for this project for several reasons, including its capabilities to optimize and synthesize designs for FPGA (Field Programmable Gate Array) devices. In fact, FPGAs are highly suited for applications requiring deterministic and static memory management. By removing dynamic memory allocation and using static memory through ETL, the designs can be better optimized for FPGAs. Vitis helps in synthesizing these optimized designs efficiently. Also, FPGAs have a fixed amount of resources (e.g., logic blocks, memory), making it essential to manage memory statically and predictably. Vitis assists in ensuring the design fits within the available resources. By ensuring `tiny_dnn` uses static memory, the HLS process becomes straightforward and the resulting hardware more reliable and easier to verify.

5 Implementation

This section outlines the changes made to the tiny_dnn library to remove the dependencies on dynamic memory using the ETL library.

5.1 Main changes

In order to remove dependencies from dynamic memory it was necessary to replace the elements that required it by using the containers provided by the ETL library. In particular, `std::vector` has been identified as the main cause of dependence on dynamic memory. It is designed to dynamically manage memory by allocating and deallocating memory as needed, expensive for devices with tight performance constraints. For this reason, its counterpart of the ETL library, the `etl::vector`, have been used. The `etl::vector` is a fixed capacity vector defined as follows: `etl::vector<typename T,`

size_t SIZE>. Since etl::vector requires specifying the size of the vector at the time of its declaration, it was necessary to define parameters chosen based on the network structure.

```
#define MAX_INPUT_SIZE 25
#define MAX_TENSOR_SIZE 25
#define MAX_OUTPUT_SIZE 3
#define MAX_LAYERS 5
#define MAX_NODES 10
#define MAX_CHANNEL_SIZE 3
#define MAX_CONNECTIONS 10
```

The parameters above have been selected considering a simple 5x5 input image with 3 channels and 3 output classes. In particular, the parameter MAX_INPUT_SIZE represents the value of width*height of the input image, MAX_TENSOR_SIZE is the maximum size of a generic tensor, MAX_OUTPUT_SIZE is the number of classes to predict, MAX_CHANNEL_SIZE is the number of channels the image is composed of. Finally MAX_LAYER_SIZE, MAX_NODES and MAX_CONNECTIONS define the characteristics of the network. The two main important variables to be set are tensor_t and vec_t, in fact an image is considered as a tensor_t made of vec_t of MAX_TENSOR_SIZE float numbers. For this reason they have been redefined as follows:

```
typedef etl::vector<float_t, MAX_TENSOR_SIZE> vec_t;
typedef etl::vector<vec_t, MAX_CHANNEL_SIZE> tensor_t;
```

The test code that was executed to demonstrate the correct operation of the modified network is the following.

```
int test_tiny_dnn() {
    tensor_t image(MAX_CHANNEL_SIZE, vec_t(MAX_TENSOR_SIZE));

    for (size_t j = 0; j < rows; ++j){
        for (size_t i = 0; i < cols; ++i) {
            image[j][i] = static_cast<float>(rand()) / RAND_MAX;
        }
    }

    network<sequential> net;
    net << fc(MAX_TENSOR_SIZE, 3) << tanh() << smax();

    return net.predict(input);
}
```

Since, for resource reasons, it was not in the interest of this project to train a network but only to evaluate the test phase, a fictitious image generated as a tensor_t and a

simple network was used. During the test phase the images are passed one at a time and, for this reason, the image is passed as a one dimension tensor vector. The following functions are the mainly used during inference.

```
tensor_t predict(const tensor_t &in) {
    return fprop(in);
}
```

The `predict` function handles the prediction process calling the `fprop` on the input image.

```
etl::vector<vec_t,MAX_CHANNEL_SIZE> fprop
                        (const etl::vector<vec_t,MAX_CHANNEL_SIZE> &in) {
    return fprop(etl::vector<tensor_t, 1>{in})[0];
}

etl::vector<tensor_t, 1> fprop(const etl::vector<tensor_t, 1> &in) {
    return net_.forward(in);
}
```

The function `fprop` is responsible for forward propagating the input through the network for one sample. It converts the input tensor to a format suitable for layer-wise processing and calls the `forward` method on the network object.

```
etl::vector<tensor_t,1> forward(const etl::vector<tensor_t,1> &first)
override {
    etl::vector<etl::vector<const vec_t *,1>,MAX_CHANNEL_SIZE> reordered_data;
    reorder_for_layerwise_processing(first, reordered_data);

    nodes_.front()->set_in_data(&reordered_data[0], 1);

    for (auto l : nodes_) {
        l->forward();
    }
    etl::vector<const tensor_t *, 1> out;
    nodes_.back()->output(out);

    return normalize_out(out);
}
```

This function performs the core forward propagation for the entire network. In particular, it reorders the input data for layer-wise processing using `reorder_for_layerwise_processing`, it sets the input data for the first node of the network, iterates through all nodes in the network, calling their `forward` method, and finally collects the output from the last node and normalizes it.

```
// transform indexing so that it's more suitable for per-layer operations
// input:  [sample][channel][feature]
// output: [channel][sample][feature]
void reorder_for_layerwise_processing(
        const etl::vector<tensor_t, 1> &input,
        etl::vector<etl::vector<const vec_t *,1>,MAX_CHANNEL_SIZE> &output) {
    size_t sample_count  = input.size();
    size_t channel_count = input[0].size();
    output.resize(channel_count);
    for (size_t i = 0; i < channel_count; ++i) {
        output[i].resize(sample_count);
    }
    for (size_t sample = 0; sample < sample_count; ++sample) {
        assert(input[sample].size() == channel_count);
        for (size_t channel = 0; channel < channel_count; ++channel) {
            output[channel][sample] = &input[sample][channel];
        }
    }
}
```

This utility function reorders the input data to facilitate layer-wise operations. It transforms the data from `[sample][channel][feature]` format to `[channel][sample][feature]` format.

```
void forward() {
    // the computational graph
    fwd_in_data_.resize(in_channels_);
    fwd_out_data_.resize(out_channels_);

    // Organize input/output vectors from storage (computational graph).
    // Internally ith_in_node() will create a connection/edge in the
    // computational graph and will allocate memory in case that it's not
    // done yet.
    for (size_t i = 0; i < in_channels_; i++) {
      fwd_in_data_[i] = ith_in_node(i)->get_data();
    }

    // resize outs and stuff to have room for every input sample in
    // the batch
    set_sample_count(fwd_in_data_[0]->size());

    // Internally ith_out_node() will create a connection/edge to the
    // computational graph and will allocate memory in case that it's not
```

```
    // done yet. In addition, gradient vector are initialized to default
    // values.
    for (size_t i = 0; i < out_channels_; i++) {
      fwd_out_data_[i] = ith_out_node(i)->get_data();
      ith_out_node(i)->clear_grads();
    }

    // call the forward computation kernel/routine
    forward_propagation(fwd_in_data_, fwd_out_data_);
}
```

This function represents the forward propagation method for a single layer within the network. Key steps include resizing the input and output data vectors to match the number of channels, organizing input and output vectors from the computational graph, setting the sample count and initializing gradient vectors, calling `forward_propagation` to execute the actual computation for the layer.

```
void forward_propagation(
            const etl::vector<tensor_t *, MAX_CHANNEL_SIZE> &in_data,
            etl::vector<tensor_t *, MAX_CHANNEL_SIZE> &out_data) override {

    // forward fully connected op context
    fwd_ctx_.set_in_out(in_data, out_data);

    fwd_ctx_.setParallelize(layer::parallelize());
    fwd_ctx_.setEngine(layer::engine());

    // launch fully connected kernel
    kernel_fwd_->compute(fwd_ctx_);
}
```

This function executes the forward propagation computation for the fully connected layer performing the following steps: sets the context for input and output data, configure parallelization and execution engine settings and invokes the kernel's `compute` method to perform the actual computation of the forward pass. After these steps, the code of the `tanh` and `softmax` layers is executed. The activation layers functions to be called did not involve any memory allocation changes.

5.2 Project run

The main objective of the project was to run the project without allocation of dynamic memory and make the synthesis on Vitis in order to evaluate the improvements in terms of energy consumption of devices through the use of Vivado software. However, Vitis was

not able to carry out the synthesis using the network previously presented by us, even if very simple.

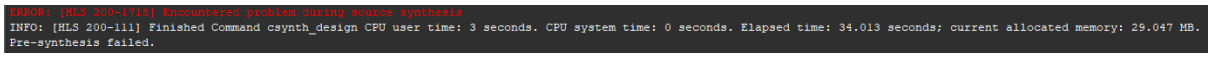

Figure 1: Vitis synthesis

Despite attempts to reduce the parameters constituting the size of the vectors, the program was not able to complete the synthesis. For this reason, not being able to evaluate the changes made through Vivado, it was performed a local execution. The objective was thus to identify any errors in the execution phase not highlighted by Vitis. Below are the results obtained from the execution of the test code.

As expected, since there was no training phase, the predictions provided by the network are random. While running locally it was possible to get the output of the network. For this reason we deduced that the synthesis with Vitis had failed due to limitations of the platform. However, the local execution made it possible to assess the effectiveness of the changes made.

```
martimarino@PC-MM:/mnt/c/Users/marti/Downloads/tinyDNN-master-tinyDNNAlt-src-tinyDNNAlt_vanilla/tinyDNN-master-tinyDNNAlt-src-tinyDNNAlt_vanilla/tinyDNNAlt/src/tinyDNNAlt_vanilla$ g++ -g -I./ -fexceptions -Wno-unknown-pragmas prova.cpp
martimarino@PC-MM:/mnt/c/Users/marti/Downloads/tinyDNN-master-tinyDNNAlt-src-tinyDNNAlt_vanilla/tinyDNN-master-tinyDNNAlt-src-tinyDNNAlt_vanilla/tinyDNNAlt/src/tinyDNNAlt_vanilla$ ./a.out
Test image:
0.840188 0.394383 0.783099 0.79844 0.911647 0.197551 0.335223 0.76823 0.277775 0.55397 0.477397 0.628871 0.364784 0.513401 0.95223 0.916195 0.635712 0.717297 0.141603 0.606969 0.0163006 0.242887 0.137232 0.804177 0.156679 0.400944 0.12979 0.108809 0.998924 0.218257 0.512932 0.839112 0.61264 0.296032 0.637552 0.524287 0.493583 0.972775 0.292517 0.771358 0.526745 0.769914 0.400229 0.891529 0.283315 0.352458 0.807725 0.919026 0.0697553 0.949327 0.525995 0.0860558 0.192214 0.663227 0.890233 0.348893 0.0641713 0.020023 0.457702 0.0630958 0.23828 0.970634 0.902208
 0.85092 0.266666 0.53976 0.375207 0.760249 0.512535 0.667724 0.531606 0.0392803 0.437638 0.931835 0.93081
Input size: 25

** Network output **
Output size: 1
Prob. classes: 0.333333 0.333333 0.333333
```

Figure 2: Testing results

Since Vitis failed to synthesize the network, although simple, we deduced that the goal of our project was not attainable and, for this reason, no further experiments with more complex networks were carried out.

6 Conclusions

The initial objective of the project was to evaluate the improvements made in terms of energy consumption by removing dependencies from the dynamic memory of the `tiny_dnn` library. Since the synthesis with Vitis proved to be impracticable, we were able to demonstrate that the intended objective was unachievable. Although it was not possible to achieve the initial objective, it was nevertheless possible to demonstrate, through a local execution of the project, the correctness of the changes previously made.

References

[1] Amd vivado. https://www.xilinx.com/products/design-tools/vivado.html.

[2] tinydnn. https://gitlab.jsc.fz-juelich.de/epi-wp1-public/tinyDNN.

[3] Embedded template library. https://etlcpp.com/.

[4] Vitis. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html.