10

Deploying a Convolutional Neural Network on Edge MCU and Neuromorphic Hardware Platforms

Simon Narduzzi[1], Dorvan Favre[1,2], Nuria Pazos Escudero[2] and L. Andrea Dunbar[1]

[1]CSEM, Switzerland
[2]HE-Arc, Switzerland

Abstract

The rapid development of embedded technologies in recent decades has led to the advent of dedicated inference platforms for deep learning. However, unlike development libraries for the algorithms, hardware deployment is highly fragmented in both technology, tools, and usability. Moreover, emerging paradigms such as spiking neural networks do not use the same prediction process, making the comparison between platforms difficult. In this paper, we deploy a convolutional neural network model on different platforms comprising microcontrollers with and without deep learning accelerators and an event-based accelerator and compare their performance. We also report the perceived effort of deployment for each platform.

Keywords: neuromorphic computing, IoT, kendryte, DynapCNN, STM32, performance, comparison, benchmark.

10.1 Introduction

Edge computing is a key tool in harnessing the possibilities of artificial intelligence. Some advantages of edge over cloud processing are low latency, allowing real-time application and connectivity independence, i.e., no need

of infrastructure and no transmission of sensitive data, allowing improved security and privacy-preserving applications. However, perhaps the most important and as yet untapped potential of edge computing is in the low power possibilities. Low power allows always-on IoT devices for seamlessly integrated intelligent systems. Creating edge-based IoT devices often requires limited hardware resources, both in terms of power and on-device memory. Today's intelligence is mainly based on Deep Learning (DL) networks which are power and memory hungry. This conflict has resulted in several emerging technologies and platforms to perform efficient inference at the edge.

Established companies have both targeted the IoT device by creating ultra-low-power processors (Intel Loihi, STM32 Cortex-M4), but there are also several other innovative platforms such as DynapCNN[1] and Kendryte K210[2] specialized for deep neural network inference with a very little power budget. The specialized nature and variety of products and platforms require platform-specific software tools, making the deployment of one model on several platforms cumbersome and creating a barrier to technology adoption. Moreover, the lack of hardware standardization coupled with the necessary customization of the software makes it difficult to compare, and thus choose, the best technology.

To remove this barrier, it is essential to facilitate access to platforms to non-hardware experts. Indeed, the success of DL is essentially linked to the acceleration provided by graphical processing units (GPUs). Currently, only a very small proportion of users have mastered the CUDA programming language used by the majority of GPUs. In most DL libraries, mobilization of the necessary resources can be called in a single command line, without the user having to understand the technology behind it. This kind of single instruction would empower the data scientists in the porting to edge devices.

In this short paper, we give a brief summary of works that address the challenges of implementing DL on different hardware platforms. Initially, we present our results on a basic neural network deployment on edge devices, and then we compare the performance of 3 selected devices. Finally, we describe the lessons learned and present solutions to facilitate the deployment of these models in the future.

10.2 Related Work

Benchmarking low-resource platforms is a necessary process to select the best platforms to embed algorithms. It is a tricky procedure, as the performance of a platform depends on several aspects: the available memory and

processing units, the technology of the hardware, and the frameworks and tools used during the deployment of the models to benchmark. To harmonize the performance assessment, benchmarking suites such as TinyMLPerf [3] have been created. Recently, a benchmarking suite has been developed for event-based neuromorphic hardware [4]. However, both these solutions still need manual adaptation of the code to run on new platforms. While the benchmarking gives good insights about which and why to select a certain platform. It still remains the question of how to use the benchmarking tools itself. Each platform comes with its own SDK, conversion tools, and constraint of utilization, which in turn limits the possibility of comparing the platforms between them.

Today, many benchmarks are therefore performed on just a few hardware platforms and comparing only a single use-case, as alternatives are more cumbersome. Furthermore, it is easier to benchmark and compare platforms from the same constructor, as the deployment pipelines are usually similar between devices. In this regard, standard architectures LeNet-5 and ResNet-20 have been benchmarked on a few STM32 boards [5]. Machine learning algorithms have also been compared on Cortex-M processors [6][7]. Some efforts of cross-constructor benchmarking have also been made. For example, a recent work deployed a gesture recognition and wake-up words application on an Arduino Nano BLE and a STM32 NUCLEO-F401RE [8] using a convolutional neural network.

While the above research focuses on the established STM32 Cortex-M based MCUs, some emerging processors are also explored [9], but the research in this domain remains scarce. Furthermore, the deployment pipelines are not documented, which limits the reproducibility of the results. In our research, we deploy a single neural network on three different platforms and observe their performance. We also highlight the difference between the deployment pipelines of each constructor, and we perform a qualitative study of the easiness of deployment on each system.

10.3 Methods

In this section, we present the selected task and associated experimental setup, and a method to evaluate the effort of the deployment.

10.3.1 Neural Network Deployment

In our experiment, we use 3 different boards. We select boards from different constructors to show the (large) variety of tools and processing available in

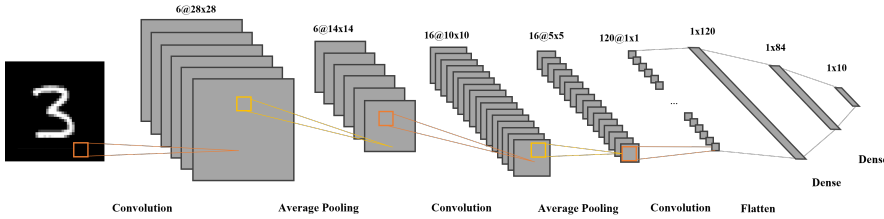

Figure 10.1 Illustration of LeNet-5 architecture.

edge devices today. These sample devices are a very small subset of the large variety of devices today, but they show that with only three different board manufacturers, an extensive adaptation of the deployment pipeline is necessary. The selected 3 devices for our experiments are the following: a Kendryte K210 from Canaan, a dual-core RISC-V processor with floating-point units; an STM32L4R9 from STMicroelectronics (ST) with an ARM Cortex-M4 core also including floating-point unit, and SynSense DynapCNN, an event-based processor. Table 10.1 summarizes the major differences between these platforms.

10.3.1.1 Task and Model

We tested the selected platforms on a simple LeNet-5 [10] networks trained on MNIST, which architecture is displayed in Figure 10.1. This architecture, composed of convolutions layers, average pooling and dense layers, is compatible with all selected platforms. The architecture was trained for 30 epochs with a learning rate $1e-4$. Tensorflow 2.9.1 was used to define the H5 model running on the Sipeed and ST boards, while PyTorch 1.11.0 was used for DynapCNN. Unfortunately, our efforts to transfer the weights from the Tensorflow model to the PyTorch failed, and we had to train the models separately. The Keras and PyTorch models reached an accuracy of 99.44% and 99.38% on the train set, respectively. We perform inference on the first 1000 images of the test dataset.

10.3.1.2 Experimental Setup

For each platform, we used the latest tools available at the time at which this article was written.

Kendryte K210

The Kendryte K210 is used with the Sipeed MaixDock M1. The Neural networks embedded in this device were converted from Keras H5 file format,

using Tensorflow 2.9.1 and associated TFLite. The firmware version of the Kendryte is 0.6.2, and the version of the NNCase package used for conversion is 0.2.

STM32L4R9

The STM32L4R9 board with an Arm Cortex-M4 core processor from ST is programmed in C. Due to the complexity of hardware initialization, ST provides a tool, STM32CubeMX 6.5.0, which automatically generates an initial C project for a specific board. The tool X-CUBE-AI 7.1.0 converts TFLite models into C files which are, alongside the X-CUBE-AI inference library, added to the project. The Keras H5 file network is converted to TFLite format using Tensorflow 2.8.2 and Python 3.6. Gcc-arm-none-eabi 15:10.3-2021.07-4 and Make 4.2.1 are used to compile the whole project, and STM32CubeProgrammer 2.10.0 is used to upload the binaries on the device.

DynapCNN

The SynSense DynapCNN processor was programmed using Python 3.7.13 with PyTorch 1.11.0, Sinabs 0.3.3 (and underlying Sinabs-DynapCNN 0.3.1.dev3), and Samna 0.14.33.0 libraries. The neural network is written in PyTorch and converted to a spiking version using Sinabs, while Samna is used to map the network to the hardware. The inputs are presented to the network using a preprocessing function that generates spikes[1] from random sampling of the image, using the following function, where tWindow is the duration of the spiking frame and img has shape [channels, width, height]:

```
def to_spikes(img, tWindow=100):
    rnd = (np.random.rand(self.tWindow, *img.shape)
    img = rnd < img.numpy()/255.0).astype(float)
    return torch.from_numpy(img).float()
```

During our simulation, we found 100 timesteps to be sufficient to reach equivalent accuracy between the spiking and non-spiking version of MNIST.

10.3.1.3 Deployment

For standalone platforms, the network was converted and uploaded to the platform. For Kendryte, the inference script was written such that the model

[1] Spikes are binary events (on or off) distributed in input space and time.

Table 10.1 Relevant technical specifications of the devices (from constructor websites).

Board	Kendryte K210	STM32L4R9	DynapCNN
Processor ISA	Dual-core RISC-V 64b	ARM Cortex-M4	Event-based
Power Consumption	300mW	66mW	1mW
Max Frequency (MHz)	900	120	-
TOPS/W	3.3	-	-
Standalone	Yes	Yes	No
Event-based	No	No	Yes
Language	MicroPython	C	Python

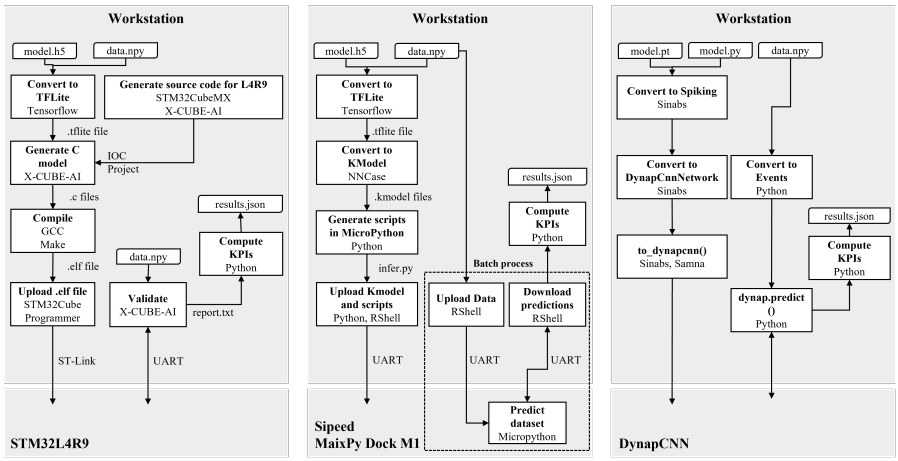

Figure 10.2 Deployment pipelines for all platforms. From left to right: STM32L4R9, Kendryte K210 and DynapCNN. For DynapCNN, the pipeline is contained in a single Python script, while the other relay on external languages and tools.

is loaded at the beginning of the script and processes images one by one. The images are transmitted via serial communication and inferred by inference script. In X-CUBE-AI, this is automatically done, while Kendryte requires a script that sends batches of images and obtains the predictions. For DynapCNN, the images are predicted by sending the corresponding events to the device and reading the output events from the buffer of the board.

The prediction time is provided automatically by the X-CUBE-AI platform, while Kendryte requires to time the prediction manually. In the MicroPython script used for inference on Kendryte, we put a counter around the line performing the inference. For DynapCNN, the reported times corresponds to the timestamp of the first output event and the final output event, respectively. Both times are averaged over the test samples. The computation of the key performance indicators (accuracy, mean time) is performed offline. Figure 10.2 illustrates the pipelines for all platforms.

10.3.2 Measuring the Ease of Deployment

One of the major criteria for the adoption of a product is the ease of use, meaning how much one user is autonomous in using the device. This highly depends on the user skills, but also on the quality of the documentation. For embedded machine learning, the documentation should explicitly describe the procedure to deploy a model once the user receives the new platform. We have identified 5 different phases that are required when using a microcontroller product for AI acceleration.

- **Acquisition (A)**: this phase comprises the effort needed to place an order for the device and the time necessary to ship the device. A small effort would correspond to ordering the platform from a website and receiving it within the next week. A large effort requires to contact the company by phone or email and wait for two month to receive the device.
- **Setup (S)**: this phase comprises the effort needed to install the required environment. A small effort would require installing a python package from pip or an executable available from the constructor website. A large effort requires installing multiple packages which versions depend on the firmware of the device or the version of Python packages used to train the model, as well as dependencies on external tools.
- **Getting started (G)**: this phase is the effort needed to replicate the examples given in the documentation. A small effort would correspond to a full deployment example done within one hour. A large effort would require support from the constructor.
- **Model preparation (M)**: this phase comprises the effort needed to convert a PyTorch/Tensorflow model to the proprietary format of the device. A small effort would correspond to a single command line with arguments. A large effort corresponds to manually writing the neural network in the proprietary format and transferring the weights, with limited help from the conversion tool, or requiring intervention from the constructor.
- **Inference (I)**: this phase comprises the effort needed to perform inference once the model is embedded to the device. A small effort would correspond to a single command line or instruction to perform inference, a medium effort requires writing an inference script and deploying it manually on the hardware platform. A large effort would require intervention from the constructor.

Each phase is assigned with a number between 1 and 5. The total score represents the complexity of deployment. A low value (5) corresponds to a

small effort necessary to deploy a model on a never-used platform, while 25 corresponds to a large effort.

10.4 Results

In this section, we present the results and metrics recorded for each platform, and the effort perceived by the team to perform the experiments.

10.4.1 Inference Results

The models were successfully deployed on all platforms. Table 10.2 summarizes the results on the 1000 first samples of MNIST test dataset. It can be observed that the balanced accuracy is not homogeneous between the platforms. This difference is certainly caused by the different transformations affecting the models during the deployment (conversion). While we initially tried to deploy full-precision models and a quantized version of them, we only had time to deploy it on the ST platform. The evaluation of quantized-aware trained models and evaluation DynapCNN and Kendryte K210 using integer weights is a future work. The models run faster when using 8-bit integer precision on STM32 (even if the platform is made to compute 32-bit floats). The Kendryte K210 is the fastest to compute synchronous frames while DynapCNN is the fastest to provide a result in a 32-bit precision, with 98.79% precision using only the first spike[2]. Unfortunately, only the DynapCNN provides an estimation of the energy consumption, obtained with Sinabs by computing the average number of synpatic operations over the course of the simulations. All the metrics are averaged over the test partition.

Table 10.2 Results on MNIST dataset for all platforms. For the DynapCNN, we report the accuracy and latency for the first spike prediction and over the entire simulation.

Platform	Kendryte K210	STM32L4R9		DynapCNN
Bit Precision	float-32	float-32	int-8	float-32
Size (KB)	94.2	359.2	90.5	-
Accuracy	97.23%	98.26%	94.07%	98.79% / 99.09%
Latency (ms)	54.17	80.82	36.23	41.3 / 294.9
Energy (µJ)	-	-	-	144.5

[2]Some samples (with indices [18, 247, 493, 495, 717, 894, 904, 947] in test set) did not produce any spikes for an unknown reason. In that case, we removed the associated labels and compute the balanced accuracy on the 992 remaining samples.

Table 10.3 Perceived effort for each stage of the inference. 1: small, 5: large.

Board	A	S	G	M	I	Total
Kendryte	1	3	2	3	3	12
STM32L4R9	1	2	4	3	2	12
DynapCNN	3	1	3	1	1	9

10.4.2 Perceived Effort

Table 10.3 summarizes the team perceived effort for each of these phases in a qualitative manner. We observe a high variation in the effort perceived for each platform. The model preparation phase seems to be critical. In all the platforms, this phase is perceived as requiring a great effort. Kendryte K210 and STM32L4R9 require the most human intervention to build a complete deployment pipeline, while the deployment pipeline of DynapCNN is automated.

10.5 Conclusion

Although the development of embedded machine learning holds great promise, the lack of consistency and standardization across devices makes development extremely platform-dependent. Deploying a model on these devices requires to use of low-level tools, such as C language. However, most models are developed using (high-level) Python-based tools. The deployment process of a model therefore requires adaptation of the model from Python to C, which is time-consuming and is prone to errors and artifacts in the final implementation. Platform providers are aware of this problem and have started putting effort into facilitating the deployment by providing automated tools and interfaces with DL frameworks. Specifically, for the platforms used in these experiments, Sipeed has ported MicroPython to the Maix Dock, allowing to write code close to the one used to train the model; SynSense provides a library that allows interaction with the DynapCNN directly from a Python script, and allow simulation of the model before deployment, to get a quick idea of performance. Finally, the well-established ST-Microelectronic provides the X-CUBE-AI tool, which, in addition to analyzing the model before deployment, offers the possibility of validating the model on the target and retrieves relevant metrics without writing a single line of code.

However, these tools are recent and standards are not yet established. To promote and accelerate the development of machine learning on embedded interfaces, it is necessary to provide standardized tools accessible to model

developers, where a minimum of knowledge about the platform is required. This will increase the adoption of the technologies. Some points seem essential to facilitate the adoption of low-power technologies, in particular:

- Up-to-date documentation: documents specifying platform schematics, APIs and dependencies on external tools must be carefully maintained.
- The documentation should contain examples for each API call.
- Model conversion tools should be compatible with most deep learning libraries (Tensorflow and PyTorch) and should detail which version and which operations (layers) are supported by each version of the tool. Ideally, conversion tools should be based on community standards, such as the ONNX format.
- Model conversion tools should be automated and provide understandable warnings and error messages.

To reduce the entry barrier for these low-power platforms for developers of Deep Learning models the following interfaces would be beneficial:

- A hardware simulation interface, in order to obtain a quick feedback on the feasibility of deploying the model on the platform, and to provide an interpretable error in case of memory exhaustion or unsupported layer.
- An evaluation of the key performance indicators relevant for edge computing, such as memory consumption, model speed (number of cycles per inference) and energy used during inference.

These interfaces will enable rapid prototyping and comparison of models for the Edge, while providing a solid foundation for iterating and developing new inference techniques.

Acknowledgements

This work is supported through the project ANDANTE. ANDANTE has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876925. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, Belgium, Germany, Netherlands, Portugal, Spain, Switzerland. The authors are responsible for the content of this publication.

References

[1] Q. Liu, O. Richter, C. Nielsen, S. Sheik, G. Indiveri, and N. Qiao. Live demonstration: face recognition on an ultra-low power event-driven

convolutional neural network asic. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[2] Canaan website. Kendryte K210 description page, 2022.

[3] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[4] C. Ostrau, C. Klarhorst, M. Thies, and U. Rückert. Benchmarking of neuromorphic hardware systems. In *Proceedings of the Neuro-inspired Computational Elements Workshop*, pages 1–4, 2020.

[5] L. Heim, A. Biri, Z. Qu, and L. Thiele. Measuring what really matters: Optimizing neural networks for tinyml. *arXiv preprint arXiv:2104.10645*, 2021.

[6] V. Falbo, T. Apicella, D. Aurioso, L. Danese, F. Bellotti, R. Berta, and A. D. Gloria. Analyzing machine learning on mainstream microcontrollers. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society*, pages 103–108. Springer, 2019.

[7] R. Sanchez-Iborra and A. F. Skarmeta. Tinyml-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 20(3):4–18, 2020.

[8] A. Osman, U. Abid, L. Gemma, M. Perotto, and D. Brunelli. Tinyml platforms benchmarking. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society*, pages 139–148. Springer, 2022.

[9] M. de Prado, M. Rusci, A. Capotondi, R. Donze, L. Benini, and N. Pazos. Robustifying the deployment of tinyml models for autonomous mini-vehicles. *Sensors*, 21(4):1339, 2021.

[10] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.