plaidml

plaidml

Jan 17, 2020

CONTENTS:

CHAPTER

ONE

INSTALLATION INSTRUCTIONS

PlaidML supports Ubuntu, macOS , and Microsoft Windows operating systems.

Ubuntu

If necessary, install Python's `pip` tool. OpenCL 1.2 or greater is alsorequired.

```
sudo add-apt-repository universe && sudo apt update
sudo apt install python3-pip
sudo apt install clinfo
```

Run `clinfo`, and if it reports `"Number of platforms" == 0`, you can install a driver (GPU) or enable a CPU via one of these options:

- **Nvidia** – For Nvidia GPUs, run:

```
sudo add-apt-repository ppa:graphics-drivers/ppa && sudo apt update
sudo apt install nvidia-modprobe nvidia-384 nvidia-opencl-icd-384 libcuda1-384
```

- **AMD** – For AMD graphics cards, download the AMDGPU PRO driver and follow the instructions provided by AMD for the chip.
- **Intel® Xeon® Processors OR Intel® Core™ Processors** – In lieu of installing specific drivers, you can install ngraph with pip, or you can build the nGraph Library with the cmake flag *-DNGRAPH*PLAIDML*ENABLE=TRUE*.

Python

Although PlaidML can be run with Python2, we recommend Python3, as well as judicious use of a Virtualenv. To create one just for using PlaidML:

```
python3 -m venv plaidml-venv
source plaidml-venv/bin/activate
```

Keras

There are two ways to get Keras working on your system:

1. Isolate it to your *venv* as follows:

```
pip install -U plaidml-keras
```

2. Alternatively, install the PlaidML wheels system-wide with:

```
sudo -H pip install -U plaidml-keras
```

Finally, set up PlaidML to use a preferred computing device:

```
plaidml-setup
```

You can test the installation by running MobileNet in plaidbench. Remember to use `sudo -H` if you're working outside of a virtual environment.

```
pip install plaidml-keras plaidbench
plaidbench keras mobilenet
```

You can adapt any Keras code by using the PlaidML backend instead of the TensorFlow, CNTK, or Theano backend that you'd normally use; simply change the Keras backend to `plaidml.keras.backend`. You can do this by modifying

`~/.keras/keras.json` so that the backend line reads `"backend": "plaidml.keras.backend"` If this file does not exist, see the [Backend instructions for Keras]. If you don't need anything special in your Keras settings, you can set the `~/.keras/keras.json` as follows:

```
{
    "epsilon": 1e-07,
    "floatx": "float32",
    "image*data*format": "channels_last",
    "backend": "plaidml.keras.backend"
}
```

Another option is to globally set the `KERAS_BACKEND` environment variable to *plaidml.keras.backend*. A monkey-patch technique involving `plaidml.keras.install_backend()` may still work, but should be considered deprecated in favor of the above methods.

macOS

A computer listed on Apple's compatibility list with support for OpenCL 1.2 is required; those from 2011 and later usually fit this requirement.

Python

Although PlaidML can be run with Python2, we recommend Python3, as well as judicious use of a Virtualenv. To create one just for using PlaidML:

```
python3 -m venv plaidml-venv
source plaidml-venv/bin/activate
```

Keras

To install PlaidML with Keras, run the following:

```
pip install -U plaidml-keras
```

Finally, set up PlaidML to use a preferred computing device:

```
plaidml-setup
```

PlaidML should now be installed! You can test the installation by running MobileNet in plaidbench.

```
pip install plaidml-keras plaidbench
plaidbench keras mobilenet
```

Microsoft Windows

These instructions assume Windows 10 without Python installed; adapt accordingly.

1. First install Chocolatey by starting an Administrator PowerShell and running:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object System.Net.
↪WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

You'll likely need to reboot your shell at this point.

2. Install Python:

```
choco install -y python git vcredist2015
```

3. Switch to an unprivileged PowerShell to install and set up PlaidML with Keras

```
pip install -U plaidml-keras
plaidml-setup
```

PlaidML should now be installed! You can test the installation by running MobileNet in plaidbench.

```
pip install plaidml-keras plaidbench
plaidbench keras mobilenet
```

Intel® SDK for OpenCL™ Applications

CHAPTER

TWO

BUILDING FROM SOURCE

2.1 Install Anaconda

Install Anaconda. You'll want to use a Python 3 version. After installing Anaconda, you'll need to restart your shell, to pick up its environment variable modifications (i.e. the path to the conda tool and shell integrations). For Microsoft Windows, you'll also need the Visual C++ compiler (2017+) and the Windows SDK, following the Bazel-on-Windows instructions.

2.2 Install bazelisk

The Bazelisk tool is a wrapper for Bazel which provides the ability to enfore a particular version of Bazel. Download the latest version for your platform and place the executable somewhere in your PATH (e.g. `/usr/local/bin`). You will also need to mark it as executable. Example:

```
wget https://github.com/bazelbuild/bazelisk/releases/download/v0.0.8/bazelisk-darwin-
↪amd64
mv bazelisk-darwin-amd64 /usr/local/bin
chmod +x /usr/local/bin/bazelisk
```

https://github.com/bazelbuild/bazelisk/releases

2.3 Configure the build

Use the *configure* script to configure your build. Note: the *configure* script requires Python 3. By default, running the *configure* script will: * Create and/or update your conda environment * Configure pre-commit hooks for development purposes * Configure bazelisk based on your host OS

```
./configure
```

On Windows, use:

```
python configure
```

Here's an example session:

```
$ ./configure
Configuring PlaidML build environment
conda found at: /usr/local/miniconda3/bin/conda
Creating conda environment from: $HOME/src/plaidml/environment.yml
```

(continues on next page)

(continued from previous page)

```
Searching for pre-commit in: $HOME/src/plaidml/.cenv/bin
pre-commit installed at .git/hooks/pre-commit
bazelisk version
Bazelisk version: v0.0.8
Starting local Bazel server and connecting to it...
Build label: 0.28.1
Build target: bazel-out/darwin-opt/bin/src/main/java/com/google/devtools/build/lib/
↪bazel/BazelServer_deploy.jar
Build time: Fri Jul 19 15:22:50 2019 (1563549770)
Build timestamp: 1563549770
Build timestamp as int: 1563549770
Using variant: macos*x86*64
Your build is configured.
Use the following to run all unit tests:
bazelisk test //...
```

2.4 Build the PlaidML Python wheel

```
bazelisk build //plaidml:wheel
```

2.5 Install the PlaidML Python wheel

```
pip install -U bazel-bin/plaidml/wheel.pkg/dist/*.whl
plaidml-setup
```

2.6 PlaidML with Keras

The PlaidML-Keras Python Wheel contains the code needed for integration with Keras. You can get the latest release of the PlaidML-Keras Python Wheel by running:

```
pip install plaidml-keras
```

You can also build and install the wheel from source.

2.7 Set up a build environment

Follow the setup instructions for Build the PlaidML Python wheel, above.

2.8 Build the PlaidML-Keras wheel

```
bazelisk build //plaidml/keras:wheel
```

2.9 Install the PlaidML-Keras Python wheel

```
pip install -U bazel-bin/plaidml/keras/wheel.pkg/dist/*.whl
```

2.10 Testing PlaidML

Unit tests are executed through bazel:

```
bazelisk test //...
```

Unit tests for frontends are marked manual and must be executed individually (requires running `plaidml-setup` prior to execution)

```
bazelisk run //plaidml/keras:backend_test
```

CHAPTER

THREE

CONTRIBUTING TO PLAIDML

We welcome contributions to PlaidML from anyone. This document contains: * Guidelines for creating successful PRs * Outlines the contribution process * Lists general areas for contribution * Provides resources and context to ease development, where relevant and available Before starting any work, please ensure you are able to build and test PlaidML. Guidelines ****** * Create unit tests for new features and bug fixes. Integration tests are required for larger features.

- C++ code conforms to the Google Style Guide for CPP.
- Python code conforms to the Google Python Style Guide.

3.1 Process

1. Ensure there is an open issue assigned to you before doing (too much) work: * If you're tackling an open issue that isn't assigned, please assign it to yourself. * If you'd like to solve an issue that is already assigned, please comment on the issue to ensure you're not duplicating effort. * If you'd like to provide a new feature, open a new issue. Please provide a reasonably-detailed description of what you'd like to do, and clearly indicate that you're willing to do the work.
2. Work on a fork as usual in GitHub. Please ensure the same tests travis runs will pass before creating a PR.
3. Review the License file in the `plaidml` repo and the Guidelines on this page.

4. Once tests have passed, a maintainer will assign the issue to themselves and run the PR through the (currently private) performance test suite. If there are issues, we will attempt to resolve them, but we may provide details and ask the author to address. 5. Once the performance regression suite has passed, we will accept and merge the PR. Areas for Contribution ********** * Ops for Frontends

- PlaidML welcomes implementations for currently unimplemented operations as well as Tile code for novel operations supported by research.
- Please read Adding Tile Ops and How to Write Tile Code tutorials.
- ML Framework Frontends (e.g., Keras, Pytorch, etc) * PlaidML welcomes integrations with any established ML framework or interop (NNVM, ONNX, etc). * You can find commonly used operations in the [plaidml.op](api/plaidml.op.rst) module. * Please read building a frontend tutorial.
- HALs for Backend Targets (OpenCL, Vulkan, SPIR-V, HVX, etc) * There is no documentation for the HAL currently. The interface is fairly straightforward and the *OpenCL HAL <../tile/hal/opencl>* provides a good example of a complete HAL.

Please follow the process above before embarking on anything major (like integrating a new frontend or backend).

CHAPTER

FOUR

TROUBLESHOOTING

Having trouble getting PlaidML to work? Well, you're in the right place! Before you open a new issue on GitHub, please take a look at the Common Issues, Enable Verbose Logging in PlaidML, and Run Backend Tests. These steps will help enable us to provide you with better support on your issue.

4.1 Common Issues

4.1.1 PlaidML Setup Errors

4.1.2 Memory Errors

```
OSError: exception: access violation reading 0x0000000000000030
```

This error might be caused by a memory allocation failure, and it fails silently. You can fix this error by decreasing your batch size and trying again.

```
plaidml.exceptions.ResourceExhausted: Out of memory
```

This error is caused by incorrect Tile syntax.

4.1.3 Bazel Issues

For any Bazel-specific issues you're encountering, we recommend that you first visit Bazel's installation documentation which has a comprehensive overview of Bazel on various platforms. Any issues commonly encountered by PlaidML users are documented below.

```
Encountered error while reading extension file 'workspace.bzl': no such package
↪'@toolchain//'
```

On MacOS devices, *toolchain* errors often indicate that the user does not have Xcode properly installed. Even if you have Xcode Command Line Tools installed, you may not have a proper installation of Xcode itself. To check your installation of Xcode, first print the path of the active developer directory:

```
xcode-select -p
```

The resulting path should be `/Applications/Xcode.app/Contents/Developer`. If that is not the path you are seeing when you run *xcode-select -p*, please go to the App Store and download Xcode. After verifying that Xcode is properly installed, you will need to reset your Bazel instance before running Bazel again:

```
bazelisk clean --expunge
```

4.1.4 PlaidML Exceptions

```
Applying function, tensor with mismatching dimensionality
```

This error may be caused by a known issue with the *BatchDot* operation, where results are inconsistent across backends. The Keras documentation for BatchDot matches the Theano backend's implemented behavior and the *default* behavior within PlaidML. The TensorFlow backend implements BatchDot in a different way, and this causes a mismatch in the expected output shape (there is an open issue against TensorFlow to get this fixed). If you have existing Keras code that was written for the TensorFlow backend, and it is running into this issue, you can enable experimental support for TensorFlow-like *BatchDot* behavior by setting the environment variable *PLAIDML*BATCHDOT*TF_BEHAVIOR* to *True*.

```
ERROR:plaidml:syntax error, unexpected -, expecting "," or )
```

This error may be caused by special characters, such as -, that are used in variable names within your code. Please try removing and/or replacing special characters in your variable names, and try running again.

4.2 Run Backend Tests

Backend Tests provide us with useful information that we can use to help solve your issue. To run backend tests on PlaidML, follow these steps:

1. Verify that you have the PlaidML Python Wheel built as specified in Building from source
2. Run the backend tests through Bazel

```
bazel test --config macos*x86*64 @com*intel*plaidml//plaidml/keras:backend_test
```

4.3 Enable Verbose Logging

You can enable verbose logging through the environment variable *PLAIDML_VERBOSE*. *PLAIDML_VERBOSE* should be set to an integer specifying the level of verbosity (valid levels are 0-4 inclusive, where 0 is not verbose and 4 is the most verbose).

For instance, the following command would set a verbosity level of 1.

```
export PLAIDML_VERBOSE=1
```

CHAPTER

FIVE

TILE EDSL

The C++ Tile eDSL (Embedded Domain Specific Language) provides developers with a way of describing a neural network so that the Stripe-based PlaidML compiler can construct an efficient implementation. This tutorial is intended to help machine learning practitioners (or anyone with a background in software engineering and mathematics) get started using the C++ Tile eDSL.

5.1 Scope and Warning

This tutorial provides an introduction to the C++ Tile eDSL. It is intended to help machine learning practitioners get started writing Tile code as quickly as possible, and as such covers core features, not every language detail. This is a tutorial, not a spec, and as such will consist of a series of examples, with a summary reference section at the end. This tutorial covers how to use the C++ Tile eDSL, not how Tile code is constructed and manipulated by PlaidML. It does not cover the workings of PlaidML utilities such as the pmlc compiler. Tile and PlaidML are still being developed and the APIs discussed here are subject to change.

5.2 How to Write Tile Code

5.2.1 Sum Over Axis

We're ready to look at some C++ Tile code! Here's an operation that takes the sum over axis *0* of a 2D tensor (in Keras this would be `K.sum(I, axis=0)`):

C++

```
Tensor sum_over_axis(const Tensor& I) {
  TensorDim M, N;
  TensorIndex m, n;
  I.bind_dims(M, N);
  auto O = TensorOutput(N);
  O(n) += I(m, n); // contraction
  return O;
}
```

Python

```
def sum_over_axis(I):
  M, N = TensorDims(2)
  m, n = TensorIndexes(2)
  I.bind_dims(M, N)
  O = TensorOutput(N)
```

(continues on next page)

(continued from previous page)

```
  # contraction
  O[n] += I[m, n]
  return O
```

An operation such as this which merges together values across one or more indices is called a *contraction*. The syntax may look a bit odd at first, but it's related to summation notation. Below we show how this C++ Tile code is related to the mathematical formula for the operation by using colors to highlight corresponding pieces:

$$O[n] = \sum_m I[m, n]$$

```
O(n) |   += | I(m, n)|;|
```

In green, notice that the summation symbol is represented as += in C++ Tile code. Some portions of the notation do not perfectly correspond. Here's why:

- Summation notation includes a `m` subscript to indicate that `m` is the variable being summed over. Tile code implicitly sums over all valid indices (valid means not out of range for any tensor, and not failing any additional user-specified constraints as discussed in later examples).
- Tile must be explicitly given the shape of any new tensor created, done in this code by `TensorOutput(N)`. In this case we want `N` to match the size of the last dimension of `I`, which is specified by using `I.bind_dims(M, N)`. It is possible, however, to make this dimension of `O` larger or smaller, which would zero-pad or truncate `O` respectively. For example,

  C++

  ```
  auto O = TensorOutput(N + 1);
  ```

  Python

  ```
  O = TensorOutput(N+1)
  ```

  would result in a *0* as the last element of *O* if we're still assuming *N* is the size of the last dimension of *I*.
- As is the case for all C++ statements, they must end with a semicolon.

5.2.2 Max Over Axis

Taking the maximum over axis `0` looks very similar to taking the sum over axis `0`. Just like a sum is represented in Tile with +=, a max is represented by >=. Thus, the Tile code for max over axis `0` is just a single character change from sum over axis `0`. Let's look at it as a Tile function:

C++

```
Tensor max_over_axis(const Tensor& I) {
  TensorDim M, N;
  TensorIndex m, n;
  I.bind_dims(M, N);
  auto O = TensorOutput(N);
  O(n) >= I(m, n);
  return O;
}
```

Python

```
def max_over_axis(I):
  M, N = TensorDims(2)
  m, n = TensorIndexes(2)
  I.bind_dims(M, N)
  O = TensorOutput(N)
  O[n] >= I[m, n]
  return O
```

Again, this corresponds closely to mathematical notation:

$$O[n] = \max_m I[m, n]$$

```
O(n) |  >= | I(m, n)|;|
```

5.2.3 Matrix Multiply

Next we'll consider matrix multiplication. Let's look at the mathematical expression for the matrix multiplication `C = AB` written out in element-level detail:

$$C[i, j] = \sum_k (A[i, k] \cdot B[k, j])$$

We can convert this to C++ Tile code using the same correspondence as the previous example: The summation sign becomes plus-assignment, the summation index is omitted, dimensions are given for the output tensor, and the statement ends in a semicolon. Here's the result:

C++

```
C(i, j) += A(i, k) * B(k, j);
```

Python

```
C[i, j] += A[i, k] * B[k, j];
```

To have correct dimensions, we need `I` to be the first dimension of `A` and `J` the last dimension of `B`. Here's how this looks as part of a full Tile function:

C++

```
Tensor matmul(const Tensor& A, const Tensor& B) {
  TensorDim I, J, K;
  TensorIndex i, j, k;
  A.bind_dims(I, K);
  B.bind_dims(K, J);
  auto C = TensorOutput(I, J);
  C(i, j) += A(i, k) * B(k, j);
  return C;
}
```

Python

```
def matmul(A, B):
  I, J, K = TensorDims(3)
  i, j, k = TensorIndexes(3)
  A.bind_dims(I, K)
  B.bind_dims(K, J)
```

(continues on next page)

(continued from previous page)

```
  C = TensorOutput(I, J)
  C[i, j] += A[i, k] * B[k, j]
  return C
```

Notice that we use `bind_dims` on inputs and we use `TensorOutput` on outputs. Input dimensions can be repeated, which results in an error if the Tile function is passed inputs whose corresponding dimensions don't all have the specified size (for example *A.bind_dims(K, K)* would be constrained to a square).

5.2.4 Global Min

There is a min contraction `<=` analogous to the max contraction `>=`. For the purposes of this example, however, let's use the formula `min(X) = -max(-X)`, to compute the min. We do this by combining a max computation with *elementwise* operations that perform the same operation (in this case negation) on every element of a tensor. Elementwise operations generally cannot be performed on the same line as contractions, so we write the global min function (for a 3D tensor) as follows:

C++

```
Tensor global_min(const Tensor& I) {
  TensorIndex i, j, k;
  auto Neg = -I;
  auto O_Neg = TensorOutput();
  O_Neg() >= Neg(i, j, k);
  auto O = -O_Neg;
  return O;
}
```

Python

```
def global_min(I):
  i, j, k = TensorIndexes(3)
  Neg = -I
  O_Neg = TensorOutput()
  O_Neg[()] >= Neg[i, j, k]
  O = -O_Neg
  return O
```

There are several novel pieces in this example. First, note that the elementwise operations do not include dimensions. Dimensions are inferred from the inputs in elementwise operations, and so are never specified in elementwise ops. *Neg* has the same shape as `I`, and `O` has the same shape as `O_Neg`. When an elementwise binary operation is performed, the output shape is determined using broadcasting semantics. Which brings us to the next novelty: we have our first example of a 0D tensor, `O_Neg`. Tensors in Tile are allowed to have zero dimensions. In such a case the tensor represents a scalar, i.e., a single value. In places where dimensions are specified, you can indicate a 0-dimensional tensor by using `()` for the dimensions, as in this example. Notice that we are taking the max over all axes in a single operation. Contractions implicitly aggregate over *all* indices that write to the same output location (in this case we aggregate over all values of `i`, `j`, and `k`).

5.2.5 Average

To compute the mean of a tensor, we need to sum the elements and divide by the total number of elements summed. We can do this by taking advantage of the fact that we can divide by a constant (including an input `TensorDim`) as an elementwise operation. Thus, to take the mean over axis `0` of a 2D tensor, we write:

C++

```
Tensor avg(const Tensor& I) {
  TensorDim X, Y;
  TensorIndex x, y;
  I.bind_dims(X, Y);
  auto Sum = TensorOutput();
  Sum(y) += I(x, y);
  return Sum / X;
}
```

Python

```
def avg(I):
  X, Y = TensorDims(2)
  x, y = TensorIndexes(2)
  I.bind_dims(X, Y)
  Sum = TensorOutput()
  Sum[y] += I[x, y]
  return Sum / X
```

We can perform multiple elementwise operations on the same line, including operations on constants and input dimensions. So, while it would be possible to take a global mean of a 2D tensor in stages as so:

C++

```
Tensor avg(const Tensor& I) {
  TensorDim X, Y;
  TensorIndex x, y;
  I.bind_dims(X, Y);
  auto Sum = TensorOutput();
  Sum() += I(x, y);
  PartialMean = Sum / X;
  return PartialMean / Y;
}
```

Python

```
def avg_stages(I):
  X, Y = TensorDims(2)
  x, y = TensorIndexes(2)
  I.bind_dims(X, Y)
  Sum = TensorOutput()
  Sum[()] += I[x, y]
  PartialMean = Sum / X
  return PartialMean / Y
```

it is more straightforward to merge the elementwise operations:

C++

```
Tensor avg(const Tensor& I) {
  TensorDim X, Y;
  TensorIndex x, y;
  I.bind_dims(X, Y);
  auto Sum = TensorOutput();
  Sum() += I(x, y);
  return Sum / (X * Y);
}
```

Python

```
def avg_merge(I):
  X, Y = TensorDims(2)
  x, y = TensorIndexes(2)
  I.bind_dims(X, Y)
  Sum = TensorOutput()
  Sum[()] += I[x, y]
  return Sum / (X * Y)
```

5.2.6 Max Pool 1D

Next let's implement a size 2 stride 2 maxpool in Tile. This is the operation that splits a tensor into groups of 2 and takes the larger element from each group, yielding a tensor of half the original size. This is straightforward to implement in straight C++:

C++

```
float I[N], O[N / 2];
for (int i = 0; i < N/2; ++i) {
  float curr_max = FLT_MIN;
  for (int j = 0; j < 2; ++j) {
    if (I[2 * i + j] > curr_max) {
      curr_max = I[2 * i + j];
    }
  }
  O[i] = curr_max;
}
```

Python

```
for i in range (1 , N//2):
  curr_max = numpy.finfo(float).eps
  for j in range (1 , 2):
    if I[2*i*j] > curr_max:
      curr_max = I[2*i+j]
  O[i] = curr_max
```

`for` loops over tensor indices get translated into contractions when written in Tile. The most direct (and, sadly, wrong) implementation in Tile is:

C++

```
Tensor wrong_max_pool_1d(const Tensor& I) {
  TensorDim N;
  TensorIndex i, j;
  I.bind_dims(N);
```

(continues on next page)

(continued from previous page)

```
  auto O = TensorOutput(N / 2);
  O(i) >= I(2 * i + j);
  return O;
}
```

Python

```
def wrong_max_pool_1d(I):
   N = TensorDim()
   i, j = TensorIndexes(2)
   I.bind_dims(N)
   O = TensorOutput(N // 2)
   O[i] >= I[2 * i + j]
   return O
```

If you were to run this code, every entry of `O` would equal the global max of `I`. We correctly determined that this was a maximization operation, and the indices for `O` and `I` match those used in the straight C++ code, so what went wrong? The problem with this Tile code is that there are too many "valid" indices. For example, the case `i = 1`, `j = 3` means that `O[1]` checks `I[5]` as one of the potential maximum values, even though `O[1]` is intended to be `max(I[2], I[3])`. When we wrote the code with for loops, the inner loop restricted `j` to `0` or `1`; in the Tile code, the compiler figured out the allowed values of `j` by looking at the shapes of the tensors, and the only restriction that imposes on `j` is that `j` must be an integer satisfying `0 <= 2 * i + j < N`. When can use `add_constraint` in Tile to handle such situations:

Something important to note here is that while we wrote `j < 2`, this constraint actually means `0<= j < 2`. Constraints are always bounded below by `0`. (Without a constraint, however, index variables may still be negative: the original code included e.g. `i = 1, j = -1` as valid index pair.) We determined the Tile code for this example by starting from imperative code, but this Tile code is still very similar to mathematical notation, and we could have started there instead:

$$O[i] = \max_{0 \le j < 2} I[2i + j]$$

```
O(i) |   >= | I(2 * i + j)|;|
O.add_constraint(|j < 2|);|
```

This Tile code handles odd values of `N` by rounding down the output tensor size. You may instead want to round up the output tensor size and use a smaller pool at the edge. This can be accomplished by simply adjusting the size of `O`:

C++

```
Tensor max_pool_1d(const Tensor& I) {
  TensorDim N;
  TensorIndex i, j;
  I.bind_dims(N);
  auto O = TensorOutput((N + 1) / 2);
  O(i) >= I(2 * i + j);
  O.add_constraint(j < 2);
  return O;
}
```

Python

```
def max_pool_1d(I):
  N = TensorDim()
  i, j = TensorIndexes(2)
  I.bind_dims(N)
```

(continues on next page)

(continued from previous page)

```
  O = TensorOutput((N + 1) // 2)
  O[i] >= I[2 * i + j]
  O.add_constraint(j < 2)
  return O
```

No special handling is needed for the case `i = (N - 1) / 2, j = 1`; this is out of range for `I` and so is ignored by Tile, which is exactly the intended behavior.

5.2.7 Valid Indices

When discussing contractions, we've mentioned that they accumulate over "all valid indices". Hopefully the significance of this has been clear for the specific examples we've looked at, but to write complex or novel code it helps to have a precise understanding of what is meant by "valid indices". First, index validity is determined for a full set of index variables: `j = 1` is not valid or invalid as a standalone index value, but may be part of a valid or invalid set of index variables. For example, in the code:

C++

```
I.bind_dims(N);
auto O = TensorOutput((N + 1) / 2);
O(i) >= I(2 * i + j);
O.add_constraint(j < 2);
```

Python

```
I.bind_dims(N)
O = TensorOutput[(N + 1) // 2];
O[i] >= I[2 * i + j];
O.add_constraint(j < 2);
```

with `N = 5`, the indices `i = 1, j = 1` are valid indices. However, `i = 2, j = 1` are not valid indices for this operation, nor are `i = -1000, j = 1`. A set of indices are *valid* if and only if:

1. All the index variables are integers.
2. All the index expressions for every tensor are in range. Specifically, if the index variable values are plugged into every index expression, all the resulting indices are non-negative integers less than the appropriate dimension.
3. All the constraints are satisfied. Constraints always take the form `[index expression] < [constant expression]` (where `[index expression]` is a linear polynomial in the index variables and `[constant expression]` is a linear polynomial in the input dimensions), and they always implicitly include `0 <= [index expression]`. Therefore we could also state this requirement as "every constraint's index expression is non-negative and less than its specified upper bound".

5.2.8 Skipping

The rule that all index variables must be integers allows us to "skip" certain otherwise valid entries. For example, consider the Tile function:

C++

```
Tensor skip(const Tensor& I) {
  TensorDim M, N;
  TensorIndex i, j;
  I.bind_dims(M, N);
```

(continues on next page)

(continued from previous page)

```
  auto O = TensorOutput(N);
  O(2 * i) += I(2 * i, j);
  return O;
}
```

Python

```
def skip(I):
  M, N = TensorDims(2)
  i, j = TensorIndexes(2)
  I.bind_dims(M, N)
  O = TensorOutput(N)
  O[2 * i] += I[2 * i, j]
  return O
```

This operation only writes to even entries of `O`; while `i = 1/2, j = 1` does yield valid index expressions (`O[1]` and `I[1, 1]`), using a fractional index variable `i` makes these indices invalid. Note that some elements of `O` are never written to. Any unwritten elements in the output of a contraction are initialized to `0`.

5.2.9 Cumulative Sum

Suppose we want to take the cumulative sum of a 1D tensor. That is, we want `O[i]` to be the sum of all input entries `I[k]` where `k <= i`. In summation notation, this is:

$$O[i] = \sum_{k \le i} I[k]$$

However, we can't use `k <= i` as a constraint in Tile; all the index variables must be gathered into a single index expression on one side of the inequality. Thus, we rewrite this as `0 <= i - k`. Since the `0` bound is implicitly included in all constraints, we just need to choose an upper bound large enough to never be hit. From the dimensions of the tensors, we already know `i < N` and `0 <= k`, and so `N` is an appropriate upper bound. The resulting Tile code is:

C++

```
Tensor csum(const Tensor& I) {
  TensorDim N;
  TensorIndex i, k;
  I.bind_dims(N);
  auto O = TensorOutput(N);
  O(i) += I(k);
  O.add_constraint(i - k < N);
  return O;
}
```

Python

```
def csum(I):
  N = TensorDim()
  i, k = TensorIndexes(2)
  I.bind_dims(N)
  O = TensorOutput(N)
  O[i] += I[k]
  O.add_constraint(i - k < N)
  return O
```

5.2.10 Convolution

Let's implement a 1D convolution with output size equal to input size. This is implementing the Keras backend operation:

```
K.conv1d(x, kernel, padding='valid')
```

Let's start with the mathematical formula for this operation:

$$O[n, x, c_o] = \sum_k \sum_{c_i} (I[n, x + k, c_i] \cdot K[k, c_i, c_o])$$

This is rather complicated, so let's walk through why this is the same convolution formula we're used to in machine learning. A convolution produces output for a specific batch element at a specific location in a specific channel by taking a weighted sum of the input for that same batch element at that same location *and a surrounding region* over all input channels. The weights are given by `K`, which depends on the output channel, the input channel, and the displacement within the input region relative to the reference location. This generally matches the given formula: The output `O` is given as a sum of elements from the input `I`, weighted by `K`. Looking at the meaning of the index variables, we see that it matches exactly:

- *n* represents which element of the batch we're on.
- *ci* represents which input channel we're on.
- *co* represents which output channel we're on.
- *x* represents our spatial location, giving the location being written to in *O* and the smallest element read from in *I*.
- Finally, *k* represents the kernel offset, that is, how far (in the spatial dimension) the input element we're reading is from the lower bound of the kernel.

This formula directly translates to Tile, although note that `padding='valid'` means that the spatial dimension of the output will be reduced by one less than the kernel size relative to the spatial dimension of the input:

$$O[n, x, c_o] = \sum_k \sum_{c_i} I[n, x + k, c_i] \cdot K[k, c_i, c_o]$$

```
O(n, x, co) |  += | I(n, x + k, ci) |  * | K(k, ci, co) |;|
```

C++

```
Tensor conv_1d(const Tensor& I, const Tensor& K) {
  TensorDim N, X, KX, CI, CO;
  TensorIndex n, x, k, ci, co;
  I.bind_dims(N, X, CI);
  K.bind_dims(KX, CI, CO);
  auto O = TensorOutput(N, X - KX + 1, CO);
  O(n, x, co) += I(n, x + k, ci) * K(k, ci, co);
  return O;
}
```

Python

```
def conv_1d(I, K):
  N, X, KX, CI, CO = TensorDims(5)
  n, x, k, ci, co = TensorIndexes(5)
  I.bind_dims(N, X, CI)
  K.bind_dims(KX, CI, CO)
```

(continues on next page)

(continued from previous page)

```
  O = TensorOutput(N, X - KX + 1, CO)
  O[n, x, co] += I[n, x + k, ci] * K[k, ci, co]
  return O
```

5.2.11 Dilated 2D Convolution

We can tweak this general formula for a convolution to add various features, such as different strides, changing the padding, performing the convolution depthwise, etc. For this example, we will implement a dilated 2D convolution with dilation rate (2, 3). Specfically, we'll implement the Keras backend function:

```
K.conv2d(x, kernel, padding='valid', dilation_rate=(2, 3))
```

The formula for this is very similar to the previous convolution; we just have an additional spatial dimension for each tensor, and the kernel offset index variables are multiplied by dilation scaling factors when used to determine indices for `I`:

$$O[n, x, y, c_o] = \sum_{k_x} \sum_{k_y} \sum_{c_i} I[n, x + 2k_x, y + 3k_y, c_i] * K[k_x, k_y, c_i, c_o]$$

The effective size for a dilated kernel with kernel size `K` and dilation rate `d` is `d * (K - 1) + 1`, and so to achieve *'valid'* padding for this convolution, the x dimension must be reduced by `2 * (KX - 1)` and the y dimension must be reduced by `3 * (KY - 1)`, where `KX` and `KY` are the x and y dimensions of the kernel respectively. The rest of the Tile code corresponds directly to the formula, and so we get:

C++

```
Tensor conv_2d(const Tensor& I, const Tensor& K) {
  TensorDim N, X, Y, KX, KY, CI, CO;
  TensorIndex n, x, y, kx, ky, ci, co;
  I.bind_dims(N, X, Y, CI);
  K.bind_dims(KX, KY, CI, CO);
  auto O = TensorOutput(N, X - 2 * (KX - 1), Y - 3 * (KY - 1), CO);
  O(n, x, y, co) += I(n, x + 2 * kx, y + 3 * ky, ci) * K(kx, ky, ci, co);
  return O;
}
```

Python

```
def conv_2d_dilated(I, K):
  N, X, Y, KX, KY, CI, CO = TensorDims(7)
  n, x, y, kx, ky, ci, co = TensorIndexes(7)
  I.bind_dims(N, X, Y, CI)
  K.bind_dims(KX, KY, CI, CO)
  O = TensorOutput(N, X - 2 * (KX - 1), Y - 3 * (KY - 1), CO)
  O[n, x, y, co] += I[n, x + 2 * kx, y + 3 * ky, ci] * K[kx, ky, ci, co]
  return O
```

5.2.12 Complex Convolution

This final example demonstrates a strided dilated padded grouped convolution.

$$O[n, x_0, x_1, g, c_{o,g}] = \sum_{k_0, k_1, c_{i,g}} (I[n, s_0 x_0 + d_0 k_0 - P_0, s_1 x_1 + d_1 k_1 - P_1, c_{i,g}] * K[k_0, k_1, g, c_{i,g}, c_{o,g}])$$

where `$s$` gives the stride coefficients, `$d$` gives the dilation coefficients, and `$P$` gives the padding offsets.

C++

```
Tensor complex_conv_2d(
  const Tensor& I,
  const Tensor& K,
  const std::vector<size\_t>& s,  // stride coeffs
  const std::vector<size\_t>& d   // dilation coeffs
) {
    // "same-lower" autopadding will be applied
    TensorDim N, G, GCI, GCO;
    std::vector<TensorDim> X(2);
    std::vector<TensorDim> K(2);
    TensorIndex n, g, gci, gco;
    std::vector<TensorIndex> x(2);
    std::vector<TensorIndex> k(2);
    I.bind_dims(N, X[0], X[1], G, GCI);
    K.bind_dims(K[0], K[1], G, GCI, GCO);
    // Compute output spatial dimensions
    std::vector<TensorDim> Y(2);
    for (size_t i = 0; i < Y.size(); ++i) {
      Y[i] = (X[i] + s[i] \- 1) / s[i];
    }
    // Compute the effective kernel size after dilation
    std::vector<TensorDim> EK(2);
    for (size_t i = 0; i < EK.size(); ++i) {
      EK[i] = d[i] \* (K[i] \- 1) + 1;
    }
    // Compute the padding offset
    std::vector<TensorDim> P(2);
    for (size_t i = 0; i < P.size(); ++i) {
      P[i] = ((Y[i] \- 1) \* s[i] + EK[i] \- X[i]) / 2;
    }
    // Specify the output size
    auto O = TensorOutput(N, Y0, Y1, G, GCO);
    // Compute the convolution
    O(n, x[0], x[1], g, gco) +=
      I(n, s[0]\*x[0] + d[0]\*k[0] \- P[0], s[1]\*x[1] + d[1]\*k[1] \- P[1], g, gci) \
↪*
      K(k0, k1, g, gci, gco);
    return O;
}
```

Python

```
def complex_conv_2d(
    I,
    K,
    s0,
```

(continues on next page)

(continued from previous page)

```
    s1,  # stride coeffs
    d0,
    d1  # dilation coeffs
    ):
        # "same-lower" autopadding will be applied
        N, G, GCI, GCO = TensorDims(4)
        X0, X1 = TensorDims(2)
        K0, K1 = TensorDims(2)
        n, g, gci, gco = TensorIndexes(4)
        x0, x1 = TensorIndexes(2)
        k0, k1 = TensorIndexes(2)
        I.bind_dims(N, X0, X1, G, GCI)
        K.bind_dims(K0, K1, G, GCI, GCO)

        # Compute output spatial dimensions
        Y0, Y1 = TensorDims(2)
        Y0 = (X0 + s0 - 1) // s0
        Y1 = (X1 + s1 - 1) // s1

        #Compute the effective kernel size after dilation
        EK0, EK1 = TensorDims(2)
        EK0 = d0 * (K0 - 1) + 1
        EK1 = d1 * (K1 - 1) + 1

        #Compute the padding offset
        P0, P1 = TensorDims(2)
        P0 = ((Y0 - 1) * s0 + EK0 - X0) // 2
        P1 = ((Y1 - 1) * s1 + EK1 - X1) // 2

        # Specify the output size
        O = TensorOutput(N, Y0, Y1, G, GCO)

        # Compute the convolution
        O[n, x0, x1, g, gco] += I[n, s0 * x1 + d0 * k0 - P0, s1 * x1 + d1 * k1 -
                                  P1, g, gci] * K[k0, k1, g, gci, gco]
        return O
```

5.3 Reference

5.3.1 Contractions

There are five *aggregation* operations:

- *operator* += or *sum*: When multiple values are computed for the same output location, they are added together.
- *operator* *= or *product*: when multiple values are computed for the same output location, they are multiplied together.
- *operator* >= or *max*: when multiple values are computed for the same output location, the largest one is used.
- *operator* <= or *min*: when multiple values are computed for the same output location, the smallest one is used.
- *operator* = or *assign*: when multiple values are computed for the same output location, an error is raised. Note that the compiler errs on the side of caution and may raise an error even when no output location is assigned to multiple times. If the programmer manually confirms that there is at most one value computed for each output

location, then any of the other aggregation operations will have equivalent behavior and can be used to bypass this error checking.

There are limited operations available inside a contraction. Principally, contractions allow the use of complex index expressions to determine which elements are read from a tensor. If there is only one tensor used in the contraction, such index manipulations are the only legal options. If there are two tensors used inside the contraction, you also choose a *combination* operation to determine how their values are combined. The only combination operations that are currently well-supported are multiplication (*) and addition (+). Contractions aggregate over all sets of *valid indices*. A set of indices is valid for a contraction if and only if:

- All index variables are integers
- All index expressions used in tensors are within bounds
- All user-specified constraints are satisfied

5.3.2 Elementwise Operations

Elementwise operations never specify indices or dimensions. The shape of the output tensor is inferred from the shape of the input tensor(s). In most binary operations, if the input tensors have different shapes, the output shape is determined by broadcasting together the input shapes. If this is impossible or ambiguous, it is an error. Common operations (not comprehensive; example tensor variable names provided to illustrate syntax):

- Addition: *O = A + B;*
- Subtraction: *O = A - B;*
- Multiplication: *O = A * B;*
- Division: *O = A / B;*
- Equality: *O = A == B;*
- Inequality: *O = A != B;*
- Less: *O = A < B;*
- Square Root: *O = sqrt(A);*
- Exponential: *O = exp(A);*
- Power: *O = pow(A, B);*
- Sine: *O = sin(A);*
- Hyperbolic Tangent: *O = tanh(A);*
- Natural Log: *O = log(A);*
- Sigmoid: *O = sigmoid(A);*
- Conditional: *O = select(C, T, F);* (*C* may be a single value or a higher dimensional tensor to be evaluated elementwise. *T* and *F* must have the same shape, and unless *C* is known to be a constant at compile time, both will be evaluated.)

5.3.3 Types

- *Tensor*: Multidimensional arrays of a fixed shape. The scope of a tensor is the entire function. By convention, tensors begin with a capital letter.
- *TensorDim*: Positive integers initially passed to a function as sizes of input tensors. The scope of a dimension is the entire function. By convention, dimensions begin with a capital letter.
- *TensorIndex*: Symbolic integers used in contractions to directly index a tensor or as part of a formula to compute a tensor index. The scope of an index is a single operation. By convention, indices begin with a lower case letter.

CHAPTER

SIX

API

6.1 Core

PlaidML Core

Contents

6.1.1 Initialization

C++

void `plaidml::init()`
Initializes PlaidML's Core API.

Python

Note: Initialization of PlaidML's Core Python API happens automatically wherever the module `plaidml.core` is imported.

6.1.2 Objects

C++

enum `core_objects::DType`
Enumerates all of the data types in PlaidML.

Values:

INVALID = PLAIDML_DATA_INVALID

BOOLEAN = PLAIDML_DATA_BOOLEAN

INT8 = PLAIDML_DATA_INT8

UINT8 = PLAIDML_DATA_UINT8

INT16 = PLAIDML_DATA_INT16

UINT16 = PLAIDML_DATA_UINT16

INT32 = PLAIDML_DATA_INT32

UINT32 = PLAIDML_DATA_UINT32

INT64 = PLAIDML_DATA_INT64

UINT64 = PLAIDML_DATA_UINT64

BFLOAT16 = PLAIDML_DATA_BFLOAT16

FLOAT16 = PLAIDML_DATA_FLOAT16

FLOAT32 = PLAIDML_DATA_FLOAT32

FLOAT64 = PLAIDML_DATA_FLOAT64

class TensorShape

This is a *TensorShape*.

Public Functions

TensorShape()

TensorShape constructor

TensorShape(DType *dtype*, **const** std::vector<int64_t> &*sizes*)

TensorShape constructor

Parameters

- `dtype`: DType

TensorShape(DType *dtype*, **const** std::vector<int64_t> &*sizes*, **const** std::vector<int64_t> &*strides*)

TensorShape constructor

Parameters

- `dtype`: DType
- `sizes`: const vector<int64_t>
- `strides`: const vector<int64_t>

TensorShape(**const** std::shared_ptr<plaidml_shape> &*ptr*)

TensorShape constructor

Parameters

- `ptr`: const shared_ptr<plaidml_shape>

DType **dtype**() **const**

dtype

Return DType

size_t **ndims**() **const**

Returns the number of dimensions in the *TensorShape*

Return size_t

uint64_t **nbytes**() **const**
nbytes

Return uint64_t

class View
This is a *View*.

Public Functions

char ***data**()
data

size_t **size**()
size

void **writeback**()
writeback

class Buffer
This is a *Buffer*.

Public Functions

Buffer()
Buffer constructor

Buffer(**const** std::string &*device*, **const** *TensorShape* &*shape*)
Buffer constructor

Parameters

- `device`: string
- `shape`: *TensorShape*

Buffer(plaidml_buffer **ptr*, **const** *TensorShape* &*shape*)
Buffer constructor

Parameters

- `ptr`: plaidml_buffer*
- `shape`: *TensorShape*

plaidml_buffer ***as_ptr**() **const**
Returns a pointer to the *Buffer*.

Return plaidml_buffer*

View **mmap_current**()
mmap_current

Return *View*

View **mmap_discard**()
mmap_discard

Return *View*

struct Settings
These are the *Settings*.

Public Static Functions

static std::string **get**(**const** std::string &*key*)
Gets the setting specified by `key`

Return string

Parameters

- `key`: string

static void **set**(**const** std::string &*key*, **const** std::string &*value*)
Sets the setting specified by `key` to the `value` specified.

Parameters

- `key`: string
- `value`: string

Python

6.2 EDSL

Embedded Domain Specific Language

Contents

6.2.1 Initialization

C++

void `plaidml::edsl::init`()
Initializes PlaidML's EDSL API.

Python

Note: Initialization of PlaidML's EDSL Python API happens automatically wherever the module `plaidml.edsl` is imported.

6.2.2 Objects

C++

class Program
This is a program.

Public Functions

Program(**const** std::string &*name*, **const** std::vector<*Tensor*> &*outputs*, **const** std::vector<std::tuple<*Tensor*, *Tensor*>> &*updates* = {})
Program constructor

std::string **str**() **const**
Return the *Program* as a string.

const std::vector<*ProgramArgument*> &**args**() **const**
args

const std::vector<*ProgramArgument*> &**inputs**() **const**
inputs

const std::vector<*ProgramArgument*> &**outputs**() **const**
outputs

class TensorDim
A symbolic object used to specify the dimensions of a *Tensor*

Public Functions

TensorDim()
TensorDim constructor

TensorDim(**const** std::shared_ptr<plaidml_dim_expr> &*ptr*)
TensorDim constructor

TensorDim(int64_t *value*)
TensorDim constructor

TensorDim **operator-**() **const**
Represents a subtraction operator overload.

std::string **str**() **const**
Returns the *TensorDim* as a string.

int64_t **as_int**() **const**
Returns the *TensorDim* as an int.

class TensorIndex
A symbolic object used to directly index a *Tensor* or to compute a *Tensor*'s index as part of a formula.

Public Functions

TensorIndex()
TensorIndex constructor

TensorIndex(int64_t *value*)
TensorIndex constructor

TensorIndex(**const** std::string &*name*)
TensorIndex constructor

TensorIndex **operator-**() **const**
Represents an subtraction operator overload on a *TensorIndex*

Constraint **operator<**(int64_t *rhs*) **const**
TODO

Constraint **operator<**(**const** *TensorDim* &*rhs*) **const**
TODO

std::string **str**() **const**
Returns the *TensorIndex* as a string.

struct Constraint
This is a constraint.

Public Members

TensorIndex **lhs**
lhs

TensorDim **rhs**
rhs

class IndexedTensor
This is an *IndexedTensor*

Public Functions

IndexedTensor &**operator+=**(**const** *IndexedTensor* &*rhs*)
Represents an aggregation_op of SUM in a contraction

IndexedTensor &**operator*=**(**const** *IndexedTensor* &*rhs*)
Represents an aggregation_op of PROD in a contraction

IndexedTensor &**operator>=**(**const** *IndexedTensor* &*rhs*)
Represents an aggregation_op of MAX in a contraction

IndexedTensor &**operator<=**(**const** *IndexedTensor* &*rhs*)
Represents an aggregation_op of MIN in a contraction

IndexedTensor &**operator=**(**const** *IndexedTensor* &*rhs*)
Represents an aggregation_op of ASSIGN in a contraction

IndexedTensor **operator+**(**const** *IndexedTensor* &*rhs*) **const**
Represents a combo_op of PLUS in a contraction

IndexedTensor **operator*** (**const** *IndexedTensor* &*rhs*) **const**
Represents a combo_op of MULTIPLY in a contraction

IndexedTensor **operator==** (**const** *IndexedTensor* &*rhs*) **const**
Represents a combo_op of EQ in a contraction

class LogicalShape
This is a *LogicalShape*.

Public Functions

LogicalShape (DType *dtype*, **const** std::vector<int64_t> &*dims*)
LogicalShape constructor

std::string **str** () **const**
Returns a *LogicalShape* as a string

DType **dtype** () **const**
Returns the datatype of the *LogicalShape*

size_t **ndims** () **const**
Returns the number of dimensions of the *LogicalShape*

std::vector<int64_t> **int_dims** () **const**
Returns the dimensions of the *LogicalShape* as a vector of integers.

bool **operator==** (**const** *LogicalShape* &*rhs*) **const**
TODO

class Tensor
A multidimensional array of a fixed shape.

Public Functions

Tensor ()
Tensor constructor

Tensor (int *value*)
Tensor constructor

Return *Tensor*

Parameters

- `value`: int

Tensor (unsigned *value*)
Tensor constructor

Return *Tensor*

Parameters

- `value`: unsigned int

Tensor (int64_t *value*)
Tensor constructor

Return *Tensor*

Parameters

- `value`: int64_t

Tensor(double *value*)
Tensor constructor

Return *Tensor*

Parameters

- `value`: double

Tensor(**const** *TensorDim* &*dim*)
Tensor constructor

Tensor(**const** std::vector<int64_t> &*dims*)
Tensor constructor

Tensor(**const** std::vector<*TensorDim*> &*dims*)
Tensor constructor

Tensor(**const** std::initializer_list<*TensorDim*> &*dims*)
Tensor constructor

Tensor(**const** std::string &*name*, **const** std::vector<*TensorDim*> &*dims*)
Tensor constructor

Tensor(**const** std::string &*name*, **const** std::initializer_list<*TensorDim*> &*dims*)
Tensor constructor

Tensor(**const** *Tensor* &*rhs*)
Tensor constructor

Tensor &**operator=**(**const** *Tensor* &*rhs*)
Represents an operator overload for = for a *Tensor*

Tensor **operator-**() **const**
Represents an eltwise negation

Tensor **operator~**() **const**
Represents an eltwise bit_not

std::string **str**() **const**
TODO

Tensor &**no_reduce**()
Enable no_reduce on a contraction

Tensor &**use_default**(**const** *Tensor* &*rhs*)
Set use_default on a contraction

Tensor &**add_constraint**(**const** *Constraint* &*constraint*)
TODO

Tensor &**add_constraints**(**const** std::vector<*Constraint*> &*constraints*)
TODO

LogicalShape **shape**() **const**
Return the tensor's shape

void **bind_dims**(**const** std::vector<*TensorDim*> &*dims*) **const**
TODO

template<typename ...**Ts**>
void **bind_dims**(*Ts*... *dims*) **const**
TODO

struct TensorRef
A reference to a *Tensor*

Public Functions

TensorRef(**const** *Tensor* &*tensor*)
TensorRef constructor

operator Tensor() **const**
TODO

bool **operator<**(**const** *TensorRef* &*rhs*) **const**
TODO

bool **operator==**(**const** *TensorRef* &*rhs*) **const**
TODO

Public Members

Tensor **tensor**
The `Tensor` that the `TensorRef` is referencing

struct ProgramArgument
Description for *ProgramArgument*

Public Members

bool **is_input**
TODO

TensorRef **tensor**
TODO

LogicalShape **shape**
TODO

std::shared_ptr<*Buffer*> **buffer**
TODO

Python

6.2.3 Primitives

C++

Tensor `plaidml::edsl::`**`abs`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise absolute value of `x`.

Return *Tensor*

Parameters

- `x`: *Tensor*

Tensor `plaidml::edsl::`**`cast`**`(`**`const`** *Tensor* &*x*, DType *dtype*`)`
Casts the element type of a tensor `x` to the type specified by `dtype`.

Return *Tensor*

Parameters

- `x`: *Tensor*
- `dtype`: DType

Tensor `plaidml::edsl::`**`ceil`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise ceiling of `x`.

Return *Tensor*

Parameters

- `x`: *Tensor*

Tensor `plaidml::edsl::`**`cos`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise cosine of `x`.

Return *Tensor*

Parameters

- `x`: *Tensor*

Tensor `plaidml::edsl::`**`cosh`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise hyperbolic cosine of `x`.

Return *Tensor*

Parameters

- `x`: *Tensor*

Tensor `plaidml::edsl::`**`exp`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise natural exponential function of `x`: e^x.

Return *Tensor*

Parameters

- `x`: *Tensor*

Tensor `plaidml::edsl::`**`floor`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise floor of `x`.

Return *Tensor*

Parameters

- `x`: *Tensor*

Tensor `plaidml::edsl::gather`(**const** *Tensor* &*x*, **const** *Tensor* &*y*)
Takes an input tensor (`x`) and a set of indices to gather over (`y`), and returns an output tensor that gathers the input tensor from the indices specified.

Return *Tensor*

Parameters

- `x`: *Tensor*
- `y`: *Tensor*

Tensor `plaidml::edsl::ident`(**const** *Tensor* &*x*)
Returns the identity of `x`.

Return *Tensor*

Parameters

- `x`: *Tensor*

Tensor `plaidml::edsl::index`(**const** *Tensor* &*x*, size_t *axis*)
Returns the index of `x` at the specified `axis`.

Return *Tensor*

Parameters

- `x`: *Tensor*
- `axis`: size_t

Tensor `plaidml::edsl::log`(**const** *Tensor* &*x*)
Computes the elementwise natural logarithm of `x`: *ln*(`x`).

Return *Tensor*

Parameters

- `x`: *Tensor*

Tensor `plaidml::edsl::pow`(**const** *Tensor* &*x*, **const** *Tensor* &*y*)
Computes the elementwise `y`th power of `x`.

Return *Tensor*

Parameters

- `x`: *Tensor*
- `y`: *Tensor*

Tensor `plaidml::edsl::prng`(**const** *Tensor* &*state*, **const** std::vector<int64_t> &*dims*)
Generates a *Tensor* of elementwise pseudorandom numbers using the seed values specified in `state`.

Return *Tensor*

Parameters

- `state`: *Tensor*
- `dims`: vector<int64_t>

Tensor `plaidml::edsl::reshape`(**const** *Tensor* &*x*, **const** std::vector<int64_t> &*dims*)
Takes an input tensor `x` and reshapes it according to `dims`.

Return *Tensor*

Parameters

- x: *Tensor*
- dims: vector<int64_t>

Tensor plaidml::edsl::**reshape**(**const** *Tensor* &*x*, **const** std::vector<*TensorDim*> &*dims*)
Takes an input tensor x and reshapes it according to dims.

Return *Tensor*

Parameters

- x: *Tensor*
- dims: vector<TensorDim>

Tensor plaidml::edsl::**round**(**const** *Tensor* &*x*)
Rounds x elementwise.

Return *Tensor*

Parameters

- x: *Tensor*

Tensor plaidml::edsl::**scatter**(**const** *Tensor* &*x*, **const** *Tensor* &*y*, **const** *Tensor* &*z*)
Takes an input tensor (x), a set of indices to scatter over (y), and the number of elements in the scattered tensor (z), and returns an output tensor that scatters the input tensor across the number of elements specified.

Return *Tensor*

Parameters

- x: *Tensor*
- y: *Tensor*
- z: *Tensor*

Tensor plaidml::edsl::**select**(**const** *Tensor* &*cond*, **const** *Tensor* &*true_case*, **const** *Tensor* &*false_case*)
Performs an elementwise conditional which returns the corresponding element in true_case if the condition is evaluated to be true or the corresponding element in false_case if the condition is evaluated to be false.

Return *Tensor*

Parameters

- cond: *Tensor*
- true_case: *Tensor*
- false_case: *Tensor*

Tensor plaidml::edsl::**shape**(**const** *Tensor* &*x*)
Returns the shape of x as a *Tensor*.

Return *Tensor*

Parameters

- x: *Tensor*

Tensor plaidml::edsl::**sin**(**const** *Tensor* &*x*)
Computes the elementwise sine of x.

Return *Tensor*

Parameters

- x: *Tensor*

Tensor `plaidml::edsl::`**`sinh`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise hyperbolic sine of `x`.

Return *Tensor*

Parameters

- x: *Tensor*

Tensor `plaidml::edsl::`**`sqrt`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise square root of `x`.

Return *Tensor*

Parameters

- x: *Tensor*

Tensor `plaidml::edsl::`**`tan`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise tangent of `x`.

Return *Tensor*

Parameters

- x: *Tensor*

Tensor `plaidml::edsl::`**`tanh`**`(`**`const`** *Tensor* &*x*`)`
Computes the elementwise hyperbolic tangent of `x`.

Return *Tensor*

Parameters

- x: *Tensor*

Tensor `plaidml::edsl::`**`zero`**`()`
Returns a *Tensor* with a value of 0.

Return *Tensor*

Python

6.2.4 Examples

```
Tensor sum_over_axis(const Tensor& I) {
   TensorDim M, N;
   TensorIndex m, n;
   I.bind_dims(M, N);
   auto O = TensorOutput(N);
   O(n) += I(m, n); // contraction
   return O;
}
```

$$O[n] = \sum_{m} I[m, n]$$

```
O(n) |  += | I(m, n)|;|
```

6.3 Execution

PlaidML Execution

Contents

6.3.1 Initialization

C++

void `plaidml::exec::init()`
Initializes PlaidML's Execution API.

Python

Note: Initialization of PlaidML's Execution Python API happens automatically wherever the module `plaidml.exec` is imported.

6.3.2 Objects

C++

`struct Binding`
Bindings bind a Tensor to a *Buffer*.

Public Members

edsl::*Tensor* **`tensor`**
The tensor to bind.

Buffer **`buffer`**
The buffer to be bound to.

`class Executable`
This is an *Executable*.

Public Functions

Executable(**const** edsl::*Program* &*program*, **const** std::vector<*Binding*> &*inputs*, **const** std::vector<*Binding*> &*outputs*)
Executable constructor

Executable(**const** edsl::*Program* &*program*, **const** std::string &*device*, **const** std::string &*target*, **const** std::vector<*Binding*> &*inputs*, **const** std::vector<*Binding*> &*outputs*)
Executable constructor

void **run**()
run

struct Binder
This is a *Binder*.

Public Functions

Binder(**const** edsl::*Program* &*program*)
Constructs a *Binder*. By default, this constructor uses the environment variables `PLAIDML_DEVICE` and `PLAIDML_TARGET` to specify your device and target. You can override these using the `set_device` and `set_target` functions

Binder &**set_device**(**const** std::string &*value*)
Set the device for the *Binder* to use.

Return *Binder*

Parameters

- `value`: string

Binder &**set_target**(**const** std::string &*value*)
Set the target for the *Binder* to use.

Return *Binder*

Parameters

- `value`: string

Buffer **input**(**const** edsl::*Tensor* &*tensor*)
input

Return *Buffer*

Parameters

- `tensor`: Tensor

Buffer **output**(**const** edsl::*Tensor* &*tensor*)
output

Return *Buffer*

Parameters

- `tensor`: Tensor

Binder &**set_input**(**const** edsl::*Tensor* &*tensor*, **const** *Buffer* &*buffer*)
set_input

Return *Binder*

Parameters

- `tensor`: Tensor
- `buffer`: *Buffer*

Binder &**set_output**(**const** edsl::*Tensor* &*tensor*, **const** *Buffer* &*buffer*)
set_output

Return *Binder*

Parameters

- `tensor`: Tensor
- `buffer`: *Buffer*

std::shared_ptr<*Executable*> **compile**()
compile

Return shared_ptr<Executable>

Python

CHAPTER

SEVEN

CONFIGURATION

If you want to use PlaidML from the command line, you can set the following environment variables to select the proper configurations for your device. This is equivalent to running *plaidml-setup* and selecting these settings when prompted.

- *PLAIDML_EXPERIMENTAL* - (0 or 1) determines whether to enable experimental mode in PlaidML
- *PLAIDML*DEVICE*IDS* - (string) the name of the device to use with PlaidML (to see a list of devices, run `plaidml\-setup`)

Below is an example of how to set the device configuration environment variables for PlaidML. .. code-block:

```
export PLAIDML_EXPERIMENTAL=1
export PLAIDML*DEVICE*IDS=opencl*intel*uhd*graphics*630.0
```

INDEX

C

P