Adaptive Vision WEAVER

WEAVER is an inference engine capable of executing Keras models. It focuses on high performance, versatility and ease of use.

Installation guide

WEAVER SDK comes in two versions:

- CPU which supports execution on CPU only
- GPU which supports execution on CPU and CUDA-capable GPU (faster than CPU counterpart)

Requirements

- (GPU version only) A CUDA-capable GPU, with "Compute Capability" greater than or equal 3.5 and less than or equal 7.5. Minimum 3 GB of a graphic memory is recommended. Display Driver with at least 456.38 version is required (recommended latest).
- At least 3.5 GB disk space for program files, SSD recommended.
- At least 8 GB RAM memory.
- 64-bit processor (Intel i5, i7 or better are recommended). AVX support is required.
- Windows 7, 8 or 10.

After installation

The WEAVER SDK installation directory is saved in the `WEAVER_SDK_PATH5_1` environment variable. It contains multiple subdirectories:

- `bin\x64` – this subdirectory contains multiple DLL files, required to run WEAVER SDK. These files needs to be accessible by a program using WEAVER SDK. It means, that they need to be copied next to the executable or, this directory needs to be added to the `PATH` environment variable (in rare cases, with GPU version, it may result with errors from other aplications using CUDA).
- `include\WeaverApi` – this subdirectory contains 2 header files. `WeaverApi.h` contains definitions of C API for WEAVER SDK and `WeaverApi_cpp.h` contains definitions of C++ API. C API should be used only for developing C applications and creating wrappers in other languages (e.g. Python, Java, C#).
- `lib\x64` – this subdirectory contains import library (`WeaverApi.lib`) allowing for linking with WEAVER SDK.
- `licenser` – this subdirectory contains Adaptive Vision License Manager, allowing for easy license management.

Examples are installed into `%ProgramData%\Adaptive Vision\Adaptive Vision WEAVER SDK 5.1\Examples` directory. They show a typical usage of WEAVER SDK to perform a classification of image with a MobileNet network using C API or C++ API. Please note that built executables require access to the binaries located in `%WEAVER_SDK_PATH5_1%\bin\x64`.

Typical ways to achieve it are described above and in a top comment in each example source code.

Tensor

Tensor holds or maps data used as an input or an output for running a model. Currently only one data type is supported: single precision floating point numbers.

Data management

Tensor can be created in 3 ways:

- owning and managing its data; with `weaver_tensor_create` (C API) or `weaver::tensor::tensor(const format&)` (C++ API)
- mapping to preallocated memory, allowing for reading and writing; with `weaver_tensor_create_mapped_writable` (C API) or `weaver::tensor::tensor(const format&, map_writable_data)` (C++ API)
- mapping to preallocated memory, allowing for reading only; with `weaver_tensor_create_mapped` (C API) or `weaver::tensor::tensor(const format&, map_readonly_data)` (C++ API)

In the first case, creating a tensor allocates required memory which is deallocated during the tensor destruction. The second and third case are very similar. They both require a pointer to preallocated memory which is not deallocated by any operation on the tensor. Please note, that the mapped memory cannot be deallocated until the mapping tensor is destroyed. The only difference between these two cases are operations allowed on mapped memory, which is described more closely below.

Accessing the data managed by a tensor is separated in 2 operations:

- accessing the data for reading only, which is done by `weaver_tensor_get_data` (C API) or `weaver::tensor::data()` (C++ API)
- accessing the data for writing (and reading), which is done by `weaver_tensor_get_writable_data` (C API) or `weaver::tensor::data_writable()` (C++ API)

Returned values and possible errors are summarized by the table below.

	memory owning	writable mapping	read-only mapping
Getting data for reading	Ok, returns a pointer	Ok, returns a pointer	Ok, returns a pointer
Getting data for writing (and reading)	Ok, returns a pointer	Ok, returns a pointer	Error, returns the NULL pointer (C API) or throws a `weaver::exception` (C++ API)

Please note, that the functions from C API may also fail in case of passing a corrupted (e.g. NULL) tensor.

Data order and dimensions

Data managed by a tensor is also described with the two more (after a type) parameters: data order and data dimensions. These two are closely related. Currently, only supported data orders are:

- NL,
- NHWC,
- NCHW,

where N is a batch size, L is a length of linear data (e.g. confidence values), H is a height of image-like data, W is a width of image-like data and C is a depth (number of channels) of image-like data. Order of the letters starts from the outermost dimension (which changes the least frequently), to the innermost one (which changes the most frequently). Tensor dimensions are a list of sizes of each dimension, ordered by the data order.

The NL data order is used mostly for results of a classification, as it denotes tensors containing some linear data, like confidence values. This specific data order means that memory holds L values, than another L values, and so on, multiplied N times. For example, classification results of 3 images to 5 classes, grouped with the NL data order, would yield a tensor with dimensions: 3, 5 and holding 15 values: 5 confidence values for the first image, than 5 values for the second one and the last 5 values for the third one.

The NHWC and NCHW data orders are used mostly for image-like tensors. The first one is similar to an images stored in an “interleaved” format, where a single pixel holds values for all channels. The second one is similar to an images stored in a “planar” format, where single image is composed from multiple subsequent “subimages”, one for each channel.

For example, let us assume some sample data as 2 images with 3 channels (red-green-blue, for easy referencing), height equal to 5 and width equal to 7. This data in NHWC data order would yield a tensor with dimensions: 2, 5, 7, 3 and holding 210 values. The first 3 values would be red, green and blue channel values for the top left pixel of the first image. Subsequent 3 values would be red, green and blue channel values for the second pixel in the first row of the first image, and so on. After 21 values, which compose the first row of the first image, another 21 values would be placed, composing the second row of the first image, and so on. After 105 values, which compose the first image, second image data is placed, starting from the top left pixel.

Same data in NCHW data order would yield a tensor with dimensions: 2, 3, 5, 7 and holding 210 values. The first 35 values would compose an “image” containing values of the red channel in each coordinate of the first image, row after row. Subsequent 35 values would compose a similar “image” for the green channel and further 35 values would compose an analogous “image” for the blue channel. After these 105 values, which compose the first image, second image data is placed, starting from its red channel.

Please note, that tensor currently does not support byte padding and assume that all values are tightly packed in memory, without any gaps. On the other hand, such padding is common in many external libraries related to an image processing. Removing the padding may be done by a copying only meaningful data (without the padding) from a source to a tensor or by using specialized function from the library.

Usage

WEAVER SDK is designed to be used as a part of C/C++ projects developed with Microsoft Visual Studio 2015-2019.

Project Configuration

Projects using WEAVER SDK have to add:

- `%WEAVER_SDK_PATH5_1%\include` path to *Configuration Properties* > *C/C++* > *General* > *Additional Include Directories*.
- `%WEAVER_SDK_PATH5_1%\lib` path to *Configuration Properties* > *Linker* > *General* > *Additional Library Directories*.
- `WeaverApi.lib` to *Configuration Properties* > *Linker* > *Input* > *Additional Dependencies*

All program using WEAVER SDK have to load DLL files from `%WEAVER_SDK_PATH5_1%\bin\x64`. Common ways to ensure that, are:

- copy contents of the `%WEAVER_SDK_PATH5_1%\bin\x64` next to final executable file, or
- add `%WEAVER_SDK_PATH5_1%\bin\x64` to the `PATH` environment variable, or
- copy contents of the `%WEAVER_SDK_PATH5_1%\bin\x64` to some directory listed in the `PATH` environment variable.

Please note that the last 2 options may lead in rare cases and in GPU version to errors in other applications using CUDA.

The first option may be accomplished by adding `xcopy "$(WEAVER_SDK_PATH5_1)\bin\$(PlatformName)" "$(OutDir)" /d /y` to *Configuration Properties* > *Build Event* > *Post Build Event* > *Command Line*.

API usage

A typical use of WEAVER SDK can be divided into several steps:

1. Deploying a model.
   This is done by `weaver_model_deploy` (C API) or by `weaver::model` constructor (C++ API). It requires path to a Keras model file (saved with weights and architecture) and a target device, used for running the model later. Deploying the model allocates memory on the selected device for model weights and other data.
   Also, it is strongly encouraged to set desired data orders of output tensors. The order should be chosen depending on “type” of the specific output tensor (e.g. if image is expected, correct orders would be NCHW or NHWC; if classification result is expected, correct order would be NL) and an algorithm used for parsing output data. Also, desired data order can be set or changed later with `weaver_model_set_desired_output_order` (C API) or `weaver::model::set_desired_output_ordering` (C++ API).
2. Input data loading and preparing.
   This step covers obtaining input data which is outside scope of WEAVER SDK. In many cases, it is done with external libraries, like Adaptive Vision Library or OpenCV.
   Generally speaking, it comes to getting a raw input data from a file, a camera or an other source and converting the data to a format (e.g. floating point numbers ranging from -1 to 1) expected by the specific network input.

3. Creating input tensors.
   Input tensors are direct inputs for the model in WEAVER SDK. Tensors may own its memory or map some external (possibly read-only) memory. Please note, that tensors does not support byte padding which is added in many cases in image types from external libraries. Common way to ensure that data handled by a tensor has no padding is to copy only meaningful bytes from a source memory to the tensor data. Also, some external libraries may have a function for removing such padding.
4. Running the model.
   This is done with `weaver_model_run` (C API) or `weaver::model::run` (C++ API). It performs calculations on the device selected during the model deployment. First call may take longer than following ones as it allocates memory needed for executing layers. Output tensors are created during the call and does not need to be preallocated before.
5. Parsing output tensors.
   This step covers getting meaningful data from raw output data which is outside scope of WEAVER SDK. In most cases, it comes to finding a maximum in confidence values (in case of classification) or converting output tensor data to an image (e.g. in case of autoencoders).

Distributing

Once the application is ready, it is time for preparing a distribution package or an installer. There are several requirements that needs to be fulfilled:

- The final executable file of the application needs to have access to the contents of `%WEAVER_SDK_PATH5_1%\bin\x64`. Common ways to ensure that are described in “Project Configuration” section.
- The computer that the application will run on needs a valid license for the use of Adaptive Vision WEAVER SDK product. Licenses can be managed with the License Manager application, that is installed with Adaptive Vision WEAVER SDK.
- A license file (`*.avkey`) can be also manually copied to the end user's machine without installing Adaptive Vision WEAVER SDK. It must be placed in a subdirectory of the *AppData* system folder. The typical location for the license file is `C:\Users\%USERNAME%\AppData\Local\Adaptive Vision\Licenses`. Remember that the license is valid per machine, so every computer that runs the application needs a separate license file.
- Alternatively to the (`*.avkey`) files we support USB Dongle licenses.

Troubleshooting

Errors in WEAVER SDK are reported with return values (C API) or `weaver::exception` exceptions (C++ API). Possible errors are described more closely in comments to `weaver_status_t` enumeration values in `WeaverApi.h`. Also, in most cases, additional info is available. It can be retrieved with `weaver_get_last_error_info()` function (C API) or `weaver::exception::what()` method (C++ API). It should help in solving problems.
Please note that additional info may by partially encrypted. If the public part does not help in solving an issue, contact with Adaptive Vision Support is required.

Supported Keras layers and features

The table below lists the supported layers with possible restrictions. For parameters not explicitly mentioned, WEAVER can accept any value valid from the Keras' point of view.

Keras layer	**Restrictions**
Activation	activation=relu/relu6/sigmoid/softmax/tanh
Add	Only two inputs
BatchNormalization	axis=-1
Conv2D	activation=None/tanh/relu/sigmoid/softmax/elu
Conv2DTranspose	activation=None/tanh/relu/sigmoid/softmax/elu output_padding=None
Concatenate	axis=-1
Cropping2D	
Dense	activation=None/tanh/relu/sigmoid/softmax/elu
DepthwiseConv2D	activation=None/tanh/relu/sigmoid/softmax/elu depth_multiplier=1
Dropout	
ELU	
GlobalAveragePooling2D	
InputLayer	
LeakyReLU	
MaxPooling2D	padding=valid
Multiply	Only two inputs
ReLU	negative_slope=0 threshold=0
Reshape	target_shape have to count 1, 2 or 3 values
PReLU	shared_axes=None/[1, 2]
TimeDistributed	
UpSampling2D	interpolation=nearest Both values in size must be the same in case of running a model on CUDA.
ZeroPadding2D	

Additionally:

- For layers that have "data_format" parameter, only "channels_last" is supported.
- Nested models are also supported as long as they contain only supported layers.
- WEAVER works with float32 data type only.
- WEAVER works with Keras Functional models only. Python script `train_mnist_model.py` located in "Trained MNIST Model" example shows definition and training of an example model.