Towards an open and standardized mechanism for deploying ML/AI models to hardware accelerators.

The OAX consortium.

May 2024

Abstract

This position paper introduces the first outlines of OAX; a novel initiative to make AI model deployment easier and more accessible. OAX is composed of a set of standardized computing processes (conversion toolchains) and interfaces (APIs / ABIs) that are jointly aimed to make it extremely easy for developers to run trained AI models on novel AI accelerators (NPUs, FPUs, GPUs, etc., generally referred to as XPUs). We present the high level technical structure of OAX, consisting of OAX conversion toolchain(s), which convert hardware agnostic model specifications (i.e., a model specification in a common format such as ONNX) to a hardware specific model specification, and OAX runtimes; highly efficient modular blocks that are easily integrated within a larger ML/AI pipeline running on an edge device. From a developer perspective, OAX enables moving between various accelerator targets with minimal effort such that any application developer can select the chipset best suited for their applications without incurring large software development cost. We outline the components of OAX, provide examples, and discuss its limitations. In this position paper we also discuss the envisioned organization of the OAX consortium and actively invite members of the community to contribute.

Contents

1 Introduction

The Open, AI Accelerator (OAX) initiative, (see, https://oax-standard.org) intends to provide an easy-to-use, easy-to-expand, *standardized* methodology of adopting specific (edge) AI accelerators such as NPUs, GPUs, FPUs or the like—we will call them XPUs throughout this position paper—in edge AI / ML applications. As such, OAX aims to alleviate the three most prominent hurdles towards using (Vision) edge AI solutions as identified in the March 2024 Edge AI + Vision alliance developer survey [1]:

1. "... the hardware required is too costly" (60%),
2. "... the development (software) is too costly" (57%), and,
3. "... the hardware required is too power hungry" (48%).

By standardizing and effectively decoupling AI/ML pipeline software development from XPU acceleration, OAX makes it easier to switch to more cost or power efficient AI specific hardware and reduces overal pipeline development time.

OAX is designed to make it easy to take a *trained* AI model and execute it (i.e., generate inferences from that model) on novel (edge) AI hardware. For AI solution developers (ISVs) OAX should make it easy to reap the benefits of the new chipsets that are becoming available rapidly without having to worry about the target hardware when setting up and testing their initial AI pipeline. For those designing and bringing to the market new XPUs, OAX is aimed at lowering the barriers of entry by providing a unified way in which – if adhered to – any developer can easily utilize the advantages of the newly introduced XPU.

Please note that the aims of this position paper are threefold:

1. First, we want to detail why we feel a standard mechanism for the deployment of ML/AI models, developed and maintained by our edge AI community, is useful.

2. Second, we want to provide a rough first outline of such a standard. Currently, this should merely be regarded as a *proposal*; details of the standard should be developed by the community in several focus groups and joint working groups.

3. Third, we want to sketch the community driven approach we think is necessary to make the standard a success. Starting from the first focus group, to, potentially, reaching the status of an accredited international standard.

Specifically on the last point, we are currently (May 2024) actively inviting those interested to join our first focus group(s) (please see Section 5 below for details regarding the organizational structure). Please contact us at https://oax-standard.org when you are interested in contributing to the development of OAX.

1.1 The aims OAX

At a high level, OAX aims to make it easier to develop edge AI applications that reap the benefits from newly introduced XPUs. More specifically,

For *AI solution developers*, OAX aims to:

- Provide a unified way of *converting* a trained AI/Ml model that is available in a standardized, generic, format to a specific format that runs on the target hardware. Standardizing the conversion process saves valuable development time whilst it removes the burden of having to learn a new software ecosystem, toolchain, and – in some cases – model training tools required to reap the benefits of a specific XPU. It makes models truly portable.

- Provide a unified way of *executing* the specific model on the target hardware. I.e., OAX provides a standardized ABI/API to load a model, pass input data to this model, run the inference, and retrieve the model output. This standardization makes that developer can build the surrounding pipeline / application without having to worry about the particularities of the chipset used for AI/ML inferences once the application is deployed. This makes pipelines XPU independent.

Jointly, the above allows AI solution developers to easily move between different hardware targets and reap the fruits of new developments in accelerator design that benefit their use case (i.e., use more cost efficient or energy efficient chipsets).

For *manufacturers* of XPUs, OAX aims to:

- Provide a unified way of allowing "access" to their hardware. By adopting OAX, there is a single point of entry for developers looking to test and embrace the new hardware. This lowers barriers of entrance to the market.

- Reduce the need to extensively develop one's own software ecosystem. By adopting OAX, higher level software tools and ecosystems can easily incorporate new hardware without the need for the manufacturer of the accelerator to provide all the parts of a full AI/ML pipeline: accelerator manufacturers can simply focus on their hardware and the core software components needed to run an AI model on their hardware.[1] This increases the overal market size.

Note that OAX should be designed for maximum flexibility for the XPU manufacturer to convert a trained AI model to their preferred format and to – if desirable – keep toolchains, or other core bits of software that are deemed business valuable, proprietary by simply providing compiled versions or version that are accessible in the cloud.

1.2 The scope of OAX

Currently, OAX focusses on providing a unified method of *accelerating* (i.e., (partly) moving to the XPU) trained AI/ML models. *Models*, in our context, are simply (mathematical) functions that are specified using some hardware agnostic description standard: we adopt ONNX [2] for this purpose in our first implementation of OAX.[2]. At its core, ONNX is simply a standardized (file)

[1] At this moment in time, some individual XPU makers, provide an extensive software ecosystem which effectively locks in application developers by needlessly inflating development costs associated with switching hardware. In our view, in the long run, this hinders overall market growth and leads to suboptimal applications. OAX is, in part, designed to be a fully open – both in terms of code and in terms of governance – response to undesirable lock in.

[2] In recent years ONNX has grown to include not just the implementation agnostic model specification, but also a set of model optimization tools and even model runtimes, we discuss relationships of OAX to these extended initiatives in the related work section of this position paper. At this point in the text we treat ONNX simply as a standardized file format to encode a set of computations.

format used to express which computations (or "operations"), on which data, need to be carried out in which order. ONNX describes the computations start-to-end by specifying a Directed Acyclic Graph (DAG), in which nodes represent operations, and edges represent tensors "flowing" from one computation to the other. Throughout we assume that AI/ML models have one or multiple typed tensors as input (I), and one or multiple typed tensors as output (O) .

OAX does *not* facilitate using the accelerator for application/vertical specific tasks such as decoding of a video stream or fast Fourier transform of a vibration sensor; such operations might be performed well by a specific XPU, but they do not fall into the scope of the initial specification which is meant to be application agnostic. Also, OAX does *not* concern model training: we assume the trained model is available in ONNX format.[3]

The diagram below provides a high level abstraction of a common edge ML/AI pipeline found in many applications in which sensor data is,

1. pre-processed,
2. fed to an AI/ML model for inference,
3. its results are post-processed, and,
4. the final output is visualized.

Within OAX we assume such a pipeline runs on a generic CPU (multiple architectures will be supported), whereas the AI model — or, in some cases, individual operations within the AI model, depending on the accelerator manufacturer's choice — is *accelerated* on the XPU using the generic interface of OAX runtime.[4] Figure 2 below provides a simple overview of the initial scope of OAX within the larger AI/ML pipeline. OAX allows for the block labelled "AI/ML model" to be easily moved from CPU to virtually any XPU by providing *a)* the mechanism to convert a generic model description to a specific model description suited for execution on the XPU, and *b)* providing a standardized interface to access the XPU from the process running on the CPU.

Figure 1: Overview of a typical AI/ML pipeline from sensor to application. Although many production pipelines will have more moving parts — potentially multiple models, aggregation, etc. — at a high level it is very often possible to isolate one (or multiple) AI/ML models within the pipeline that would benefit from XPU acceleration.

[3] Our specification of OAX toolchains does allow individual XPU makers to include training or validation data into the conversion process from generic to specific model specification as is often needed for (e.g.,) effective quantization.

[4] Note that we also provide CPU based runtimes to make sure that OAX can also be used on non-XPU enabled hardware for development purposes.

1.3 High level implementation

At a very high level, the envisioned specification of OAX is surprisingly simple. As a starting point, we assume a (trained) AI model $\mathcal{M}$ is available in ONNX format. Hence, there is access to a generic, deployment independent, description of the function that is implemented by AI/ML model: $\mathcal{M}: I \rightarrow O$. From this starting point we postulate that the following two steps are necessary and sufficient to accelerate an AI/ML model (or parts thereof[5]) within the larger pipeline:

1. *Conversion* of the model's generic ONNX model specification to a format that can run on the accelerator. Note that the result might itself be in ONNX format (for example when operations are accelerated one-by-one by the runtime, see below), or the result might be a totally proprietary binary. The core insight is that there is a "conversion" step necessary from generic model description to a model description that is specific and can be (partly) accelerated using the XPU of interest. For the conversion step OAX merely provides a unified way of going from ONNX to the target's specification (whatever this might be) including standardized error handling and formatting. We call the conversion process the **OAX toolchain** and formally the process itself is simply a function that maps the generic model specification into an XPU specific one: $f: \mathcal{M} \rightarrow \mathcal{M}_{xpu}$.

2. *Running* the specific model specification, i.e., generating inferences. Once a specific model specification is available—after conversion—we simply need a unified way to, from a process running on the CPU, pass data to the model, execute its computation, and return the results. We call this the **OAX runtime.** Note that the OAX runtime itself is a process running on the host CPU that is responsible for all interactions with the XPU. Abstractly, the runtime computes $O = \mathcal{M}_{xpu}(I)$.

The above are deliberately—and contrary to some other attempts in this space—two separate steps: The first of the two steps can be executed anywhere: the conversion of a generic AI model to the target specific model can be done in the cloud, on your local machine, or (in the future) using some hosted conversion service. It is often done only once per (edge) AI target. The second step is executed, repeatedly, on the (fleet of) edge device(s) that runs the AI/ML pipeline (see 2). Thus, conversion and running are often two separate processes, carried out on physically separate devices.

Given the availability of a standardized method—OAX—for sharing both steps with the larger developer community, it is easy for developers to adopt a new XPU into their edge AI applications. To adopt a new XPU, a developer simply:

[5] When thinking about models as a DAG, it is easy to see that a subgraph of the DAG is a DAG itself. As many XPU only support a small set of operations — smaller than the most current ONNX standard operations set — often only a subgraph of the original graph is meaningfully accelerated.

1. Executes the **OAX toolchain**, on whichever machine they prefer, to generate the XPU specific model $\mathcal{M}_{xpu}$. This step eliminates having to learn XPU specific conversion or training tools

2. "Swaps" the **OAX runtime** in the larger application pipeline to match the selected hardware. Given a single interface for each and every runtime,[6] this step eliminates any re-engineering necessary to adopt the new XPU.

As a first implementation of OAX we make a number of specific technical choices. For the **OAX toolchain** we use pre-defined Docker images. Usage of Docker images to implement the **OAX toolchain** allows maximum flexibility for those contributing toolchains to use whatever conversion tools and dependencies they need inside the image. By simply standardizing the high level interactions with the container, we provide a unified method of model conversion while abstracting away any implementation details. On top of this, Docker images are easily managed and distributed and can be executed cross platform. For the **OAX runtime** we allow any (compiled) format, as long as there is a unified runtime ABI (shared library) available such that the runtime can be integrated in `C++` code. This should allow any low level application developer to utilize the OAX runtime (and will allow for higher level (e.g., `Python` tools to be built on top of the lower-level runtime(s).

1.4 Organization of this position paper

The remainder of this position paper is organized as follows; first, we detail the usage of OAX from a user / developer perspective. Next, we describe the core specification of OAX (the latest version of which can always be found at https://github.com/oax-standard). Subsequently, we discuss the organization of the consortium and the ways in which the community can contribute to moving OAX forward. Finally, we discuss relationships to other open and commercial projects that also aim, in one way or another, to make (edge) AI development and deployment easier. We close off with a discussion of the known limitations of OAX.

2 OAX from a user / developer perspective

Here we provide a few simple and self-contained[7] examples of how to use the OAX standard to switch from a CPU[8] only implementation of a (simplified) AI pipeline to one in which the model is accelerated using a Hailo Accelerator (using an initial implementation of OAX to move over to this XPU).

[6] Including runtimes that run on CPUs, and hence allow for easy local evaluation if no XPUs are readily available.

[7] In the body text we focus on the core lines of code that illustrate the utility of OAX, working code examples our found on the OAX github pages, https://github.com/oax-standard.

[8] We obviously provide a CPU only implementation of the OAX Runtime as a CPU is a member of the set of XPUs.

2.1 Getting started from C++

We start our example by assuming there is a hardware agnostic model file, `model-generic.onnx`, available to the user. We first show how to use the OAX toolchain image to convert the hardware-agnostic model to a XPU specific model, `model-xpu.oax`. Subsequently we show how to load the associated OAX XPU runtime using `C++` and use it to generate inferences.

2.1.1 Model Conversion

In this section we show how to, from the command line, run the OAX toolchain to transform the generic `model-generic.onnx` into a specific version of this model that can be run on the targeted accelerator. Given that the docker image has been downloaded to the local machine by the user, it suffices to run the following command:

```
> ls
model-generic.onnx

> docker run -v $pwd:/root/model oax-xpu-toolchain:latest
    /root/model/model-generic.onnx /root/model/
> ls
model-generic.onnx
model-xpu.oax
conversion-log.json
```

The workhorse of the above is clearly the `docker run` call which runs the (content of) the container once. We provide the following arguments:

- `-v $pwd:/root/model` The `-v` flag for Docker run sets up a shared files system between the docker and the host in the pre-specified location and thus allows for the model file to be accessed from the container and for the output to be written to the local file system.
- `/root/model/model.onnx` The path to the input file. Note that, depending on the XPU manufacturers specification, this can simply be a `.onnx` file, or it can be a `.zip` file containing the generic model description in ONNX and additional files for model conversion (i.e., validation examples used when quantizing the model).
- `/root/model/model-xpu.oax` The location and name of the output file. Users are free to choose the naming convention of the specific model file that is created.

The container should always produce the document `conversion-log.json`; this file should contain the conversion log containing human and machine readable conversion steps, and potential error messages. If conversion succeeds — and thus the `errors` key of the `conversion-log.json` file is empty—the container will also produce a the xpu specific model file under the user provided

name. We provide more details regarding the values in the `conversion-log.json` file and the possible input in our description of OAX in Section 3. However, exact details of the format and data in the conversion-log file, for example whether it should contain structured information regarding the properties of the model, should be specified by the community as a whole.[9]

2.1.2 Loading the OAX runtime

When interacting with the OAX runtime using `C/C++` on an edge device the application developer will need to load the XPU specific runtime into their project by including the correct shared library file:

```
// Load the runtime library
handle = dlopen("./libRuntimeLibrary.so", RTLD_NOW);

// Load each runtime function one at a time
runtime_initialization_t runtime_initialization =
    (runtime_initialization_t)dlsym(handle, "runtime_initialization");
// ...
runtime_inference_execution_t runtime_inference_execution =
    (runtime_inference_execution_t) dlsym(handle,
    "runtime_inference_execution");
runtime_destruction_t runtime_destruction = (runtime_destruction_t)
    dlsym(handle, "runtime_destruction");
```

2.1.3 Initializing the runtime

Before starting to use XPU runtime for inference, the runtime needs to connect to the XPU, allocate resources, load the optimized model, etc. That's achieved by calling the two functions below:

```
// initialize runtime environment
if (runtime_initialization()) {
    print_error_message(runtime_error_message());
    dlclose(handle);
    return 1;
}

// load model
const char *file_path = model_path;
if (runtime_model_loading(file_path)) {
    print_error_message(runtime_error_message());
    dlclose(handle);
    return 1;
}
```

[9] In our current internal implementation of OAX we also use mime-types to specify the output model format of the toolchain; using mime-types for this purposes is a technical detail that can be debated in the upcoming focus groups intended to further specify OAX.

2.1.4 Generating inferences

Once the model file is available (the result from the toolchain), and the correct shared library file has been loaded, the core code necessary to run inferences while benefiting form the XPU is follows:

```
// The input_tensors and output_tensors will contain the model input and
    output values.
tensors_struct input_tensors, output_tensors;
while(1){
    // Start inference on the input tensors
    if (runtime_inference_execution(&input_tensors, &output_tensors)) {
        print_error_message(runtime_error_message());
        break;
    }
    // here, you can use the model output tensors
    // ...

    // instruct the runtime to cleanup any intermediate results after
        we're done with the output tensors.
    if (runtime_inference_cleanup()) {
        print_error_message(runtime_error_message());
        break;
    }
}

// finalize runtime environment
if (runtime_destruction()) {
    print_error_message(runtime_error_message());
    dlclose(handle);
    return 1;
}
```

A complete and executable example of the above can be found int the OAX github: https://github.com/oax-standard.

3 An outline of OAX

Remark:
The current discussion of OAX reflects our current, first, example implementation. We are currently actively looking to seek the community to refine the specification in a number of focus groups planned summer 2024.

In this section we provide a first, rough, outline of OAX, including details regarding the I/O of the toolchain and the signature of the Runtime (i.e., the functions it needs to expose for its users).

3.1 OAX Toolchains

We start by detailing the input and output to the **OAX Toolchain**. As demonstrated before, the toolchain is implemented as an easy to distribute Docker image; at the end of this section we provide an example of core parts of the Dockerfile needed to create the image. However, the overall design philosophy for the toolchain image is to impose as little restrictions as possible: it is up to the developer implementing the toolchain container to choose its external structure, its permitted input files, and the resulting output file format(s).

3.1.1 Input to the Docker Container

As discussed above, the only input to the Docker container is a file. This could be a model file in `onnx` format, but, alternatively for those XPU manufacturers who wish to use validation or training data to optimize model conversion for their chip, could be a `.zip` archive containing the model file and any additionally necessary files as specified by the XPU manufacturer in the documentation of the contributed XPU container.[10].

3.1.2 Output from the Docker Container

The Docker container effectively writes out two files:

1. The model file. This can be either a `.onnx` file, a target specific binary, or anything the XPU manufacturer chooses to provide a device specific model description ($\mathcal{M}_{xpu}$).

2. A conversion log file, called `conversion-log.json`. This file contains information regarding the conversion, potential model information, suggestions, errors, etc. The exact format needs specification, but we envision the `json` to contain:

    - Meta data regarding the model.

[10] Again, we stress that the discussion here is merely a first proposal; we might — as a community – decide to enforce more structure on the container specification.

- Human readable messages regarding the model conversion and toolchain documentation aimed at improving the conversion process.
- Computer readable error codes in case the conversion fails.

3.1.3 Example of a Dockerfile

As mentioned before, the conversion toolchain expects two parameters: the path of the input model, and the output directory where it can save the results artifacts. Therefore, it's sensible to define a script as an entrypoint of the Docker image that can trigger the rest of the conversion as illustrated in the Dockerfile below:

```
ARG DEBIAN_FRONTEND=noninteractive
FROM ubuntu:22.04

WORKDIR /app

# You can for example install any dependencies the toolchain needs here
RUN #apt-get update -qq -y --fix-missing

COPY src/ /app/src/

ENTRYPOINT ["bash", "/app/src/run-toolchain.sh"]
```

3.1.4 Contributing OAX Toolchains

We have collected a first implementation of an OAX Toolchain for the Hailo accelerator in our github pages (see https://github.com/oax-standard). Copying this image should serve as a template for developers wishing to contribute their own XPU conversion toolchains.

3.2 OAX Runtimes

In this section we provide more detail regarding the **OAX runtime** which should be made available as a shared library file (see https://github.com/oax-standardfor examples). We first provide a high level overview of the runtime. Next, we describe the content of the shared library file itself, before detailing the exposed functions.

3.2.1 Overview of the runtime

The OAX runtime effectively implements the following steps:[11]

1. During the initialization step, the runtime is required to establish a connection with the XPU to ensure its usability, as well as initialize all components unrelated to the model.

[11] Where again we stress that this is merely intended to serve as a first suggestion.

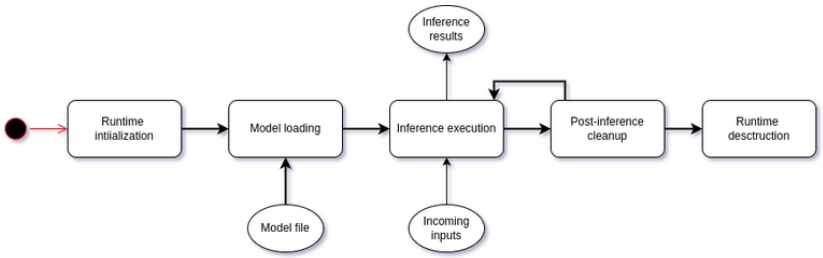

Figure 2: Overview of the interactions with the OAX runtime: The runtime is initialized, after which the XPU specific model file(s) are loaded. Next, inferences can be generated by passing a list of tensors to the runtime. The runtime also facilitates cleanup and destruction.

2. The second stage involves loading the optimized model from a file. At this juncture, the runtime should be capable of executing inference on incoming inputs.

3. Inference run execution: Once the optimized model is loaded, the runtime will be fed incoming inputs one at a time and expected to return output results.

4. After completing each inference run, the runtime proceeds with post-run cleanup tasks. This may involve releasing any temporary resources allocated during the inference process, resetting internal states, or preparing for the next set of inference runs. Subsequently, the runtime either returns to step 3 to continue with further inference execution as necessary, or goes to the final stage.

5. Runtime resources destruction: It ensures that the runtime environment is properly cleaned up and ready for shutdown or further initialization if needed.

3.2.2 Shared Library file

We make runtime available using shared library files. This allows a developer to include the XPU runtime in their edge AI project by simply "swapping" the included library; as the interface to the runtime is standardized – i.e., the shared library file implements the same methods – the surrounding code should not need any adaptation to move from one XPU[12] to another.

[12]Readers should mentally include standard CPUs into the set of available XPUs; this allows OAX to be used to develop a pipeline on a general purpose machine and subsequently, effortlessly, move it over to an edge device that includes a dedicated ML/AI accelerator.

3.2.3 Runtime interface

The runtime should provide an implementation of all functions listed below, which we split up in core functionality functions and utility functions:

- Core functionality functions:
  - `runtime_initialization`: This stage involves the initialization of the runtime, ensuring all necessary components are properly set up and ready for operation.
  - `runtime_model_loading`: During this stage, the runtime loads the model from storage or memory, preparing it for inference tasks.
  - `runtime_inference_execution`: In this stage, the runtime executes the inference process, utilizing the loaded model to process input data and generate output predictions.
  - `runtime_destruction`: This stage marks the finalization of the runtime process, where any remaining resources are released, and the runtime environment is shut down gracefully.
- Utility functions:
  - `runtime_inference_cleanup`: After completing the inference tasks, the runtime performs cleanup operations, releasing any allocated resources and preparing for subsequent executions.
  - `runtime_name`: denotes the name of the runtime, serving as a useful identifier to distinguish it from other runtimes within the system.
  - `runtime_version`: This parameter signifies the version of the runtime, providing valuable information for tracking and ensuring compatibility with other components or systems.
  - `runtime_error_message`: is utilized to retrieve the error message in case of a failure during any of the five stages of the runtime process. It serves as a valuable tool for diagnosing issues and troubleshooting problems encountered during runtime execution.

3.2.4 Contributing OAX runtimes

As for the OAX toolchain, we provide a number of first implementations of the runtime in our github repository, see https://github.com/oax-standard.

4 Additional tools and resources

Next to the core implementation of the toolchain and runtime described above, we aim to build OAX with a number of tools and utilities to increase its value both for developers and for contributing XPU manufacturers. Here we list envisioned Python tools, skeleton implementations, utility toolchains, and testing tools separately.

Remark:
At this point in time we simply list a number of desiderata; i.e., tools that we aim to develop within the OAX ecosystem. However, specific working groups should contribute their own tools, and the first focus group discussions should be used to prioritize tool development.

4.1 Python tools

Throughout this position paper we have demonstrated the core interactions with the runtime using the `C++` shared library (and hence we provide `C++` examples. We aim to develop a number of `Python` tools that allow for interactions with the OAX XPU runtimes directly from `Python`. A first example of such a Python based interaction with the lower level `C++` OAX runtime can already be found in our github: https://github.com/oax-standard. We aim to develop this further into a mature package allowing the `Python` developer community to easily integrate OAX into their development process.

4.2 Skeleton implementations

We want to ensure that contributing OAX toolchains and runtime, both by XPU manufacturers as well as by the community, is as easy as possible. As such we are looking to provide several skeleton implementations of the OAX toolchain and runtime, each with utilities (one can imagine simple `Python` based utilities to check the submitted `.onnx` graph, inspect its operators, recognize specific model architectures, etc.).

4.3 Utility toolchains

Interestingly, OAX "ONNX-to-XPU" toolchains can easily become part of a larger ecosystem of toolchains. For example, a simple container implementation of a toolchain which validates an ONNX graph and potentially changes its version would be well-sought after (and useful for model developers also outside of OAX). Conversion toolchain containers from other model formats to ONNX, i.e., conversion from PyTorch of from TensorFlow to ONNX, would also provide a valuable contribution to OAX (and to the edge AI community as a whole). Such model conversion from one-format-to-the-other is currently also supported by MMdnn [3].

4.4 OAX testing and profiling tools

When OAX matures, it will also prove a valuable ground for (automated) testing and profiling. XPU-contributed OAX toolchains can easily be tested using a variety of common model architectures to validate the conversion process and ensure the newly contributed XPU supports the most sought after operators.

OAX could provide a breeding ground to ensure a standardized list of operators that minimally need to be supported for an XPU to be useful within the larger edge AI ecosystem.

Similarly, OAX runtimes could easily be used to standardize model performance testing. Given a standardized set of trained models and input examples, a contributed OAX runtime can be tested automatically by feeding the input to the runtime in a standardized way and monitoring the output accuracy, running speed, power consumption, etc. of the XPU. Thus, OAX can provide the basic building block to develop an industry benchmark that XPU manufacturers can easily use to demonstrate the superiority of their hardware.

5 The OAX consortium organization

In the sections above we have detailed the utilities of OAX and we have tried to provide a high-level description of its first implementation. However, we are committed to make OAX a community initiative which is not owned by a single XPU manufacturer or ISV but rather serves as an open community to jointly grow the field of edge AI by allowing the market to accelerate. To ensure community participation we describe our envisioned organizational structure, the (currently) envisioned end-goal, and we extend an invite for those interested to join our first focus groups.

5.1 Envisioned organizational structure

Although it will take time to mature the organization of OAX, we envision building towards a structure that is akin to other successful standard in the Vision field such as ONVIF and ONNX. The ONVIF organization is mature and consist of the following bodies:

- **Steering Committee.** The Steering Committee is responsible for strategy and budgeting.
- **Technical Committee.** The Technical Committee (TC) drives the development of the core specifications as well as the technical direction and roadmap.
- **Technical Services Committee.** The Technical Services Committee (TSC) is responsible for the development of tools, testing and other auxiliary infrastructure.
- **Communication Committee.** The Communication Committee is responsible for the organization's external and internal communication.

The Technical Committee, the Technical Services Committee, and the Communication Committee will have several working groups (WGs) focussing on particular components of their tasks. We intend to move towards a similar organization structure to ensure continuity of OAX.

5.1.1 Envisioned process

In parallel to building a mature organizational structure, we aim to start the process of creating an actual standard for (parts of) OAX. At this moment in time we are in contact with ITU (https://www.itu.int/en/ITU-T/studygroups/2017-2020/16/Pages/q5.aspx) to aid us in this process and to coordinate the first focus groups to detail the first technical and organizational structures OAX.

5.2 Joining the first OAX focus groups

While the above sketches a direction of the organization of OAX, you can get involved straight away. We invite you to express your interest in joining our first focus group(s) – or in the future joining any of the respective working groups – by leaving your email at https://oax-standard.org.

6 Limitations

The first high level OAX description as provided in Section 3 contains a number of known gaps or discussion points. We highlight a few points that will need attention when further developing the standard:

- **Model class / architecture support.** As model architectures — and underlying operators — are evolving rapidly, it is an open question whether or not OAX should include specific (minimum) requirements regarding model architecture or operator support.
- **(Exact) runtime timing.** As numerous edge AI deployments — especially in an industrial setting — will need mechanisms to exactly time model execution (and potentially abort if timing requirements are exceeded), it is likely that explicit timing of the OAX runtime in a realtime OS will need more specification.
- **Runtime memory management.** Although our first implementation provides a rudimentary mechanism to free up memory, efficient memory use between CPU and XPU, and sometimes even within the XPU (which might be dependent on the model choice) can be key to squeezing out the last bits of performance. OAX should provide XPU manufacturers with all the tools the need to ensure their hardware is used to its fullest efficiency.
- **Runtime behavior for multiple models.** In many applications an application developer will use multiple models, either in sequence or in parallel. Additionally some XPUs support sequential use of the hardware with multiple inputs, or otherwise batched inputs to the XPU. These technical details will need to be developed either inside of the OAX runtime specification, or best-practices for the (e.g.,) parallel use of multiple OAX runtimes will need to be documented.

- **Runtime concurrency / asynchronous processing.** A number of XPUs thrive by having concurrent processing; currently OAX runtimes are setup to generate (batches of) inferences one-by-one; we intend to more fully work out alternative, asynchronous, models of execution.
- **XPU selection on target device.** We currently see multiple device manufacturers equipping edge AI devices with multiple different XPUs. Hence, it becomes key to control potentially multiple runtimes on a single device. Best practices dealing with such devices will need to be developed.

We are likely at this point overlooking a number of issues pertinent to both application developers (ISVs) and XPU manufacturers; we hope the first focus groups will allow us to more clearly define the limitations of the suggested approach and work on solutions in respective working groups.

7 Related projects

Given the large challenges that developers are facing to utilize new XPU efficiently, it is not surprise that there are many project which relate — in one way or another — to the development of OAX. Table 1 provides an overview. We aim to distill best-practices from each, and cordially invite contributors to any of these initiatives to collaborate on making edge AI more accessible.

Name	**Origin**	**Type**
OneAPI[4]	Intel	Acceleration Library
CUDA X[5]	NVIDIA	Acceleration Library
ArmCompute Library[6]	Arm	Acceleration Library
CoreML[7]	Apple	Acceleration Library
MMDNN[3]	Microsoft	Conversion toolchain utility
Paddle Paddle[8]	Paddle	Model to binary
NCNN[9]	Tencent	Model to binary
MNN[10]	Alibaba	Model to binary
ONNX Runtime[11]	Community	Runtime
Triton Inference server[12]	NVIDIA	Runtime
Apache TVM[13]	Community	Runtime
Huggingface deploy[14]	Hugging face	Training platform specific deployment
Open VINO[15]	OpenVino	Vendor specific toolchain + runtime
Vision AI DevKit[16]	Qualcomm	Vendor specific toolchain + runtime
Hailo AI Suite[17]	Hailo	Vendor specific toolchain + runtime
iMX OnnxRuntime[18]	NXP	Vendor specific toolchain + runtime

Table 1: Overview of a selection of projects relating to OAX. We see projects focussing on low level acceleration, model conversion, and model execution. Next to these projects we also see a number of vendor specific AI conversion and deployment projects.

8 Conclusions

In this position paper we have tried to detail the need for an open standard to make it easier to develop edge AI applications that benefit from newly developed, edge AI specific, hardware (XPUs). Next, we sketched a first outline of such a standard (working examples of which can be found at https://github.com/oax-standard). Finally we discussed the envisioned future organization of the standard and discussed a number of known limitations and relationships to other projects.

At this moment in time (May 2024), we encourage readers who want to join in specifying OAX and contributing to its community to join our first focus group(s) which will be planned in summer 2024. If you are interested, please do leave your email address at https://oax-standard.org.

References

[1] https://www.edge-ai-vision.com/2023-computer-vision-and-perceptual-ai-developer-survey/.

[2] https://onnx.ai/

[3] https://github.com/microsoft/MMdnn

[4] https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.htm

[5] https://www.nvidia.com/en-us/technologies/cuda-x/

[6] https://www.arm.com/technologies/compute-library

[7] https://developer.apple.com/documentation/coreml

[8] https://github.com/PaddlePaddle/FastDeploy

[9] https://github.com/Tencent/ncnn

[10] https://github.com/alibaba/MNN

[11] https://onnxruntime.ai/

[12] https://developer.nvidia.com/triton-inference-server

[13] https://tvm.apache.org/

[14] https://github.com/orgs/huggingface/repositories?q=optimum

[15] https://github.com/openvinotoolkit/openvino

[16] https://azure.github.io/Vision-AI-DevKit-Pages/

[17] https://hailo.ai/products/hailo-software/hailo-ai-software-suite/

[18] https://github.com/nxp-imx/onnxruntime-imx