AMD MIGraphX Documentation

Release 2.9.0

Advanced Micro Devices, Inc.

May 18, 2024

CONTENTS

Welcome to the MIGraphX docs home page! To learn more, see *What is MIGraphX?*.

Our documentation is structured as follows:

API reference

- *C++ API reference*
- *Python API reference*

Command-line tool

- *MIGraphX driver*

Tutorial

- *Contributing to MIGraphX*

CHAPTER

ONE

CONTRIBUTING TO THIS DOCUMENTATION

We welcome collaboration! If you'd like to contribute to our documentation, you can find instructions on our Contribute to ROCm docs page. Known issues are listed on GitHub. Licensing information for all ROCm components is listed on our Licensing page.

CHAPTER

TWO

INDEX AND SEARCH

- genindex
- search

CHAPTER

THREE

WHAT IS MIGRAPHX?

AMD MIGraphX is AMD's graph inference engine that accelerates machine learning model inference. The optimized execution engine is useful for deep learning neural networks.

You can utilize MIGraphX functions using *C++ APIs*, *Python APIs*, and the *command-line tool migraphx-driver*.

CHAPTER

FOUR

MIGRAPHX DRIVER

The MIGraphX driver is a command-line tool that allows you to utilize many of the MIGraphX core functions without having to write a program. It can read, compile, run, and test the performance of a model with randomized data.

It is installed by default when you install MIGraphX. You can find it in `/opt/rocm/bin/migraphx-driver` or in `AMDMIGraphX/build/bin/migraphx-driver` after building the source code.

4.1 Commands

The table below summarizes the MIGraphX driver commands.

Table 4.1: commands

Command	Description
op	Prints all operators of MIGraphX when followed by the option `--list` or `-l`
params	Prints the input and output parameter shapes
run	Compiles, allocates parameters, evaluates, and prints input graph
read	Loads and prints input graph
compile	Compiles and prints input graph
verify	Runs reference and GPU implementations and checks outputs for consistency
perf	Compiles and runs input graph followed by printing the performance report

4.2 Options

The table below summarizes the various options to be used with the *MIGraphX driver commands*. To learn which options can be used with which commands, see the *MIGraphX driver options*.

Table 4.2: commands

Option	Description
–help | -h	Prints help section.
–model <resnet50|inceptionv3|alexnet>	Loads one of the three default models.
–onnx	Loads the file as an ONNX graph.
–tf	Loads the file as a tensorflow graph.
–migraphx	Loads the file as a migraphx graph.
–migraphx-json	Loads the file as a migraphx JSON graph.
–batch	Sets batch size for a static model. Sets the batch size at runtime for a dynamic batch model.
–nhwc	Treats tensorflow format as nhwc.

continues on next page

Table 4.2 – continued from previous page

–nchw	Treats tensorflow format as nchw.
–skip-unknown-operators	Skips unknown operators when parsing and continues to parse.
–trim | -t	Trims instructions from the end.
–optimize | -O	Optimizes read
–graphviz | -g	Prints a graphviz representation
–brief	Makes the output brief
–cpp	Prints the program in .cpp format
–json	Prints the program in .json format
–text	Prints the program in .txt format
–binary	Prints the program in binary format
–output | -o	Writes output in a file
–fill0	Fills parameter with 0s
–fill1	Fills parameter with 1s
–input-dim	Sets static dimensions of a parameter
–dyn-input-dim	Sets dynamic dimensions of a parameter
–default-dyn-dim	Sets default dynamic dimension
–gpu	Compiles on the GPU
–cpu	Compiles on the CPU
–ref	Compiles on the reference implementation
–enable-offload-copy	Enables implicit offload copying
–disable-fast-math	Disables fast math optimization
–exhaustive-tune	Enables exhaustive search to find the fastest kernel
–fp16	Quantizes for fp16
–int8	Quantizes for int8
–fp8	Quantize for `Float8E4M3FNUZ` type
–rms-tol	Sets tolerance for the RMS error (Default: 0.001)
–atol	Sets tolerance for elementwise absolute difference (Default: 0.001)
–rtol	Sets tolerance for elementwise relative difference (Default: 0.001)
–per-instruction | -i	Verifies each instruction
–reduce | -r	Reduces program and verifies
–iterations | -n	Sets the number of iterations to run for perf report
–list | -l	Lists all the MIGraphX operators

4.3 Usage

This section demonstrates the usage of MIGraphX driver tool with some commonly used options. Note that these examples use a simple MNIST ConvNet as the input graph for demonstration purposes as models of higher complexity generate considerably larger outputs in most cases.

4.3.1 Option: op

$ /opt/rocm/bin/migraphx-driver op –list

View Output

```
@literal
@param
@return
abs
```

(continues on next page)

(continued from previous page)

```
acos
acosh
add
argmax
argmin
as_shape
asin
asinh
atan
atanh
batch_norm_inference
broadcast
capture
ceil
check_context::migraphx::gpu::context
clip
concat
contiguous
convert
convolution
cos
cosh
deconvolution
div
dot
elu
equal
erf
exp
flatten
floor
gather
gpu::abs
gpu::acos
gpu::acosh
gpu::add
gpu::add_clip
gpu::add_gelu
gpu::add_gelu_new
gpu::add_relu
gpu::add_tanh
gpu::argmax
gpu::argmin
gpu::asin
gpu::asinh
gpu::atan
gpu::atanh
gpu::batch_norm_inference
gpu::ceil
gpu::clip
gpu::concat
gpu::contiguous
```

(continues on next page)

(continued from previous page)

```
gpu::conv_bias
gpu::conv_bias_relu
gpu::convert
gpu::convolution
gpu::cos
gpu::cosh
gpu::deconv
gpu::div
gpu::elu
gpu::equal
gpu::erf
gpu::exp
gpu::floor
gpu::gather
gpu::gelu
gpu::gelu_new
gpu::gemm
gpu::greater
gpu::layernorm
gpu::leaky_relu
gpu::less
gpu::log
gpu::logsoftmax
gpu::lrn
gpu::max
gpu::min
gpu::mul
gpu::mul_add
gpu::mul_add_relu
gpu::pad
gpu::pooling
gpu::pow
gpu::prelu
gpu::quant_convolution
gpu::quant_gemm
gpu::recip
gpu::record_event
gpu::reduce_max
gpu::reduce_mean
gpu::reduce_min
gpu::reduce_prod
gpu::reduce_sum
gpu::relu
gpu::rnn_var_sl_last_output
gpu::rnn_var_sl_shift_output
gpu::rnn_var_sl_shift_sequence
gpu::round
gpu::rsqrt
gpu::set_stream
gpu::sigmoid
gpu::sign
gpu::sin
```

(continues on next page)

(continued from previous page)

```
gpu::sinh
gpu::softmax
gpu::sqdiff
gpu::sqrt
gpu::sub
gpu::tan
gpu::tanh
gpu::triadd
gpu::triadd_clip
gpu::triadd_relu
gpu::triadd_sigmoid
gpu::triadd_tanh
gpu::wait_event
greater
gru
hip::allocate
hip::copy
hip::copy_from_gpu
hip::copy_to_gpu
hip::hip_allocate_memory
hip::hip_copy_literal
identity
im2col
leaky_relu
less
load
log
logsoftmax
lrn
lstm
max
min
mul
multibroadcast
neg
outline
pad
pooling
pow
prelu
quant_convolution
quant_dot
recip
reduce_max
reduce_mean
reduce_min
reduce_prod
reduce_sum
ref::batch_norm_inference
ref::convolution
ref::deconvolution
ref::dot
```

(continues on next page)

(continued from previous page)

```
ref::elu
ref::im2col
ref::leaky_relu
ref::logsoftmax
ref::lrn
ref::op
ref::pad
ref::pooling_average
ref::pooling_max
ref::quant_convolution
ref::rnn_var_sl_last_output
ref::softmax
relu
reshape
rnn
rnn_last_cell_output
rnn_last_hs_output
rnn_var_sl_last_output
rnn_var_sl_shift_output
rnn_var_sl_shift_sequence
round
rsqrt
scalar
sigmoid
sign
sin
sinh
slice
softmax
sqdiff
sqrt
squeeze
sub
tan
tanh
transpose
undefined
unknown:
unsqueeze
```

4.3.2 Option: params

> $ /opt/rocm/bin/migraphx-driver params simple_graph.pb

View Output

```
Reading: simple_graph.pb
x: float_type, {1, 28, 28}, {784, 28, 1}
```

4.3.3 Option: run (ONNX file input)

$ /opt/rocm/bin/migraphx-driver run –onnx simple_graph.onnx

View Output

```
Compiling ...
Reading: simple_graph.onnx
@0 = check_context::migraphx::gpu::context -> float_type, {}, {}
@1 = hip::hip_allocate_memory[shape=float_type, {256}, {1},id=scratch] -> float_type,
↪{256}, {1}
@2 = hip::hip_copy_literal[id=@literal:1] -> float_type, {784, 128}, {128, 1}
x:0 = @param:x:0 -> float_type, {1, 28, 28}, {784, 28, 1}
@3 = reshape[dims={-1, 784}](x:0) -> float_type, {1, 784}, {784, 1}
@4 = load[offset=0,end=512](@1) -> float_type, {1, 128}, {128, 1}
@5 = gpu::gemm[alpha=1,beta=0](@3,@2,@4) -> float_type, {1, 128}, {128, 1}
@6 = hip::hip_copy_literal[id=@literal:0] -> float_type, {128}, {1}
@7 = hip::hip_copy_literal[id=@literal:2] -> float_type, {10}, {1}
@8 = hip::hip_copy_literal[id=@literal:3] -> float_type, {128, 10}, {10, 1}
@9 = multibroadcast[output_lens={1, 128}](@6) -> float_type, {1, 128}, {0, 1}
@10 = load[offset=512,end=1024](@1) -> float_type, {1, 128}, {128, 1}
@11 = gpu::add_relu(@5,@9,@10) -> float_type, {1, 128}, {128, 1}
@12 = load[offset=0,end=40](@1) -> float_type, {1, 10}, {10, 1}
@13 = gpu::gemm[alpha=1,beta=0](@11,@8,@12) -> float_type, {1, 10}, {10, 1}
@14 = multibroadcast[output_lens={1, 10}](@7) -> float_type, {1, 10}, {0, 1}
@15 = load[offset=40,end=80](@1) -> float_type, {1, 10}, {10, 1}
@16 = gpu::add(@13,@14,@15) -> float_type, {1, 10}, {10, 1}
#output_0 = @param:#output_0 -> float_type, {1, 10}, {10, 1}
@17 = gpu::softmax[axis=1](@16,#output_0) -> float_type, {1, 10}, {10, 1}
@18 = @return(@17)

Allocating params ...
@0 = check_context::migraphx::gpu::context -> float_type, {}, {}
@1 = hip::hip_allocate_memory[shape=float_type, {256}, {1},id=scratch] -> float_type,
↪{256}, {1}
@2 = hip::hip_copy_literal[id=@literal:1] -> float_type, {784, 128}, {128, 1}
x:0 = @param:x:0 -> float_type, {1, 28, 28}, {784, 28, 1}
@3 = reshape[dims={-1, 784}](x:0) -> float_type, {1, 784}, {784, 1}
@4 = load[offset=0,end=512](@1) -> float_type, {1, 128}, {128, 1}
@5 = gpu::gemm[alpha=1,beta=0](@3,@2,@4) -> float_type, {1, 128}, {128, 1}
@6 = hip::hip_copy_literal[id=@literal:0] -> float_type, {128}, {1}
@7 = hip::hip_copy_literal[id=@literal:2] -> float_type, {10}, {1}
@8 = hip::hip_copy_literal[id=@literal:3] -> float_type, {128, 10}, {10, 1}
@9 = multibroadcast[output_lens={1, 128}](@6) -> float_type, {1, 128}, {0, 1}
@10 = load[offset=512,end=1024](@1) -> float_type, {1, 128}, {128, 1}
@11 = gpu::add_relu(@5,@9,@10) -> float_type, {1, 128}, {128, 1}
@12 = load[offset=0,end=40](@1) -> float_type, {1, 10}, {10, 1}
@13 = gpu::gemm[alpha=1,beta=0](@11,@8,@12) -> float_type, {1, 10}, {10, 1}
@14 = multibroadcast[output_lens={1, 10}](@7) -> float_type, {1, 10}, {0, 1}
@15 = load[offset=40,end=80](@1) -> float_type, {1, 10}, {10, 1}
@16 = gpu::add(@13,@14,@15) -> float_type, {1, 10}, {10, 1}
#output_0 = @param:#output_0 -> float_type, {1, 10}, {10, 1}
@17 = gpu::softmax[axis=1](@16,#output_0) -> float_type, {1, 10}, {10, 1}
@18 = @return(@17)
```

4.3.4 Option: read

$ /opt/rocm/bin/migraphx-driver read simple_graph.pb

View Output

```
Reading: simple_graph.pb
@0 = @literal{0.0136018, -0.0839988, 0.0375392, 0.0613085, -0.125795, 0.176185, 0.
↪0761055, 0.0093384, -0.110057, -0.170587} -> float_type, {10}, {1}
@1 = @literal{ ... } -> float_type, {128, 10}, {10, 1}
@2 = @literal{ ... } -> float_type, {128}, {1}
@3 = @literal{ ... } -> float_type, {784, 128}, {128, 1}
@4 = @literal{-1, 784} -> int32_type, {2}, {1}
x = @param:x -> float_type, {1, 28, 28}, {784, 28, 1}
@5 = reshape[dims={-1, 784}](x) -> float_type, {1, 784}, {784, 1}
@6 = identity(@3) -> float_type, {784, 128}, {128, 1}
@7 = dot[alpha=1,beta=1](@5,@6) -> float_type, {1, 128}, {128, 1}
@8 = identity(@2) -> float_type, {128}, {1}
@9 = broadcast[axis=1,dims={1, 128}](@8) -> float_type, {1, 128}, {0, 1}
@10 = add(@7,@9) -> float_type, {1, 128}, {128, 1}
@11 = relu(@10) -> float_type, {1, 128}, {128, 1}
@12 = identity(@1) -> float_type, {128, 10}, {10, 1}
@13 = dot[alpha=1,beta=1](@11,@12) -> float_type, {1, 10}, {10, 1}
@14 = identity(@0) -> float_type, {10}, {1}
@15 = broadcast[axis=1,dims={1, 10}](@14) -> float_type, {1, 10}, {0, 1}
@16 = add(@13,@15) -> float_type, {1, 10}, {10, 1}
@17 = softmax[axis=1](@16) -> float_type, {1, 10}, {10, 1}
@18 = identity(@17) -> float_type, {1, 10}, {10, 1}
```

4.3.5 Option: compile (on GPU, quantized for fp16)

$ /opt/rocm/bin/migraphx-driver compile –gpu –fp16 simple_graph.pb

View Output

```
Compiling ...
Reading: simple_graph.pb
@0 = check_context::migraphx::gpu::context -> float_type, {}, {}
@1 = hip::hip_allocate_memory[shape=float_type, {456}, {1},id=scratch] -> float_type,
↪{456}, {1}
@2 = hip::hip_copy_literal[id=@literal:0] -> half_type, {784, 128}, {128, 1}
@3 = load[offset=256,end=1824](@1) -> half_type, {1, 28, 28}, {784, 28, 1}
x = @param:x -> float_type, {1, 28, 28}, {784, 28, 1}
@4 = gpu::convert[target_type=1](x,@3) -> half_type, {1, 28, 28}, {784, 28, 1}
@5 = reshape[dims={-1, 784}](@4) -> half_type, {1, 784}, {784, 1}
@6 = load[offset=0,end=256](@1) -> half_type, {1, 128}, {128, 1}
@7 = gpu::gemm[alpha=1,beta=0](@5,@2,@6) -> half_type, {1, 128}, {128, 1}
@8 = hip::hip_copy_literal[id=@literal:2] -> half_type, {128, 10}, {10, 1}
@9 = hip::hip_copy_literal[id=@literal:1] -> half_type, {128}, {1}
@10 = hip::hip_copy_literal[id=@literal:3] -> half_type, {10}, {1}
@11 = load[offset=256,end=512](@1) -> half_type, {1, 128}, {128, 1}
@12 = broadcast[axis=1,dims={1, 128}](@9) -> half_type, {1, 128}, {0, 1}
@13 = gpu::add_relu(@7,@12,@11) -> half_type, {1, 128}, {128, 1}
```

(continues on next page)

(continued from previous page)

```
@14 = load[offset=0,end=20](@1) -> half_type, {1, 10}, {10, 1}
@15 = gpu::gemm[alpha=1,beta=0](@13,@8,@14) -> half_type, {1, 10}, {10, 1}
@16 = broadcast[axis=1,dims={1, 10}](@10) -> half_type, {1, 10}, {0, 1}
@17 = load[offset=20,end=40](@1) -> half_type, {1, 10}, {10, 1}
@18 = gpu::add(@15,@16,@17) -> half_type, {1, 10}, {10, 1}
@19 = load[offset=0,end=20](@1) -> half_type, {1, 10}, {10, 1}
@20 = gpu::softmax[axis=1](@18,@19) -> half_type, {1, 10}, {10, 1}
output = @param:output -> float_type, {1, 10}, {10, 1}
@21 = gpu::convert[target_type=2](@20,output) -> float_type, {1, 10}, {10, 1}
```

4.3.6 Option: verify

$ /opt/rocm/bin/migraphx-driver verify simple_graph.pb

View Output

```
Reading: simple_graph.pb
@0 = @literal{0.0136018, -0.0839988, 0.0375392, 0.0613085, -0.125795, 0.176185, 0.
↪0761055, 0.0093384, -0.110057, -0.170587} -> float_type, {10}, {1}
@1 = @literal{ ... } -> float_type, {128, 10}, {10, 1}
@2 = @literal{ ... } -> float_type, {128}, {1}
@3 = @literal{ ... } -> float_type, {784, 128}, {128, 1}
@4 = @literal{-1, 784} -> int32_type, {2}, {1}
x = @param:x -> float_type, {1, 28, 28}, {784, 28, 1}
@5 = reshape[dims={-1, 784}](x) -> float_type, {1, 784}, {784, 1}
@6 = identity(@3) -> float_type, {784, 128}, {128, 1}
@7 = dot[alpha=1,beta=1](@5,@6) -> float_type, {1, 128}, {128, 1}
@8 = identity(@2) -> float_type, {128}, {1}
@9 = broadcast[axis=1,dims={1, 128}](@8) -> float_type, {1, 128}, {0, 1}
@10 = add(@7,@9) -> float_type, {1, 128}, {128, 1}
@11 = relu(@10) -> float_type, {1, 128}, {128, 1}
@12 = identity(@1) -> float_type, {128, 10}, {10, 1}
@13 = dot[alpha=1,beta=1](@11,@12) -> float_type, {1, 10}, {10, 1}
@14 = identity(@0) -> float_type, {10}, {1}
@15 = broadcast[axis=1,dims={1, 10}](@14) -> float_type, {1, 10}, {0, 1}
@16 = add(@13,@15) -> float_type, {1, 10}, {10, 1}
@17 = softmax[axis=1](@16) -> float_type, {1, 10}, {10, 1}
@18 = identity(@17) -> float_type, {1, 10}, {10, 1}

@0 = @literal{0.0136018, -0.0839988, 0.0375392, 0.0613085, -0.125795, 0.176185, 0.
↪0761055, 0.0093384, -0.110057, -0.170587} -> float_type, {10}, {1}
@1 = @literal{ ... } -> float_type, {128, 10}, {10, 1}
@2 = @literal{ ... } -> float_type, {128}, {1}
@3 = @literal{ ... } -> float_type, {784, 128}, {128, 1}
@4 = @literal{-1, 784} -> int32_type, {2}, {1}
x = @param:x -> float_type, {1, 28, 28}, {784, 28, 1}
@5 = reshape[dims={-1, 784}](x) -> float_type, {1, 784}, {784, 1}
@6 = identity(@3) -> float_type, {784, 128}, {128, 1}
@7 = dot[alpha=1,beta=1](@5,@6) -> float_type, {1, 128}, {128, 1}
@8 = identity(@2) -> float_type, {128}, {1}
@9 = broadcast[axis=1,dims={1, 128}](@8) -> float_type, {1, 128}, {0, 1}
```

(continues on next page)

(continued from previous page)

```
@10 = add(@7,@9) -> float_type, {1, 128}, {128, 1}
@11 = relu(@10) -> float_type, {1, 128}, {128, 1}
@12 = identity(@1) -> float_type, {128, 10}, {10, 1}
@13 = dot[alpha=1,beta=1](@11,@12) -> float_type, {1, 10}, {10, 1}
@14 = identity(@0) -> float_type, {10}, {1}
@15 = broadcast[axis=1,dims={1, 10}](@14) -> float_type, {1, 10}, {0, 1}
@16 = add(@13,@15) -> float_type, {1, 10}, {10, 1}
@17 = softmax[axis=1](@16) -> float_type, {1, 10}, {10, 1}
@18 = identity(@17) -> float_type, {1, 10}, {10, 1}

@0 = @literal{0.0136018, -0.0839988, 0.0375392, 0.0613085, -0.125795, 0.176185, 0.
↪0761055, 0.0093384, -0.110057, -0.170587} -> float_type, {10}, {1}
@1 = @literal{ ... } -> float_type, {128, 10}, {10, 1}
@2 = @literal{ ... } -> float_type, {128}, {1}
@3 = @literal{ ... } -> float_type, {784, 128}, {128, 1}
x = @param:x -> float_type, {1, 28, 28}, {784, 28, 1}
@4 = ref::reshape[dims={-1, 784}](x) -> float_type, {1, 784}, {784, 1}
@5 = ref::identity(@3) -> float_type, {784, 128}, {128, 1}
@6 = ref::dot[alpha=1,beta=1](@4,@5) -> float_type, {1, 128}, {128, 1}
@7 = ref::identity(@2) -> float_type, {128}, {1}
@8 = ref::broadcast[axis=1,dims={1, 128}](@7) -> float_type, {1, 128}, {0, 1}
@9 = ref::contiguous(@8) -> float_type, {1, 128}, {128, 1}
@10 = ref::add(@6,@9) -> float_type, {1, 128}, {128, 1}
@11 = ref::relu(@10) -> float_type, {1, 128}, {128, 1}
@12 = ref::identity(@1) -> float_type, {128, 10}, {10, 1}
@13 = ref::dot[alpha=1,beta=1](@11,@12) -> float_type, {1, 10}, {10, 1}
@14 = ref::identity(@0) -> float_type, {10}, {1}
@15 = ref::broadcast[axis=1,dims={1, 10}](@14) -> float_type, {1, 10}, {0, 1}
@16 = ref::contiguous(@15) -> float_type, {1, 10}, {10, 1}
@17 = ref::add(@13,@16) -> float_type, {1, 10}, {10, 1}
@18 = ref::softmax[axis=1](@17) -> float_type, {1, 10}, {10, 1}
@19 = ref::identity(@18) -> float_type, {1, 10}, {10, 1}

@0 = check_context::migraphx::gpu::context -> float_type, {}, {}
@1 = hip::hip_allocate_memory[shape=float_type, {256}, {1},id=scratch] -> float_type,
↪{256}, {1}
@2 = hip::hip_copy_literal[id=@literal:3] -> float_type, {784, 128}, {128, 1}
x = @param:x -> float_type, {1, 28, 28}, {784, 28, 1}
@3 = load[offset=0,end=512](@1) -> float_type, {1, 128}, {128, 1}
@4 = reshape[dims={-1, 784}](x) -> float_type, {1, 784}, {784, 1}
@5 = gpu::gemm[alpha=1,beta=0](@4,@2,@3) -> float_type, {1, 128}, {128, 1}
@6 = hip::hip_copy_literal[id=@literal:1] -> float_type, {128, 10}, {10, 1}
@7 = hip::hip_copy_literal[id=@literal:2] -> float_type, {128}, {1}
@8 = hip::hip_copy_literal[id=@literal:0] -> float_type, {10}, {1}
@9 = load[offset=512,end=1024](@1) -> float_type, {1, 128}, {128, 1}
@10 = broadcast[axis=1,dims={1, 128}](@7) -> float_type, {1, 128}, {0, 1}
@11 = gpu::add_relu(@5,@10,@9) -> float_type, {1, 128}, {128, 1}
@12 = load[offset=40,end=80](@1) -> float_type, {1, 10}, {10, 1}
@13 = gpu::gemm[alpha=1,beta=0](@11,@6,@12) -> float_type, {1, 10}, {10, 1}
@14 = load[offset=0,end=40](@1) -> float_type, {1, 10}, {10, 1}
@15 = broadcast[axis=1,dims={1, 10}](@8) -> float_type, {1, 10}, {0, 1}
@16 = gpu::add(@13,@15,@14) -> float_type, {1, 10}, {10, 1}
```

(continues on next page)

(continued from previous page)

```
output = @param:output -> float_type, {1, 10}, {10, 1}
@17 = gpu::softmax[axis=1](@16,output) -> float_type, {1, 10}, {10, 1}
```

4.3.7 Option: perf

$ /opt/rocm/bin/migraphx-driver perf simple_graph.pb

View Output

```
Compiling ...
Reading: simple_graph.pb
@0 = check_context::migraphx::gpu::context -> float_type, {}, {}
@1 = hip::hip_allocate_memory[shape=float_type, {256}, {1},id=scratch] -> float_type,
↪{256}, {1}
@2 = hip::hip_copy_literal[id=@literal:3] -> float_type, {784, 128}, {128, 1}
@3 = load[offset=0,end=512](@1) -> float_type, {1, 128}, {128, 1}
x = @param:x -> float_type, {1, 28, 28}, {784, 28, 1}
@4 = reshape[dims={-1, 784}](x) -> float_type, {1, 784}, {784, 1}
@5 = gpu::gemm[alpha=1,beta=0](@4,@2,@3) -> float_type, {1, 128}, {128, 1}
@6 = hip::hip_copy_literal[id=@literal:1] -> float_type, {128, 10}, {10, 1}
@7 = hip::hip_copy_literal[id=@literal:0] -> float_type, {10}, {1}
@8 = hip::hip_copy_literal[id=@literal:2] -> float_type, {128}, {1}
@9 = broadcast[axis=1,dims={1, 128}](@8) -> float_type, {1, 128}, {0, 1}
@10 = load[offset=512,end=1024](@1) -> float_type, {1, 128}, {128, 1}
@11 = gpu::add_relu(@5,@9,@10) -> float_type, {1, 128}, {128, 1}
@12 = load[offset=0,end=40](@1) -> float_type, {1, 10}, {10, 1}
@13 = gpu::gemm[alpha=1,beta=0](@11,@6,@12) -> float_type, {1, 10}, {10, 1}
@14 = broadcast[axis=1,dims={1, 10}](@7) -> float_type, {1, 10}, {0, 1}
@15 = load[offset=40,end=80](@1) -> float_type, {1, 10}, {10, 1}
@16 = gpu::add(@13,@14,@15) -> float_type, {1, 10}, {10, 1}
output = @param:output -> float_type, {1, 10}, {10, 1}
@17 = gpu::softmax[axis=1](@16,output) -> float_type, {1, 10}, {10, 1}

Allocating params ...
Running performance report ...
@0 = check_context::migraphx::gpu::context -> float_type, {}, {}: 0.00057782ms, 1%
@1 = hip::hip_allocate_memory[shape=float_type, {256}, {1},id=scratch] -> float_type,
↪{256}, {1}: 0.000295ms, 1%
@2 = hip::hip_copy_literal[id=@literal:3] -> float_type, {784, 128}, {128, 1}: 0.
↪00027942ms, 1%
@3 = load[offset=0,end=512](@1) -> float_type, {1, 128}, {128, 1}: 0.000232ms, 1%
x = @param:x -> float_type, {1, 28, 28}, {784, 28, 1}: 0.0003206ms, 1%
@4 = reshape[dims={-1, 784}](x) -> float_type, {1, 784}, {784, 1}: 0.00033842ms, 1%
@5 = gpu::gemm[alpha=1,beta=0](@4,@2,@3) -> float_type, {1, 128}, {128, 1}: 0.212592ms,␣
↪52%
@6 = hip::hip_copy_literal[id=@literal:1] -> float_type, {128, 10}, {10, 1}: 0.
↪00085822ms, 1%
@7 = hip::hip_copy_literal[id=@literal:0] -> float_type, {10}, {1}: 0.000382ms, 1%
@8 = hip::hip_copy_literal[id=@literal:2] -> float_type, {128}, {1}: 0.0003486ms, 1%
@9 = broadcast[axis=1,dims={1, 128}](@8) -> float_type, {1, 128}, {0, 1}: 0.000299ms, 1%
@10 = load[offset=512,end=1024](@1) -> float_type, {1, 128}, {128, 1}: 0.000234ms, 1%
```

(continues on next page)

(continued from previous page)

```
@11 = gpu::add_relu(@5,@9,@10) -> float_type, {1, 128}, {128, 1}: 0.0416597ms, 11%
@12 = load[offset=0,end=40](@1) -> float_type, {1, 10}, {10, 1}: 0.0007548ms, 1%
@13 = gpu::gemm[alpha=1,beta=0](@11,@6,@12) -> float_type, {1, 10}, {10, 1}: 0.0733071ms,
↪ 18%
@14 = broadcast[axis=1,dims={1, 10}](@7) -> float_type, {1, 10}, {0, 1}: 0.00088142ms, 1%
@15 = load[offset=40,end=80](@1) -> float_type, {1, 10}, {10, 1}: 0.000408ms, 1%
@16 = gpu::add(@13,@14,@15) -> float_type, {1, 10}, {10, 1}: 0.0410144ms, 10%
output = @param:output -> float_type, {1, 10}, {10, 1}: 0.0010222ms, 1%
@17 = gpu::softmax[axis=1](@16,output) -> float_type, {1, 10}, {10, 1}: 0.0385636ms, 10%

Summary:
gpu::gemm: 0.285899ms, 69%
gpu::add_relu: 0.0416597ms, 11%
gpu::add: 0.0410144ms, 10%
gpu::softmax: 0.0385636ms, 10%
hip::hip_copy_literal: 0.00186824ms, 1%
load: 0.0016288ms, 1%
@param: 0.0013428ms, 1%
broadcast: 0.00118042ms, 1%
check_context::migraphx::gpu::context: 0.00057782ms, 1%
reshape: 0.00033842ms, 1%
hip::hip_allocate_memory: 0.000295ms, 1%

Rate: 2866.1/sec
Total time: 0.348906ms
Total instructions time: 0.414369ms
Overhead time: 0.00348144ms, -0.0654627ms
Overhead: 1%, -19%
```

4.4 Driver options

This document lists the MIGraphX driver commands along with the eligible options.

4.4.1 read

Loads and prints input graph.

4.4.2 compile

Compiles and prints input graph.

4.4.3 run

Loads and prints input graph.

4.4.4 perf

Compiles and runs input graph then prints performance report.

--iterations, -n [unsigned int]

Sets number of iterations to run for perf report (Default: 100)

4.4.5 verify

Runs reference and CPU or GPU implementations and checks outputs for consistency.

--rms-tol [double]

Sets tolerance for RMS error (Default: 0.001)

--atol [double]

Sets tolerance for elementwise absolute difference (Default: 0.001)

--rtol [double]

Sets tolerance for elementwise relative difference (Default: 0.001)

-i, --per-instruction

Verifies each instruction

-r, --reduce

Reduces program and verifies

--ref-use-double

Converts floating point values to double for the ref target

4.4.6 roctx

`roctx` provides marker information for each operation which allows MIGraphX to be used with rocprof for performance analysis. This allows you to get GPU-level kernel timing information. Here is how you can use `roctx` combined with rocprof for tracing:

```
/opt/rocm/bin/rocprof --hip-trace --roctx-trace --flush-rate 1ms --timestamp on -d
↪<OUTPUT_PATH> --obj-tracking on /opt/rocm/bin/migraphx-driver roctx <ONNX_FILE>
↪<MIGRAPHX_OPTIONS>
```

Running rocprof generates trace information for HIP, HCC and ROCTX in separate `.txt` files. To understand the interactions between API calls, utilize the *roctx.py* helper script.

CHAPTER

FIVE

CONTRIBUTING TO MIGRAPHX

This document explains the internal implementation of some commonly used MIGraphX APIs. You can utilize the information provided in this document and other documents under "Contributing to MIGraphX" section to contribute to the MIGraphX API implementation. Here is how some basic operations in the MIGraphX framework are performed.

5.1 Performing basic operations

A program is a collection of modules, which are collections of instructions to be executed when calling `eval`. Each instruction has an associated `operation` which represents the computation to be performed by the instruction.

The following code snippets demonstrate some basic operations using MIGraphX.

5.1.1 Adding literals

Here is a `add_two_literals()` function:

```
// create the program and get a pointer to the main module
migraphx::program p;
auto* mm = p.get_main_module();

// add two literals to the program
auto one = mm->add_literal(1);
auto two = mm->add_literal(2);

// make the add operation between the two literals and add it to the program
mm->add_instruction(migraphx::make_op("add"), one, two);

// compile the program on the reference device
p.compile(migraphx::ref::target{});

// evaulate the program and retreive the result
auto result = p.eval({}).back();
std::cout << "add_two_literals: 1 + 2 = " << result << "\n";
```

In the above function, a simple `program` object is created along with a pointer to the main module of it. The program is a collection of modules which starts execution from the main module, so instructions are added to the modules rather than the program object directly. The *add_literal* function is used to add an instruction that stores the literal number 1 while returning an `instruction_ref`. The returned `instruction_ref` can be used in another instruction as an input. The same *add_literal* function is used to add the literal 2 to the program. After the literals are created, the instruction is created to add the numbers. This is done by using the *add_instruction* function with the add `operation` created by `make_op` and the previously created literals passed as the arguments for the instruction. You can run this *program* by compiling it for the reference target (CPU) and then running it with *eval*. This prints the result on the console.

To compile the program for the GPU, move the file to `test/gpu/` directory and include the given target:

```
#include <migraphx/gpu/target.hpp>
```

5.1.2 Adding Parameters

While the `add_two_literals()` function above demonstrates add operation on constant values `1` and `2`, the following program demonstrates how to pass a parameter (`x`) to a module using `add_parameter()` function .

> migraphx::program p; auto* mm = p.get_main_module(); migraphx::shape s{migraphx::shape::int32_type, {1}};
>
> // add parameter "x" with the shape s auto x = mm->add_parameter("x", s); auto two = mm->add_literal(2);
>
> // add the "add" instruction between the "x" parameter and "two" to the module mm->add_instruction(migraphx::make_op("add"), x, two); p.compile(migraphx::ref::target{});

In the code snippet above, an add operation is performed on a parameter of type `int32` and literal `2` followed by compilation for the CPU. To run the program, pass the parameter as a `parameter_map` while calling *eval*. To map the parameter `x` to an *argument* object with an `int` data type, a `parameter_map` is created as shown below:

```
// create a parameter_map object for passing a value to the "x" parameter
std::vector<int> data = {4};
migraphx::parameter_map params;
params["x"] = migraphx::argument(s, data.data());

auto result = p.eval(params).back();
std::cout << "add_parameters: 4 + 2 = " << result << "\n";
EXPECT(result.at<int>() == 6);
```

5.1.3 Handling Tensor Data

The above two examples demonstrate scalar operations. To describe multi-dimensional tensors, use the *shape* class to compute a simple convolution as shown below:

```
migraphx::program p;
auto* mm = p.get_main_module();

// create shape objects for the input tensor and weights
migraphx::shape input_shape{migraphx::shape::float_type, {2, 3, 4, 4}};
migraphx::shape weights_shape{migraphx::shape::float_type, {3, 3, 3, 3}};

// create the parameters and add the "convolution" operation to the module
auto input   = mm->add_parameter("X", input_shape);
auto weights = mm->add_parameter("W", weights_shape);
mm->add_instruction(migraphx::make_op("convolution", {{"padding", {1, 1}}, {"stride", {2,
↪ 2}}}), input, weights);
```

Most programs take data from allocated buffers that are usually on the GPU. To pass the buffer data as an argument, create *argument* objects directly from the pointers to the buffers:

```
// Compile the program
p.compile(migraphx::ref::target{});

// Allocated buffers by the user
std::vector<float> a = ...;
std::vector<float> c = ...;

// Solution vector
std::vector<float> sol = ...;

// Create the arguments in a parameter_map
migraphx::parameter_map params;
params["X"] = migraphx::argument(input_shape, a.data());
params["W"] = migraphx::argument(weights_shape, c.data());

// Evaluate and confirm the result
auto result = p.eval(params).back();
std::vector<float> results_vector(64);
result.visit([&](auto output) { results_vector.assign(output.begin(), output.end()); });

EXPECT(migraphx::verify::verify_rms_range(results_vector, sol));
```

An *argument* can handle memory buffers from either the GPU or the CPU. When running the *program*, buffers are allocated on the corresponding target by default. By default, the buffers are allocated on the CPU when compiling for CPU and on the GPU when compiling for GPU. To locate the buffers on the CPU even when compiling for GPU, set the option `offload_copy=true`.

5.1.4 Importing From ONNX

To make it convenient to use neural networks directly from other frameworks, MIGraphX ONNX parser allows you to build a *program* directly from an ONNX file. For usage, refer to the `parse_onnx()` function below:

```
program p = migraphx::parse_onnx("model.onnx");
p.compile(migraphx::gpu::target{});
```

5.2 Sample programs

You can find all the MIGraphX examples in the Examples directory.

5.2.1 Build MIGraphX source code

To build a sample program ref_dev_examples.cpp, use:

make -j$(nproc) test_ref_dev_examples

This creates an executable file `test_ref_dev_examples` in the `bin/` of the build directory.

To verify the build, use:

make -j$(nproc) check

For detailed instructions on building MIGraphX from source, refer to the README file.

5.3 Data types

5.3.1 shape

struct **shape**

Public Types

enum **type_t**

Values:

enumerator **bool_type**

enumerator **half_type**

enumerator **float_type**

enumerator **double_type**

enumerator **uint8_type**

enumerator **int8_type**

enumerator **uint16_type**

enumerator **int16_type**

enumerator **int32_type**

enumerator **int64_type**

enumerator **uint32_type**

enumerator **uint64_type**

enumerator **fp8e4m3fnuz_type**

enumerator **tuple_type**

Public Functions

shape()

shape(*type_t* t)

shape(*type_t* t, std::vector<std::size_t> l)

shape(*type_t* t, std::vector<std::size_t> l, std::vector<std::size_t> s)

shape(*type_t* t, std::initializer_list<std::size_t> d)

shape(*type_t* t, std::vector<*dynamic_dimension*> dims)

shape(*type_t* t, std::vector<std::size_t> mins, std::vector<std::size_t> maxes, std::vector<std::set<std::size_t>> optimals_list)

template<class **Range**>
inline **shape**(*type_t* t, const *Range* &l)

template<class **Range1**, class **Range2**>
inline **shape**(*type_t* t, const *Range1* &l, const *Range2* &s)

shape(const std::vector<*shape*> &subs)

type_t **type**() const

const std::vector<std::size_t> &**lens**() const

const std::vector<std::size_t> &**strides**() const

std::size_t **ndim**() const

The number of dimensions in the shape, either static or dynamic. Same as the number of indices required to get a data value.

std::size_t **elements**() const

Return the number of elements in the tensor.

std::size_t **bytes**() const

Return the number of total bytes used for storage of the tensor data; includes subshapes. For dynamic shape, returns the maximum number of bytes presuming a packed shape.

std::size_t **type_size**() const

Return the size of the type of the main shape. Returns 0 if there are subshapes.

const std::vector<*dynamic_dimension*> &**dyn_dims**() const

std::vector<std::size_t> **min_lens**() const

Minimum lengths for dynamic shape. *lens()* for static shape.

std::vector<std::size_t> **max_lens**() const

Maximum lengths for dynamic shape. *lens()* for static shape.

std::vector<std::set<std::size_t>> **opt_lens**() const

Optimum lengths for dynamic shape. Empty for static shape.

std::size_t **index**(std::initializer_list<std::size_t> l) const

Map multiple indices to space index.

std::size_t **index**(const std::vector<std::size_t> &l) const

Map multiple indices to space index.

template<class **Iterator**>
inline std::size_t **index**(*Iterator* start, *Iterator* last) const

Map multiple indices from a range of iterator to a space index.

std::size_t **index**(std::size_t i) const

Map element index to space index.

std::vector<std::size_t> **multi**(std::size_t idx) const

Map element index to multi-dimensional index.

void **multi_copy**(std::size_t idx, std::size_t *start, const std::size_t *end) const

Map element index to multi-dimensional index and put them them into location provided by pointers

bool **packed**() const

Returns true if the shape is packed (number of elements and buffer size the same) with no padding

bool **transposed**() const

Returns true if the shape has been transposed. That is the strides are not in descending order

bool **broadcasted**() const

Returns true if the shape is broadcasting a dimension. That is, one of the strides are zero.

bool **standard**() const

Returns true if the shape is in its standard format. That is, the shape is both packed and not transposed.

bool **scalar**() const

Returns true if all strides are equal to 0 (scalar tensor)

bool **dynamic**() const

Return true if the shape is dynamic.

bool **any_of_dynamic**() const

Return true if this shape or any of the sub_shapes are dynamic.

shape **normalize_standard**() const

shape **as_standard**() const

shape **with_lens**(*type_t* t, const std::vector<std::size_t> &l) const

shape **with_lens**(const std::vector<std::size_t> &l) const

shape **with_type**(*type_t* t) const

shape **to_dynamic**() const

shape **to_static**(std::size_t x) const

template<class ...**Visitors**>
inline void **visit_type**(*Visitors*... vs) const

std::string **type_string**() const

const std::vector<*shape*> &**sub_shapes**() const std::size_t **element_space**() const

Returns the number of elements in the data buffer. For a dynamic shape, returns the maximum number of elements of the data buffer and assumes it is packed. Will clip to the maximum of size_t if overflows for dynamic shapes.

Public Static Functions

static const std::vector<*type_t*> &**types**()

static std::string **name**(*type_t* t)

static std::string **cpp_type**(*type_t* t)

static *shape* **from_permutation**(*type_t* t, const std::vector<std::size_t> &l, const std::vector<int64_t> &perm)

Creates an output shape with dimensions equal to the input lengths and strides determined by the permutation argument such that find_permutation() of the output shape returns the inputted permuation.

2D example: parameters: l = [2, 3], perm = [1, 0] therefore: "original" shape = {lens = [3, 2], strides = [2, 1]} output_shape = {lens = [2, 3], strides = [1, 2]

3D example: parameters: l = [2, 3, 4], perm = [1, 2, 0] therefore: "original" shape = {lens = [3, 4, 2], strides = [8, 2, 1]} output_shape = {lens = [2, 3, 4], strides = [1, 8, 2]}

template<class **Visitor**, class **TupleVisitor**>
static inline void **visit**(*type_t* t, *Visitor* v, *TupleVisitor* tv)

template<class **Visitor**>
static inline void **visit**(*type_t* t, *Visitor* v)

template<class **Visitor**>
static inline void **visit_types**(*Visitor* v)

static *type_t* **parse_type**(const std::string &s)

Friends

friend bool **operator==**(const *shape* &x, const *shape* &y)

friend bool **operator!=**(const *shape* &x, const *shape* &y)

friend std::ostream &**operator<<**(std::ostream &os, const *shape* &x)

template<class **T**>

struct **as**

Public Types

using **type** = std::conditional_t<std::is_same<*T*, bool>{}, int8_t, *T*>

Public Functions

inline *type* **max**() const

inline *type* **min**() const

inline *type* **nan**() const

template<class **U**>
inline *type* **operator()**(*U* u) const

template<class **U**>
inline *type* ***operator()**(*U* *u) const

template<class **U**>
inline const *type* ***operator()**(const *U* *u) const

inline *type* **operator()**() const

inline std::size_t **size**(std::size_t n = 1) const

inline auto **is_integral**() const

inline auto **is_signed**() const

inline auto **is_unsigned**() const

template<class **U**>
inline *type* ***from**(*U* *buffer, std::size_t n = 0) const

template<class **U**>
inline const *type* ***from**(const *U* *buffer, std::size_t n = 0) const

inline *type_t* **type_enum**() const

struct **dynamic_dimension**

Public Functions

bool **is_fixed**() const

bool **has_optimal**() const

dynamic_dimension &**operator+=**(const std::size_t &x)

dynamic_dimension &**operator-=**(const std::size_t &x)

Public Members

std::size_t **min** = 0

std::size_t **max** = 0

std::set<std::size_t> **optimals** = { }

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

Friends

friend bool **operator==**(const *dynamic_dimension* &x, const *dynamic_dimension* &y)

friend bool **operator!=**(const *dynamic_dimension* &x, const *dynamic_dimension* &y)

friend std::ostream &**operator<<**(std::ostream &os, const *dynamic_dimension* &x)

friend bool **operator==**(const *dynamic_dimension* &x, const std::size_t &y)

friend bool **operator==**(const std::size_t &x, const *dynamic_dimension* &y)

friend bool **operator!=**(const *dynamic_dimension* &x, const std::size_t &y)

friend bool **operator!=**(const std::size_t &x, const *dynamic_dimension* &y)

friend *dynamic_dimension* **operator+**(const *dynamic_dimension* &x, const std::size_t &y)

friend *dynamic_dimension* **operator+**(const std::size_t &x, const *dynamic_dimension* &y)

friend *dynamic_dimension* **operator-**(const *dynamic_dimension* &x, const std::size_t &y)

template<class **T**, class = void>

struct **get_type**

Subclassed by *migraphx::internal::shape::get_type< const T >*

template<class **T**>

struct **get_type**<bool, *T*> : public std::integral_constant<*type_t*, bool_type>

template<class **T**>

struct **get_type**<const *T*> : public migraphx::internal::*shape*::*get_type*<*T*>

template<class **T**>

struct **get_type**<double, *T*> : public std::integral_constant<*type_t*, double_type>

template<class **T**>

struct **get_type**<float, *T*> : public std::integral_constant<*type_t*, float_type>

template<class **T**>

struct **get_type**<half, *T*> : public std::integral_constant<*type_t*, half_type>

template<class **T**>

struct **get_type**<int16_t, *T*> : public std::integral_constant<*type_t*, int16_type>

template<class **T**>

struct **get_type**<int32_t, *T*> : public std::integral_constant<*type_t*, int32_type>

template<class **T**>

struct **get_type**<int64_t, *T*> : public std::integral_constant<*type_t*, int64_type>

template<class **T**>

struct **get_type**<int8_t, *T*> : public std::integral_constant<*type_t*, int8_type>

template<class T> fp8e4m3fnuz, T > : public std::integral_constant< type_t, fp8e4m3fnuz_type >

template<class **T**>

struct **get_type**<uint16_t, *T*> : public std::integral_constant<*type_t*, uint16_type>

template<class **T**>

struct **get_type**<uint32_t, *T*> : public std::integral_constant<*type_t*, uint32_type>

template<class **T**>

struct **get_type**<uint64_t, *T*> : public std::integral_constant<*type_t*, uint64_type>

template<class **T**>

struct **get_type**<uint8_t, *T*> : public std::integral_constant<*type_t*, uint8_type>

5.3.2 literal

struct **literal** : public migraphx::internal::raw_data<*literal*>

Represents a raw literal.

This stores the literal has a raw buffer that is owned by this class

Public Functions

inline **literal**()

inline explicit **literal**(*shape*::type_t shape_type)

Empty literal with a specific shape type

template<class **U**, class **T** = deduce<*U*>, *shape*::type_t **ShapeType** = *shape*::get_type<*T*>{}>
inline **literal**(*U* x)

template<class **T**>
inline **literal**(const *shape* &s, const std::vector<*T*> &x)

template<class **T**>
inline **literal**(const *shape* &s, const std::initializer_list<*T*> &x)

template<class **Iterator**>
inline **literal**(const *shape* &s, *Iterator* start, *Iterator* end)

template<class **T**, long **PrivateRequires__LINE__** = __LINE__, typename std::enable_if<(*PrivateRequires__LINE__* == __LINE__ && (migraphx::and_<sizeof(*T*) == 1>{})), int>::type = 0>
inline **literal**(const *shape* &s, *T* *x)

inline bool **empty**() const

Whether data is available.

inline const char ***data**() const

Provides a raw pointer to the data.

inline const *shape* &**get_shape**() const

inline std::vector<*literal*> **get_sub_objects**() const

inline *argument* **get_argument**() const

Convert the data to an argument.

5.3.3 argument

struct **argument** : public migraphx::internal::raw_data<*argument*>

Arguments passed to instructions.

An `argument` can represent a raw buffer of data that either be referenced from another element or it can be owned by the argument.

Public Functions

argument() = default

explicit **argument**(const *shape* &s)

template<class **F**, long **PrivateRequires__LINE__** = __LINE__, typename std::enable_if<(*PrivateRequires__LINE__* == __LINE__ && (migraphx::and_<std::is_pointer<decltype(std::declval<*F*>()())>{}>{})), int>::type = 0>

inline **argument**(*shape* s, *F* d)

template<class **T**>
inline **argument**(*shape* s, *T* *d)

template<class **T**>
inline **argument**(*shape* s, std::shared_ptr<*T*> d)

argument(*shape* s, std::nullptr_t)

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

char ***data**() const

Provides a raw pointer to the data.

bool **empty**() const

Whether data is available.

const *shape* &**get_shape**() const

argument **reshape**(const *shape* &s) const

argument **copy**() const

argument **share**() const

Make copy of the argument that is always sharing the data.

std::vector<*argument*> **get_sub_objects**() const

argument **element**(std::size_t i) const

Return the ith element.

template<class **Iterator**>
inline void **fill**(*Iterator* start, *Iterator* end)

5.3.4 raw_data

template<class **Derived**>

struct **raw_data** : public migraphx::internal::raw_data_base

Provides a base class for common operations with raw buffer.

For classes that handle a raw buffer of data, this will provide common operations such as equals, printing, and visitors. To use this class the derived class needs to provide a `data()` method to retrieve a raw pointer to the data, and `get_shape` method that provides the shape of the data.

Public Functions

template<class **Visitor**>
inline void **visit_at**(*Visitor* v, std::size_t n = 0) const

Visits a single data element at a certain index.

Parameters

- **v** – A function which will be called with the type of data
- **n** – The index to read from

template<class **Visitor**, class **TupleVisitor**>
inline void **visit**(*Visitor* v, *TupleVisitor* tv) const

template<class **Visitor**>
inline void **visit**(*Visitor* v) const

Visits the data.

This will call the visitor function with a `tensor_view`<T> based on the shape of the data.

Parameters
v – A function to be called with `tensor_view`<T>

inline bool **single**() const

Returns true if the raw data is only one element.

template<class **T**>
inline *T* **at**(std::size_t n = 0) const

Retrieves a single element of data.

Parameters
n – The index to retrieve the data from

Template Parameters
T – The type of data to be retrieved

Returns
The element as `T`

inline *auto_cast* **implicit**() const

Implicit conversion of raw data pointer.

template<class **T**>
inline tensor_view<*T*> **get**() const

Get a *tensor_view* to the data.

template<class **T**>
inline *T* ***cast**() const

Cast the data pointer.

inline std::string **to_string**() const

Friends

template<class **Stream**>
inline friend *Stream* &**operator<<**(*Stream* &os, const *Derived* &d)

struct **auto_cast**

Public Types

template<class **T**>

using **is_data_ptr** = bool_c<(std::is_void<*T*>{} or std::is_same<char, std::remove_cv_t<*T*>>{} or std::is_same<unsigned char, std::remove_cv_t<*T*>>{})>

template<class **T**>

using **get_data_type** = std::conditional_t<*is_data_ptr*<*T*>{}, float, *T*>

Public Functions

template<class **T**>
inline **operator** *T*()

template<class **T**>
inline bool **matches**() const

template<class **T**>
inline **operator** *T**()

Public Members

const *Derived* ***self**

template<class **T**, class ...**Ts**>
auto `migraphx::internal::`**`visit_all`**(*T* &&x, *Ts*&&... xs)

Visits every object together.

This will visit every object, but assumes each object is the same type. This can reduce the deeply nested visit calls. Returns a function that takes the visitor callback. Calling syntax is `visit_all(xs...)([](auto... ys) {})` where `xs...` and `ys...` are the same number of parameters.

Parameters

- **x** – A raw data object
- **xs** – Many raw data objects.

Returns

A function to be called with the visitor

template<class **T**>
auto `migraphx::internal::`**`visit_all`**(const std::vector<*T*> &x)

Visits every object together.

This will visit every object, but assumes each object is the same type. This can reduce the deeply nested visit calls. Returns a function that takes the visitor callback.

Parameters

x – A vector of raw data objects. Types must all be the same.

Returns

A function to be called with the visitor

5.3.5 tensor_view

template<class **T**>

struct **tensor_view**

Public Types

using **value_type** = *T*

using **iterator** = basic_iota_iterator<tensor_view_iterator_read<*tensor_view*<*T*>>, std::size_t>

using **const_iterator** = basic_iota_iterator<tensor_view_iterator_read<const *tensor_view*<*T*>>, std::size_t>

Public Functions

inline **tensor_view**()

inline **tensor_view**(*shape* s, *T* *d)

inline const *shape* &**get_shape**() const

inline bool **empty**() const

inline std::size_t **size**() const

inline *T* ***data**()

inline const *T* ***data**() const

template<class ...**Ts**, long **PrivateRequires__LINE__** = __LINE__, typename std::enable_if<(*PrivateRequires__LINE__* == __LINE__ && (migraphx::and_<std::is_integral<*Ts*>{}...>{})), int>::type = 0>
inline const *T* &**operator()**(*Ts*... xs) const

template<class ...**Ts**, long **PrivateRequires__LINE__** = __LINE__, typename std::enable_if<(*PrivateRequires__LINE__* == __LINE__ && (migraphx::and_<std::is_integral<*Ts*>{}...>{})), int>::type = 0>
inline *T* &**operator()**(*Ts*... xs)

template<class **Iterator**, long **PrivateRequires__LINE__** = __LINE__, typename std::enable_if<(*PrivateRequires__LINE__* == __LINE__ && (migraphx::and_<not std::is_integral<*Iterator*>{}>{})), int>::type = 0>
inline const *T* &**operator()**(*Iterator* start, *Iterator* last) const

template<class **Iterator**, long **PrivateRequires__LINE__** = __LINE__, typename std::enable_if<(*PrivateRequires__LINE__* == __LINE__ && (migraphx::and_<not std::is_integral<*Iterator*>{}>{})), int>::type = 0>
inline *T* &**operator()**(*Iterator* start, *Iterator* last)

inline *T* &**operator[]**(std::size_t i) inline const *T* &**operator[]**(std::size_t i) const

inline *T* &**front**()

inline const *T* &**front**() const

inline *T* &**back**()

inline const *T* &**back**() const

inline *iterator* **begin**()

inline *iterator* **end**()

inline *const_iterator* **begin**() const

inline *const_iterator* **end**() const

template<class **U** = *T*>
inline std::vector<*U*> **to_vector**() const

Friends

inline friend std::ostream &**operator<<**(std::ostream &os, const *tensor_view*<*T*> &x)

5.4 Operators

5.4.1 operation

struct **operation**

The operation interface represents an action an instruction will perform. All operation classes must be Copy-Constructible.

Public Functions

std::string **name**() const

A unique name identifying the operation.

void **finalize**(context &ctx)

An optional method that can be used to finalize the operator before running.

shape **compute_shape**(const std::vector<*shape*> &input) const

This is used to compute the resulting shape from an operation. If an operation cannot be run with input shapes, then it should throw an exception.

argument **compute**(context &ctx, const *shape* &output, const std::vector<*argument*> &input) const

This performs the operation's computation.

This method can be optional when the operation is only used as a placeholder to be lowered later on.

Parameters

- **ctx** – This is the context created by the `target` during compilation. Implementations can use the target's `context` class rather than the `context` interface class.

- **output** – Equivalent to running compute_shape with each shape of the argument. For a fixed shape, the returned argument will have the same shape as output. For a dynamic shape, the returned argument will be a fixed shape within the bounds set in the dynamic shape output.
- **input** – This is the argument result from the previous instruction's computation.

Returns
Return an argument of the result computation. The shape of argument should be the same the output shape.

std::ptrdiff_t **output_alias**(const std::vector<*shape*> &input) const

An optional method to return which argument the output will alias. If there is no aliased output then -1 can be returned.

Friends

friend std::ostream &**operator<<**(std::ostream &os, const *operation* &op)

An optional stream operator to print the operation. When this is not implemented, it will just print the operation's name.

bool migraphx::internal::**is_context_free**(const *operation* &x)

Returns true if operation does not require a context to run compute.

bool migraphx::internal::**has_finalize**(const *operation* &x)

Returns true if the operation has a finalize method.

5.4.2 operators

namespace **op**

Enums

enum **padding_mode_t**

Values:

enumerator **default_**

enumerator **same_lower**

enumerator **same_upper**

enum class **pooling_mode**

Values:

enumerator **average**

enumerator **max**

enumerator **lpnorm**

enum class **rnn_direction**

Values:

enumerator **forward**

enumerator **reverse**

enumerator **bidirectional**

enum class **normalize_attribute**

`normalize_attribute` settings: Note that default options are not included as enums.

i. `use_input` (default) vs. `use_output`: Affects the rank of the attribute. `use_input -> lens.size()`, `use_output -> lens.size() + vec.size()`.
ii. use_rank (default) vs use_len: `use_rank` sets the max value/index of the attribute as the rank of lens. `use_lens` sets the max value/index as the corresponding value in lens at the axes index. Uses the dynamic_dimension.max value for dynamic shapes. Returns the original vector (no normalization) if any of dynamic_dimension[axes] are not fixed.
iii. `clip_min` vs. `not_clip_min` (default): Clip values less than the minimum to the minimum or not.
iv. `include_min` vs. `exclude_min` (default): Include or exclude the minimum value/index for range checking and clipping.
v. `clip_max` vs. `not_clip_max` (default): Clip values greater than the maximum or not.
vi. `include_max` vs. `exclude_max` (default): Include or exclude the maximum value/index for range checking and clipping.
vii. `normalize_padding`: To normalize the padding to `2*(pad ndim)` dimensions.

Values:

enumerator **use_output**

enumerator **use_len**

enumerator **clip_max**

enumerator **clip_min**

enumerator **include_max**

enumerator **include_min**

enumerator **normalize_padding**

Functions

std::ostream &**operator<<**(std::ostream &os, *pooling_mode* v)

std::ostream &**operator<<**(std::ostream &os, *rnn_direction* v)

struct **abs** : public migraphx::internal::op::unary<*abs*>

#include <migraphx/op/abs.hpp>

Public Functions

inline auto **apply**() const

struct **acos** : public migraphx::internal::op::unary<*acos*>

#include <migraphx/op/acos.hpp>

Public Functions

inline auto **apply**() const

struct **acosh** : public migraphx::internal::op::unary<*acosh*>

#include <migraphx/op/acosh.hpp>

Public Functions

inline auto **apply**() const

struct **add** : public migraphx::internal::op::binary<*add*>

#include <migraphx/op/add.hpp>

Public Functions

inline value **attributes**() const

inline std::string **point_function**() const

inline auto **apply**() const

struct **allocate**

#include <migraphx/op/allocate.hpp> Static allocate: No inputs: `allocate()` `this.s` attribute set to the static output shape of the buffer. `this.s` attribute can be set to a dynamic output shape; however this will allocate the maximum buffer size for that case

Dynamic allocate: One input: `allocate(output_dims)` `output_dims` are the output buffer dimensions and has a static shape. Either `this.s` or `this.buf_type` (but not both) must be set to calculate the dynamic output shape at compute time. If `this.buf_type` is set, the *compute_shape()* of allocate at compile time will have dynamic_dimensions from {0, max_int} with rank = output_dims.ndim(). If `this.s` is set then the *compute_shape()* will output `this.s`; `this.s` should be a dynamic shape.

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs) const

inline *argument* **compute**(const *shape* &output_shape, const std::vector<*argument*> &args) const

Public Members

optional<*shape*> **s**

optional<*shape*>::type_t> **buf_type**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **argmax**

#include <migraphx/op/argmax.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

template<class **T**>
inline int64_t **calc_argmax**(*T* &input, std::vector<std::size_t> &indices, size_t item_num) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

Public Members

int64_t **axis** = 0

bool **select_last_index** = false

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **argmin**

#include <migraphx/op/argmin.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

template<class **T**>
inline int64_t **calc_argmin**(*T* &input, std::vector<std::size_t> &indices, size_t item_num) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

Public Members

int64_t **axis** = 0

bool **select_last_index** = false

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **as_shape**

#include <migraphx/op/as_shape.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs) const

inline *argument* **compute**(*shape* output_shape, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

shape **s**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **asin** : public migraphx::internal::op::unary<*asin*>

#include <migraphx/op/asin.hpp>

Public Functions

inline auto **apply**() const

struct **asinh** : public migraphx::internal::op::unary<*asinh*>

#include <migraphx/op/asinh.hpp>

Public Functions

inline auto **apply**() const

struct **atan** : public migraphx::internal::op::unary<*atan*>

#include <migraphx/op/atan.hpp>

Public Functions

inline auto **apply**() const

struct **atanh** : public migraphx::internal::op::unary<*atanh*>

#include <migraphx/op/atanh.hpp>

Public Functions

inline auto **apply**() const

template<class **Derived**>

struct **binary** : public migraphx::internal::op::op_name<*Derived*>

#include <migraphx/op/binary.hpp>

Public Functions

inline std::string **point_function**() const

inline std::string **point_op**() const

inline value **base_attributes**() const

inline value **attributes**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

struct **broadcast**

#include <migraphx/op/broadcast.hpp> 1 input version: Broadcasts a tensor from the original shape to the broadcast_lens by setting the stride of broadcasted dimensions to zero. `axis` attribute for a 1D input shape is the output dimension that stays the same. ex: broadcasting shape [1024] -> [4, 1024, 3] has axis = 1.

For higher rank input shapes, axis is an offset parameter for the broadcasting. Such that this operator would work in the opposite direction of NumPy broadcasting (left-most to rightwards element-wise comparison) ex: broadcasting shape [2, 2] -> [2, 2, 3] with axis = 0

2 input version: Broadcast the first input 1D shape into the second input shape based on the axis parameter. Handles broadcasting a 1D static shape into a higher rank dynamic shape. broadcast_lens is not used

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

uint64_t **axis** = 0

std::vector<std::size_t> **broadcast_lens** = {}

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **capture**

#include <migraphx/op/capture.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(context&, const *shape*&, const std::vector<*argument*> &args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::size_t **ins_index**

std::function<void(std::size_t ins_index, std::vector<*argument*>)> **f** = {}

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **ceil** : public migraphx::internal::op::unary<*ceil*>

#include <migraphx/op/ceil.hpp>

Public Functions

inline auto **apply**() const

struct **clip**

#include <migraphx/op/clip.hpp>

Public Functions

inline std::string **name**() const

inline value **attributes**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

struct **concat**

#include <migraphx/op/concat.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline std::vector<std::size_t> **compute_offsets**(const *shape* &output_shape, const std::vector<*argument*> &args) const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

Public Members

int64_t **axis** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **contiguous**

#include <migraphx/op/contiguous.hpp> The contiguous operator takes a non-standard input tensor and returns the same tensor but in standard form. For example, if input tensor A which has lens = (4,5) is first transposed, i.e. lens = (5,4), this tensor's data layout remained the same during the transpose operation; only it's shape lengths and strides were changed. This leaves the tensor in a non-standard form. The contiguous operator copies the underlying data such that resulting tensor is returned to a standard form.

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline auto **apply**() const

struct **convert** : public migraphx::internal::op::unary<*convert*>

#include <migraphx/op/convert.hpp>

Public Functions

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline std::string **point_op**() const

inline auto **apply**() const

inline **convert**(*shape*::type_t t)

inline **convert**()

Public Members

shape::type_t **target_type** = *shape*::half_type

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **convolution**

#include <migraphx/op/convolution.hpp> Convolution operator. Does not support optimal dimensions for spatial dimensions. Returns empty optimals.

Public Functions

inline std::string **name**() const

inline void **check_attribute_size**() const

inline value **attributes**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline std::vector<std::size_t> **calc_conv_lens**(std::vector<std::size_t> x_lens, std::vector<std::size_t> w_lens) const

inline *shape* **dynamic_compute_shape**(*shape* x_shape, *shape* w_shape) const

inline *shape* **static_compute_shape**(*shape* x_shape, *shape* w_shape) const

inline size_t **kdims**() const

inline *argument* **compute**(*shape* output_shape, std::vector<*argument*> args) const

Public Members

std::vector<std::size_t> **padding** = {0, 0}

std::vector<std::size_t> **stride** = {1, 1}

std::vector<std::size_t> **dilation** = {1, 1}

int **group** = 1

padding_mode_t **padding_mode** = default_

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **convolution_backwards**

#include <migraphx/op/convolution_backwards.hpp>

Public Functions

inline std::string **name**() const

inline void **check_attribute_size**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline std::vector<std::size_t> **calc_spatial_lens**(std::vector<std::size_t> x_lens, std::vector<std::size_t> w_lens) const

inline *shape* **dynamic_compute_shape**(*shape* x_shape, *shape* w_shape) const

inline *shape* **static_compute_shape**(*shape* x_shape, *shape* w_shape) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline size_t **kdims**() const

Public Members

std::vector<std::size_t> **padding** = {0, 0}

std::vector<std::size_t> **stride** = {1, 1}

std::vector<std::size_t> **dilation** = {1, 1} *padding_mode_t* **padding_mode** = default_

int **group** = 1

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **cos** : public migraphx::internal::op::unary<*cos*>

#include <migraphx/op/cos.hpp>

Public Functions

inline auto **apply**() const

struct **cosh** : public migraphx::internal::op::unary<*cosh*>

#include <migraphx/op/cosh.hpp>

Public Functions

inline auto **apply**() const

struct **dequantizelinear**

#include <migraphx/op/dequantizelinear.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

struct **dimensions_of**

#include <migraphx/op/dimensions_of.hpp> Returns the dimensions of the input argument from starting axis to ending axis. Atleast `end` must be set to use this operator (set `end` to ndim for default ONNX behavior of `Shape` operator) This should only be used for dynamic shapes as this can be simplified to a literal for static shapes.

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

Public Members

std::size_t **start** = 0

std::size_t **end** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **div** : public migraphx::internal::op::binary<*div*>

#include <migraphx/op/div.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

struct **dot**

#include <migraphx/op/dot.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

struct **elu** : public migraphx::internal::op::unary<*elu*>

#include <migraphx/op/elu.hpp>

Public Functions

inline std::string **point_op**() const

inline auto **apply**() const

Public Members

float **alpha** = 1

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **equal** : public migraphx::internal::op::binary<*equal*>

#include <migraphx/op/equal.hpp>

Public Functions

inline value **attributes**() const

inline std::string **point_function**() const

inline auto **apply**() const

struct **erf** : public migraphx::internal::op::unary<*erf*>

#include <migraphx/op/erf.hpp>

Public Functions

inline auto **apply**() const

struct **exp** : public migraphx::internal::op::unary<*exp*>

#include <migraphx/op/exp.hpp>

Public Functions

inline auto **apply**() const

struct **fill**

#include <migraphx/op/fill.hpp> fill(default_value, output_buffer) Fill an output buffer with the given default_value. Note that if the default_value is a literal and the output_buffer has a static shape this operator can be replaced with a literal.

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

struct **flatten**

#include <migraphx/op/flatten.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

int64_t **axis** = 1

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **floor** : public migraphx::internal::op::unary<*floor*>

#include <migraphx/op/floor.hpp>

Public Functions

inline auto **apply**() const

struct **fmod** : public migraphx::internal::op::binary<*fmod*>

#include <migraphx/op/fmod.hpp>

Public Functions

inline std::string **name**() const

inline value **attributes**() const

inline auto **apply**() const

struct **gather**

#include <migraphx/op/gather.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

Public Members

int64_t **axis** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **gathernd**

#include <migraphx/op/gathernd.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

Public Members

int **batch_dims** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **get_tuple_elem**

#include <migraphx/op/get_tuple_elem.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const *shape*&, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::size_t **index** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **greater** : public migraphx::internal::op::binary<*greater*>

#include <migraphx/op/greater.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

struct **gru**

#include <migraphx/op/gru.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

Public Members

std::size_t **hidden_size** = 1

std::vector<operation> **actv_funcs** = {*sigmoid*{}, *tanh*{}}

rnn_direction **direction** = *rnn_direction*::*forward*

float **clip** = 0.0f

int **linear_before_reset** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **highest**

#include <migraphx/op/reduce_op.hpp>

Public Functions

template<class **T**>
inline **operator** *T*() const

struct **identity**

#include <migraphx/op/identity.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(*shape*, std::vector<*argument*> args) const

inline value **attributes**() const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

struct **if_op**

#include <migraphx/op/if_op.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs, std::vector<module_ref> mods) const

inline *argument* **compute**(const *shape*&, const std::vector<*argument*> &args, const std::vector<module_ref> &mods, const std::function<std::vector<*argument*>(module_ref&, const std::unordered_map<std::string, *argument*>&)> &run) const

struct **im2col**

#include <migraphx/op/im2col.hpp>

Public Functions

inline std::string **name**() const

inline value **attributes**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

Public Members

std::vector<std::size_t> **padding** = {0, 0}

std::vector<std::size_t> **stride** = {1, 1}

std::vector<std::size_t> **dilation** = {1, 1}

padding_mode_t **padding_mode** = default_

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **isinf** : public migraphx::internal::op::unary<*isinf*>

#include <migraphx/op/isinf.hpp>

Public Functions

inline auto **apply**() const

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

struct **isnan** : public migraphx::internal::op::unary<*isnan*>

#include <migraphx/op/isnan.hpp>

Public Functions

inline auto **apply**() const

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

struct **layout** : public migraphx::internal::op::unary<*layout*>

#include <migraphx/op/layout.hpp>

Public Functions

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline auto **apply**() const

Public Members

std::vector<int64_t> **permutation**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **leaky_relu** : public migraphx::internal::op::unary<*leaky_relu*>

#include <migraphx/op/leaky_relu.hpp>

Public Functions

inline std::string **point_op**() const

inline std::string **name**() const

inline auto **apply**() const

Public Members

float **alpha** = 0.01

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **less** : public migraphx::internal::op::binary<*less*>

#include <migraphx/op/less.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

struct **load**

#include <migraphx/op/load.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs) const

inline *argument* **compute**(const *shape*&, const std::vector<*argument*> &args) const

inline lifetime **get_lifetime**() const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

shape **s**

std::size_t **offset** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

Friends

inline friend std::ostream &**operator<<**(std::ostream &os, const *load* &op)

struct **log** : public migraphx::internal::op::unary<*log*>

#include <migraphx/op/log.hpp>

Public Functions

inline auto **apply**() const

struct **logical_and** : public migraphx::internal::op::binary<*logical_and*>

#include <migraphx/op/logical_and.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

struct **logical_or** : public migraphx::internal::op::binary<*logical_or*>

#include <migraphx/op/logical_or.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

struct **logical_xor** : public migraphx::internal::op::binary<*logical_xor*>

#include <migraphx/op/logical_xor.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

struct **logsoftmax**

#include <migraphx/op/logsoftmax.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline auto **output**() const

Public Members

int64_t **axis** = 1

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **loop**

#include <migraphx/op/loop.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs, std::vector<module_ref> mods) const

inline *argument* **compute**(context &ctx, const *shape* &out_shape, const std::vector<*argument*> &args, const std::vector<module_ref> &mods, const std::function<std::vector<*argument*>(module_ref&, const std::unordered_map<std::string, *argument*>&)> &run) const

Public Members

int64_t **max_iterations** = 10

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **ref_loop**

#include <migraphx/op/loop.hpp>

Public Functions

template<class **T**>
inline void **copy**(context&, const *argument* &src, *T* &dst) const

template<class **T**>
inline void **copy**(context&, *T* src, const *argument* &dst) const

inline void **append**(const std::vector<*argument*> &iter_state, const std::vector<*argument*> &concatenated_outputs, int iter) const

inline void **set_zero**(context&, const std::vector<*argument*> &concatenated_outputs, int iter) const

inline std::unordered_map<std::string, int> **get_output_params**(const module&) const

Public Members

int64_t **max_iterations** = 0

struct **lowest**

#include <migraphx/op/reduce_op.hpp>

Public Functions

template<class **T**>
inline **operator** *T*() const

struct **lrn**

#include <migraphx/op/lrn.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

Public Members

float **alpha** = 0.0001

float **beta** = 0.75

float **bias** = 1.0

int **size** = 1

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **lstm**

#include <migraphx/op/lstm.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

Public Members

std::size_t **hidden_size** = 1

std::vector<operation> **actv_funcs** = {*sigmoid*{}, *tanh*{}, *tanh*{}}

rnn_direction **direction** = *rnn_direction*::*forward*

float **clip** = 0.0f

int **input_forget** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **max** : public migraphx::internal::op::binary<*max*>

#include <migraphx/op/max.hpp>

Public Functions

inline value **attributes**() const

inline auto **apply**() const

struct **min** : public migraphx::internal::op::binary<*min*>

#include <migraphx/op/min.hpp>

Public Functions

inline value **attributes**() const

inline auto **apply**() const

struct **mod** : public migraphx::internal::op::binary<*mod*>

#include <migraphx/op/mod.hpp>

Public Functions

inline std::string **name**() const

inline value **attributes**() const

inline auto **apply**() const

struct **mul** : public migraphx::internal::op::binary<*mul*>

#include <migraphx/op/mul.hpp>

Public Functions

inline value **attributes**() const

inline std::string **point_function**() const

inline auto **apply**() const

struct **multibroadcast**

#include <migraphx/op/multibroadcast.hpp> Broadcast multiple dimensions between two tensors. Two versions of this operator: 1 input and 2+ inputs. One input version uses output_lens attribute and broadcasts to it. 2+ inputs version broadcasts first input to the common shape at evaluation time.

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::vector<std::size_t> **output_lens** = {}

std::vector<*shape*::dynamic_dimension> **output_dyn_dims** = {}

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **multinomial**

#include <migraphx/op/multinomial.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

Public Members

shape::type_t **dtype** = *shape*::type_t::int32_type

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **nearbyint** : public migraphx::internal::op::unary<*nearbyint*>

#include <migraphx/op/nearbyint.hpp>

Public Functions

inline auto **apply**() const

struct **neg** : public migraphx::internal::op::unary<*neg*>

#include <migraphx/op/neg.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

struct **nonmaxsuppression**

#include <migraphx/op/nonmaxsuppression.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

template<class **T**>
inline *box* **batch_box**(*T* boxes, std::size_t box_idx) const

inline bool **suppress_by_iou**(*box* b1, *box* b2, double iou_threshold) const

template<class **T**>
inline std::vector<std::pair<double, int64_t>> **filter_boxes_by_score**(*T* scores_start, std::size_t num_boxes, double score_threshold) const

template<class **Output**, class **Boxes**, class **Scores**>
inline std::size_t **compute_nms**(*Output* output, *Boxes* boxes, *Scores* scores, std::size_t max_output_boxes_per_class, double iou_threshold, double score_threshold) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

Public Members

bool **center_point_box** = false

bool **use_dyn_output** = false

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **box**

#include <migraphx/op/nonmaxsuppression.hpp>

Public Functions

inline void **sort**()

inline std::array<double, 2> &**operator[]**(std::size_t i)

inline double **area**() const

Public Members

std::array<double, 2> **x**

std::array<double, 2> **y**

struct **nonzero**

#include <migraphx/op/nonzero.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

struct **one**

#include <migraphx/op/reduce_op.hpp>

Public Functions

template<class **T**>
inline **operator** *T*() const

template<class **Derived**>

struct **op_name**

#include <migraphx/op/name.hpp> Create name from class.

Subclassed by *migraphx::internal::op::binary< add >*, *migraphx::internal::op::binary< div >*, *migraphx::internal::op::binary< equal >*, *migraphx::internal::op::binary< fmod >*, *migraphx::internal::op::binary< greater >*, *migraphx::internal::op::binary< less >*, *migraphx::internal::op::binary< logical_and >, migraphx::internal::op::binary< logical_or >, migraphx::internal::op::binary< logical_xor >, migraphx::internal::op::binary< max >, migraphx::internal::op::binary< min >, migraphx::internal::op::binary< mod >, migraphx::internal::op::binary< mul >, migraphx::internal::op::binary< pow >, migraphx::internal::op::binary< prelu >, migraphx::internal::op::binary< sqdiff >, migraphx::internal::op::binary< sub >, migraphx::internal::op::prefix_scan_op< prefix_scan_sum >, migraphx::internal::op::reduce_op< reduce_max >, migraphx::internal::op::reduce_op< reduce_mean >, migraphx::internal::op::reduce_op< reduce_min >, migraphx::internal::op::reduce_op< reduce_prod >, migraphx::internal::op::reduce_op< reduce_sum >, migraphx::internal::op::scatter_op< scatter_add >, migraphx::internal::op::scatter_op< scatter_max >, migraphx::internal::op::scatter_op< scatter_min >, migraphx::internal::op::scatter_op< scatter_mul >, migraphx::internal::op::scatter_op< scatter_none >, migraphx::internal::op::scatternd_op< scatternd_add >, migraphx::internal::op::scatternd_op< scatternd_max >, migraphx::internal::op::scatternd_op< scatternd_min >, migraphx::internal::op::scatternd_op< scatternd_mul >, migraphx::internal::op::scatternd_op< scatternd_none >, migraphx::internal::op::unary< abs >, migraphx::internal::op::unary< acos >, migraphx::internal::op::unary< acosh >, migraphx::internal::op::unary< asin >, migraphx::internal::op::unary< asinh >, migraphx::internal::op::unary< atan >, migraphx::internal::op::unary< atanh >, migraphx::internal::op::unary< ceil >, migraphx::internal::op::unary< convert >, migraphx::internal::op::unary< cos >, migraphx::internal::op::unary< cosh >, migraphx::internal::op::unary< elu >, migraphx::internal::op::unary< erf >, migraphx::internal::op::unary< exp >, migraphx::internal::op::unary< floor >, migraphx::internal::op::unary< isinf >, migraphx::internal::op::unary< isnan >, migraphx::internal::op::unary< layout >, migraphx::internal::op::unary< leaky_relu >, migraphx::internal::op::unary< log >, migraphx::internal::op::unary< nearbyint >, migraphx::internal::op::unary< neg >, migraphx::internal::op::unary< recip >, migraphx::internal::op::unary< relu >, migraphx::internal::op::unary< rsqrt >, migraphx::internal::op::unary< sigmoid >, migraphx::internal::op::unary< sign >, migraphx::internal::op::unary< sin >, migraphx::internal::op::unary< sinh >, migraphx::internal::op::unary< sqrt >, migraphx::internal::op::unary< tan >, migraphx::internal::op::unary< tanh >, migraphx::internal::op::unary< unary_not >, migraphx::internal::op::binary< Derived >, migraphx::internal::op::prefix_scan_op< Derived >, migraphx::internal::op::reduce_op< Derived >, migraphx::internal::op::scatter_op< Derived >, migraphx::internal::op::scatternd_op< Derived >, migraphx::internal::op::unary< Derived >*

Public Functions

inline std::string **name**() const

struct **outline**

#include <migraphx/op/outline.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs) const

inline *argument* **compute**(const *shape*&, const std::vector<*argument*>&) const

Public Members

shape **s**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **pad**

#include <migraphx/op/pad.hpp>

Public Types

enum **pad_op_mode_t**

Values:

enumerator **constant_pad**

enumerator **reflect_pad**

enumerator **edge_pad**

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline std::size_t **pad_ndims**() const

inline bool **symmetric**() const

Public Members

std::vector<int64_t> **pads**

float **value** = 0.0f

pad_op_mode_t **mode** = constant_pad

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **pointwise**

#include <migraphx/op/pointwise.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs, std::vector<module_ref> mods) const

inline *argument* **compute**(const *shape* &output_shape, const std::vector<*argument*> &args, const std::vector<module_ref> &mods, const std::function<std::vector<*argument*>(module_ref&, const std::unordered_map<std::string, *argument*>&)> &run) const

struct **pooling**

#include <migraphx/op/pooling.hpp>

Public Functions

inline std::string **name**() const

inline void **check_attribute_size**() const

inline size_t **kdims**() const

inline value **attributes**() const

inline std::size_t **dilate_dim**(std::size_t dim, std::size_t dilation) const

inline std::vector<std::size_t> **calc_spatial_dim_out**(const std::vector<std::size_t> &input_lens, std::size_t kdims) const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

template<class **Type**, class **Out**, class **In**, class **Op**>
inline void **calc_pooling**(const *shape* &output_shape, *Out* &output, const *In* &input, const std::vector<std::size_t> &kernel_dims, const std::vector<std::size_t> &padding_vals, *Op* op) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

Public Members

pooling_mode **mode** = {*pooling_mode*::*average*}

std::vector<std::size_t> **padding** = {0, 0}

std::vector<std::size_t> **stride** = {1, 1}

std::vector<std::size_t> **lengths** = {1, 1}

std::vector<std::size_t> **dilations** = {1, 1}

bool **ceil_mode** = false

int **lp_order** = 2

padding_mode_t **padding_mode** = *padding_mode_t*::*default_*

bool **dyn_global** = false

bool **count_include_pad** = false

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **avg_pool**

#include <migraphx/op/pooling.hpp>

Public Functions

template<class **T**>
inline double **init**() const

inline double **operator()**(double x, double y) const

inline double **final**(double x, std::size_t y) const

struct **lpnorm_pool**

#include <migraphx/op/pooling.hpp>

Public Functions

lpnorm_pool() = delete

inline explicit **lpnorm_pool**(int x)

template<class **T**>
inline double **init**() const

inline double **operator()**(double x, double y) const

inline double **final**(double x, std::size_t) const

Public Members

int **p** = 0

struct **max_pool**

#include <migraphx/op/pooling.hpp>

Public Functions

template<class **T**>
inline *T* **init**() const

inline double **operator()**(double x, double y) const

inline double **final**(double x, std::size_t) const

struct **pow** : public migraphx::internal::op::binary<*pow*>

#include <migraphx/op/pow.hpp>

Public Functions

inline auto **apply**() const

template<class **Derived**>

struct **prefix_scan_op** : public migraphx::internal::op::op_name<*Derived*>

#include <migraphx/op/prefix_scan_op.hpp> Parent struct for prefix scan operations. A prefix scan is equivalent to the C++ std::exclusive_scan or std::inclusive_scan. Given a list of numbers, a prefix scan sum op returns an equal size list of running totals of the values. Other operations besides addition can be supported by their own child ops.

Public Functions

inline value **attributes**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline auto **init**() const

inline **prefix_scan_op**()

inline **prefix_scan_op**(int64_t ax)

inline **prefix_scan_op**(int64_t ax, bool excl)

inline **prefix_scan_op**(int64_t ax, bool excl, bool rev)

Public Members

int64_t **axis**

bool **exclusive** = false

bool **reverse** = false

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **prefix_scan_sum** : public migraphx::internal::op::prefix_scan_op<*prefix_scan_sum*>

#include <migraphx/op/prefix_scan_sum.hpp>

Public Functions

inline **prefix_scan_sum**()

inline **prefix_scan_sum**(int64_t ax)

inline **prefix_scan_sum**(int64_t ax, bool excl)

inline **prefix_scan_sum**(int64_t ax, bool excl, bool rev)

inline auto **op**() const

struct **prelu** : public migraphx::internal::op::binary<*prelu*>

#include <migraphx/op/prelu.hpp>

Public Functions

inline std::string **point_op**() const

inline auto **apply**() const

struct **quant_convolution**

#include <migraphx/op/quant_convolution.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline void **check_attribute_size**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline size_t **kdims**() const

inline *argument* **compute**(*shape* output_shape, std::vector<*argument*> args) const

Public Members

std::vector<std::size_t> **padding** = {0, 0}

std::vector<std::size_t> **stride** = {1, 1}

std::vector<std::size_t> **dilation** = {1, 1}

padding_mode_t **padding_mode** = default_

int **group** = 1

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **quant_dot**

#include <migraphx/op/quant_dot.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

struct **quantizelinear**

#include <migraphx/op/quantizelinear.hpp>

Public Functions

inline std::string **name**() const

inline value **attributes**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

struct **random_seed**

#include <migraphx/op/random_seed.hpp> Generates a random seed for the use of random number generators. Generating the seed at runtime guarantees there will be a different random sequence on every execution. This operation has no inputs or attributes, and outputs an unsigned integer tensor with a single value.

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs) const

inline *argument* **compute**(const *shape* &output_shape, const std::vector<*argument*>&) const

Public Members

shape::type_t **dtype** = *shape*::type_t::uint64_type

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **random_uniform**

#include <migraphx/op/random_uniform.hpp> *random_uniform* populates the passed shape with random numbers, in a uniform distribution. Range for floating-point data types is (0, 1); for integer types it is [0, <max value for the type>]

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

struct **recip** : public migraphx::internal::op::unary<*recip*>

#include <migraphx/op/recip.hpp>

Public Functions

inline std::string **point_op**() const

inline auto **apply**() const

struct **reduce_max** : public migraphx::internal::op::reduce_op<*reduce_max*>

#include <migraphx/op/reduce_max.hpp>

Public Functions

inline **reduce_max**()

inline **reduce_max**(std::vector<int64_t> ax)

inline auto **op**() const

inline auto **init**() const

struct **reduce_mean** : public migraphx::internal::op::reduce_op<*reduce_mean*>

#include <migraphx/op/reduce_mean.hpp>

Public Functions

inline **reduce_mean**()

inline **reduce_mean**(std::vector<int64_t> ax)

inline auto **op**() const

inline auto **output**(const *shape* &s) const

struct **reduce_min** : public migraphx::internal::op::reduce_op<*reduce_min*>

#include <migraphx/op/reduce_min.hpp>

Public Functions

inline **reduce_min**()

inline **reduce_min**(std::vector<int64_t> ax)

inline auto **op**() const

inline auto **init**() const

template<class **Derived**>

struct **reduce_op** : public migraphx::internal::op::op_name<*Derived*>

#include <migraphx/op/reduce_op.hpp>

Public Functions

inline value **attributes**() const

inline std::vector<int64_t> **tune_axes**(std::size_t n_dim) const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

returns a shape in which the axis or axes named for reduction by this op are set, to size 1.

Parameters

inputs – list of input shapes

Returns

shape

template<class **T**>
inline void **tune_dims**(const std::vector<int64_t> &tuned_axes, const std::vector<*T*> &in_lens, std::vector<*T*> &out_lens) const

template<class **T**>
inline void **reduce**(tensor_view<*T*> &input, *shape* &batch_shape, std::vector<int64_t> &tuned_axes, std::vector<std::size_t> &out_idx, tensor_view<*T*> &output) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline auto **init**() const

inline auto **input**() const

inline auto **output**(const *shape*&) const

inline **reduce_op**()

inline **reduce_op**(std::vector<int64_t> ax)

Public Members

std::vector<std::int64_t> **axes** = {}

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **reduce_prod** : public migraphx::internal::op::reduce_op<*reduce_prod*>

#include <migraphx/op/reduce_prod.hpp>

Public Functions

inline **reduce_prod**()

inline **reduce_prod**(std::vector<int64_t> ax)

inline auto **op**() const

inline auto **init**() const

struct **reduce_sum** : public migraphx::internal::op::reduce_op<*reduce_sum*>

#include <migraphx/op/reduce_sum.hpp>

Public Functions

inline **reduce_sum**()

inline **reduce_sum**(std::vector<int64_t> ax)

inline auto **op**() const

struct **relu** : public migraphx::internal::op::unary<*relu*>

#include <migraphx/op/relu.hpp>

Public Functions

inline std::string **point_op**() const

inline auto **apply**() const

struct **reshape**

#include <migraphx/op/reshape.hpp> 1 input version: reshape(input_data) this.dims = output_dims Makes a copy of input_data to the output shape.

2 input version: reshape(input_data, output_buffer) this.dims = unset Copies input_data to output_buffer; output_buffer already has the output shape. This version will not fail gracefully if the input shape and output_buffer shape are incompatible. There's a throw that will catch when the number of elements do not

match at runtime. This version should only be used for dynamic reshapes (output dimensions only known at runtime). If output_buffer has a static shape during compile/parse, you can use the 1 input version.

Public Functions

inline std::string **name**() const

inline *shape* **dyn_compute_shape**(*shape* s0) const

inline *shape* **static_compute_shape**(std::vector<*shape*> inputs, std::size_t n_neg_dims) const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

Public Members

std::vector<int64_t> **dims**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **reshape_lazy**

#include <migraphx/op/reshape_lazy.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **dyn_compute_shape**(*shape* s0) const

inline *shape* **static_compute_shape**(std::vector<*shape*> inputs, std::size_t n_neg_dims) const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::vector<int64_t> **dims**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

template<class **Iterator**>
static inline auto **compute_end_dim**(*Iterator* start, *Iterator* last, std::size_t dim)

template<class **OptionalPair**>
static inline *OptionalPair* **try_merge_pairs**(*OptionalPair* p2, *OptionalPair* p1)

template<class **DimIterator**, class **StrideIterator**>
static inline optional<std::size_t> **merge_strides**(*DimIterator* dim_start, *DimIterator* dim_last, *StrideIterator* stride_start, *StrideIterator* stride_last)

template<class **DimIterator**, class **StrideIterator**>
static inline auto **can_strides_merge**(*DimIterator* dim_start, *DimIterator* dim_last, *StrideIterator* stride_start, *StrideIterator* stride_last)

static inline optional<*shape*> **reshape_lazy_dims**(const *shape* &input, const std::vector<std::size_t> &rdims)

struct **reverse**

#include <migraphx/op/reverse.hpp>

Public Functions

inline std::string **name**() const

inline value **attributes**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const *shape* &s, std::vector<*argument*> args) const

Public Members

std::vector<int64_t> **axes**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **rnn**

#include <migraphx/op/rnn.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

Public Members

std::size_t **hidden_size** = 1

std::vector<operation> **actv_funcs** = {*tanh*{}, *tanh*{}}

rnn_direction **direction** = *rnn_direction*::*forward*

float **clip** = 0.0f

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **rnn_last_cell_output**

#include <migraphx/op/rnn_last_cell_output.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

struct **rnn_last_hs_output**

#include <migraphx/op/rnn_last_hs_output.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

struct **rnn_var_sl_last_output**

#include <migraphx/op/rnn_var_sl_last_output.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

Public Members

rnn_direction **direction** = *rnn_direction*::*forward*

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **rnn_var_sl_shift_output**

#include <migraphx/op/rnn_variable_seq_lens.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

Public Members

std::string **output_name** = "hidden_states"

rnn_direction **direction** = *rnn_direction*::*forward*

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **rnn_var_sl_shift_sequence**

#include <migraphx/op/rnn_variable_seq_lens.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

struct **roialign**

#include <migraphx/op/roialign.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline auto **calc_pos_weight**(const std::array<std::size_t, 2> &dims, const *shape* &comp_s, const std::array<float, 2> &roi_start, const std::array<float, 2> &bin_size, const std::array<std::size_t, 2> &bin_grid_size) const

template<class **T**, class **Op**>
inline std::tuple<double, int64_t> **calc_pooling**(const *T* &data, const std::array<std::size_t, 2> &bin_grid_size, const std::vector<*pos_weight*> &pos_weights, int64_t index, *Op* op) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

Public Members

std::string **coord_trans_mode** = "half_pixel"

pooling_mode **mode** = {*pooling_mode*::*average*}

int64_t **output_height** = 1

int64_t **output_width** = 1

int64_t **sampling_ratio** = 0

float **spatial_scale** = 1.0f

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **avg_pool**

#include <migraphx/op/roialign.hpp>

Public Functions

inline double **init**()

inline double **operator()**(double x, double y)

inline double **final**(double x, std::size_t y)

struct **max_pool**

#include <migraphx/op/roialign.hpp>

Public Functions

inline double **init**()

inline double **operator()**(double x, double y)

inline double **final**(double x, std::size_t)

struct **pos_weight**

#include <migraphx/op/roialign.hpp>

Public Members

std::array<std::size_t, 4> **pos** = {0, 0, 0, 0}

std::array<float, 4> **w** = {0.0f, 0.0f, 0.0f, 0.0f}

struct **rsqrt** : public migraphx::internal::op::unary<*rsqrt*>

#include <migraphx/op/rsqrt.hpp>

Public Functions

inline auto **apply**() const

struct **run_on_target**

#include <migraphx/op/run_on_target.hpp>

Public Functions

inline std::string **name**() const

inline migraphx::*shape* **compute_shape**(const std::vector<migraphx::*shape*> &inputs, std::vector<migraphx::module_ref> mods) const

inline migraphx::*argument* **compute**(const migraphx::*shape*&, const std::vector<migraphx::*argument*> &args, const std::vector<migraphx::module_ref> &mods, const std::function<std::vector<migraphx::*argument*>(migraphx::module_ref&, const std::unordered_map<std::string, migraphx::*argument*>&)> &run) const

Public Members

std::size_t **target_id** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **scalar**

#include <migraphx/op/scalar.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(*shape* output_shape, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::vector<std::size_t> **scalar_bcast_lens**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **scatter_add** : public migraphx::internal::op::scatter_op<*scatter_add*>

#include <migraphx/op/scatter_add.hpp>

Public Functions

inline auto **reduction**() const

struct **scatter_max** : public migraphx::internal::op::scatter_op<*scatter_max*>

#include <migraphx/op/scatter_max.hpp>

Public Functions

inline auto **reduction**() const

struct **scatter_min** : public migraphx::internal::op::scatter_op<*scatter_min*>

#include <migraphx/op/scatter_min.hpp>

Public Functions

inline auto **reduction**() const

struct **scatter_mul** : public migraphx::internal::op::scatter_op<*scatter_mul*>

#include <migraphx/op/scatter_mul.hpp>

Public Functions

inline auto **reduction**() const

struct **scatter_none** : public migraphx::internal::op::scatter_op<*scatter_none*>

#include <migraphx/op/scatter_none.hpp>

Public Functions

inline auto **reduction**() const

template<typename **Derived**>

struct **scatter_op** : public migraphx::internal::op::op_name<*Derived*>

#include <migraphx/op/scatter_op.hpp>

Public Functions

inline value **attributes**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

inline const *Derived* &**derived**() const

Public Members

int64_t **axis** = 0

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **scatternd_add** : public migraphx::internal::op::scatternd_op<*scatternd_add*>

#include <migraphx/op/scatternd_add.hpp>

Public Functions

inline **scatternd_add**()

inline auto **reduction**() const

struct **scatternd_max** : public migraphx::internal::op::scatternd_op<*scatternd_max*>

#include <migraphx/op/scatternd_max.hpp>

Public Functions

inline **scatternd_max**()

inline auto **reduction**() const

struct **scatternd_min** : public migraphx::internal::op::scatternd_op<*scatternd_min*>

#include <migraphx/op/scatternd_min.hpp>

Public Functions

inline **scatternd_min**()

inline auto **reduction**() const

struct **scatternd_mul** : public migraphx::internal::op::scatternd_op<*scatternd_mul*>

#include <migraphx/op/scatternd_mul.hpp>

Public Functions

inline **scatternd_mul**()

inline auto **reduction**() const

struct **scatternd_none** : public migraphx::internal::op::scatternd_op<*scatternd_none*>

#include <migraphx/op/scatternd_none.hpp>

Public Functions

inline **scatternd_none**()

inline auto **reduction**() const

template<class **Derived**>

struct **scatternd_op** : public migraphx::internal::op::op_name<*Derived*>

#include <migraphx/op/scatternd_op.hpp> N-dimensional Scatter operations. This struct is parent class to ops which differ in what formula is used to reduce (combine old and new values of) the scattered value. It was originally based on Onnx ScatterND operation (see https://github.com/onnx/onnx/blob/main/docs/Operators.md#ScatterND) and is also similar to Numpy numpy.add.at().

Template Parameters

Derived – a template parameter in the CRTP inheritance idiom, represents one of the child operations.

Public Functions

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

Validate input shapes and return the correct output shape. For Scatter ops, the output is the same shape as the data tensor (first input), but cast to a standard shape.

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline auto **init**() const

inline **scatternd_op**()

struct **select_module**

#include <migraphx/op/select_module.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs, const std::vector<module_ref>&) const

inline std::vector<std::string> **get_input_parameter_names**(module_ref mod) const

inline std::vector<std::string> **get_output_parameter_names**(module_ref mod) const

inline *argument* **compute**(const *shape*&, const std::vector<*argument*> &args, const std::vector<module_ref> &submodule_list, const std::function<std::vector<*argument*>(module_ref&, const std::unordered_map<std::string, *argument*>&)> &run) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*> &shapes) const

Public Members

shape **output_dyn_shapes**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **sigmoid** : public migraphx::internal::op::unary<*sigmoid*>

#include <migraphx/op/sigmoid.hpp>

Public Functions

inline std::string **point_op**() const

inline auto **apply**() const

struct **sign** : public migraphx::internal::op::unary<*sign*>

#include <migraphx/op/sign.hpp>

Public Functions

inline std::string **point_op**() const

inline auto **apply**() const

struct **sin** : public migraphx::internal::op::unary<*sin*>

#include <migraphx/op/sin.hpp>

Public Functions

inline auto **apply**() const

struct **sinh** : public migraphx::internal::op::unary<*sinh*>

#include <migraphx/op/sinh.hpp>

Public Functions

inline auto **apply**() const

struct **slice**

#include <migraphx/op/slice.hpp> Slice operator that accepts variable axes, starts and ends. All of `starts`, `ends`, and `axes` must be supplied by either their attribute or an input (but not both).

Valid calls: slice(input); axes, starts, ends set slice(input, starts); axes, ends set slice(input, ends); starts, axes set slice(input, axes); starts, ends set slice(input, starts, ends); axes set slice(input, starts, axes); ends set slice(input, ends, axes); starts set slice(input, start, ends, axes); none set

Attributes: axes: constant axes to slice over (optional) starts: constant slice starting indices (optional) ends: constant slice ending indices (optional)

Parameters: data: the input tensor to slice (dynamic or static shape) input_starts: starting indices of slice (optional, static shape) input_ends: ending indices of slice (optional, static shape) input_axes: axes to slice over (optional, static shape)

Public Functions

inline value **attributes**() const

Ensure that attribute axes is within limits. Will attempt to normalize starts and ends; but will use the dynamic_dimension.max values for dynamic shapes. This makes it so you have to renormalize for non-fixed dynamic_dimensions.

inline std::string **name**() const

template<class **A**, class **B**>
inline std::vector<std::size_t> **lens_calc**(const std::vector<std::size_t> &lengths, *A* in_starts, *A* in_ends, *B* in_axes) const

Computes the slice output shape dimensions for given starts, ends,and axes. Templated to also handle tensor views. Possibly different type between [in_starts, in_ends] and [in_axes] if in_axes is this object's axes attribute. Assumes in_starts and in_ends are normalized; in_axes are valid.

inline std::array<bool, 3> **get_set_attributes**() const

Get the attributes that are non-empty.

inline *shape* **compute_two_or_more**(std::vector<*shape*> inputs) const

Helper function for *normalize_compute_shape()*

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline auto **compute_offset**(const *shape* &s) const

Calculates the starting offset for the sliced tensor. Used in compute when only data input and all other information are in the attributes.

Parameters
s – static input shape

template<class **T**>
inline auto **compute_offset**(const *shape* &s, const *T* &input_starts, const *T* &ax_vec) const

Calculates the starting offset for the sliced tensor (for aliasing). Used for 2-4 inputs to `slice.

Parameters
- **s** – static input shape
- **input_starts** – starting indices of slice
- **ax_vec** – axes to slice on

inline std::unordered_map<std::string, std::vector<int64_t>> **normalize_starts_ends_axes**(*shape* input_shape, const optional<std::vector<int64_t>> &input_starts, const optional<std::vector<int64_t>> &input_ends, const optional<std::vector<int64_t>> &input_axes) const

If given, normalize the inputs. Otherwise get from operator attributes. Return the values in a map.

Parameters input_shape: static shape of the input input_starts: optional input_ends: optional input_ends: optional

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::vector<int64_t> **axes** = {}

std::vector<int64_t> **starts** = {}

std::vector<int64_t> **ends** = {}

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

Public Static Attributes

static constexpr std::array<bool, 3> **all_set** = {true, true, true}

Named arrays for the set attribute possibilities.

static constexpr std::array<bool, 3> **ends_axes** = {false, true, true}

static constexpr std::array<bool, 3> **starts_axes** = {true, false, true}

static constexpr std::array<bool, 3> **starts_ends** = {true, true, false}

static constexpr std::array<bool, 3> **axes_only** = {false, false, true}

static constexpr std::array<bool, 3> **ends_only** = {false, true, false}

static constexpr std::array<bool, 3> **starts_only** = {true, false, false}

static constexpr std::array<bool, 3> **none_set** = {false, false, false}

struct **softmax**

#include <migraphx/op/softmax.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline auto **output**() const

Public Members

int64_t **axis** = 1

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **sqdiff** : public migraphx::internal::op::binary<*sqdiff*>

#include <migraphx/op/sqdiff.hpp>

Public Functions

inline std::string **point_op**() const

inline auto **apply**() const

struct **sqrt** : public migraphx::internal::op::unary<*sqrt*>

#include <migraphx/op/sqrt.hpp>

Public Functions

inline auto **apply**() const

struct **squeeze**

#include <migraphx/op/squeeze.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::vector<int64_t> **axes**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **step**

#include <migraphx/op/step.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(*shape* output_shape, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::vector<int64_t> **axes**

std::vector<int64_t> **steps**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **sub** : public migraphx::internal::op::binary<*sub*>

#include <migraphx/op/sub.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

struct **tan** : public migraphx::internal::op::unary<*tan*>

#include <migraphx/op/tan.hpp>

Public Functions

inline auto **apply**() const

struct **tanh** : public migraphx::internal::op::unary<*tanh*>

#include <migraphx/op/tanh.hpp>

Public Functions

inline auto **apply**() const

struct **topk**

#include <migraphx/op/topk.hpp>

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

template<class **T**, class **Compare**>
inline *heap_vector*<*T*, *Compare*> **make_heap**(std::vector<*T*> val, *Compare* compare) const

inline *argument* **compute**(const *shape* &output_shape, std::vector<*argument*> args) const

Public Members

int64_t **k** = 1

int64_t **axis** = 0

bool **largest** = true

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

template<class **T**, class **Compare**>

struct **heap_vector**

#include <migraphx/op/topk.hpp>

Public Functions

inline **heap_vector**(const std::vector<*T*> &val, *Compare* comp)

inline void **try_push**(*T* val)

inline std::vector<*T*> **sort**()

Public Members

std::vector<*T*> **data**

Compare **compare**

struct **transpose**

#include <migraphx/op/transpose.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::vector<int64_t> **dims**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

template<class **Derived**>

struct **unary** : public migraphx::internal::op::op_name<*Derived*>

#include <migraphx/op/unary.hpp>

Public Functions

inline std::string **point_function**() const

inline std::string **point_op**() const

inline value **base_attributes**() const

inline value **attributes**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

struct **unary_not** : public migraphx::internal::op::unary<*unary_not*>

#include <migraphx/op/unary_not.hpp>

Public Functions

inline std::string **point_function**() const

inline auto **apply**() const

inline std::string **name**() const

struct **undefined**

#include <migraphx/op/undefined.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(const std::vector<*shape*> &inputs) const

inline *argument* **compute**(const *shape*&, const std::vector<*argument*>&) const

struct **unique**

#include <migraphx/op/unique.hpp>

Public Functions

template<class **T**>
inline auto **make_idx_less_fn**(const *T* &data, size_t chunk_sz) const

template<class **T**>
inline auto **sorted_uniq_indices**(const *T* &input_data, size_t chunk_sz) const

template<class **T**>
inline auto **unsorted_uniq_indices**(const *T* &input_data, size_t chunk_sz) const

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

Public Members

std::optional<int64_t> **axis**

bool **sorted** = true

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **unknown**

#include <migraphx/op/unknown.hpp>

Public Functions

inline std::string **name**() const

inline *shape* **compute_shape**(std::vector<*shape*> input) const

Public Members

std::string **op**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

Friends

inline friend std::ostream &**operator**<<(std::ostream &os, const *unknown* &x)

struct **unsqueeze**

#include <migraphx/op/unsqueeze.hpp> Adds dimensions to a tensor based on the axes attribute. `axes` are based on the number of output shape dimensions and should not contain duplicates. `steps` are for modifying dimensions added to the middle of the original shape. Each step must be a factor of the original dimension. ex: unsqueeze(shape = [3, 4, 10], axes = [2, 4, 5], steps = [2]) -> shape = [3, 4, 2, 5, 1, 1] Dynamic shape version does not handle `steps`.

Public Functions

inline value **attributes**() const

inline std::string **name**() const

inline *shape* **normalize_compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

inline std::ptrdiff_t **output_alias**(const std::vector<*shape*>&) const

Public Members

std::vector<int64_t> **axes**

std::vector<int64_t> **steps**

Public Static Functions

template<class **Self**, class **F**>
static inline auto **reflect**(*Self* &self, *F* f)

struct **where**

#include <migraphx/op/where.hpp>

Public Functions

inline std::string **name**() const

inline value **attributes**() const

inline *shape* **compute_shape**(std::vector<*shape*> inputs) const

inline *argument* **compute**(const dyn_output &dyn_out, std::vector<*argument*> args) const

struct **zero**

#include <migraphx/op/reduce_op.hpp>

Public Functions

template<class **T**>
inline **operator** *T*() const

5.5 Program

5.5.1 instruction

struct **instruction**

Public Functions

inline **instruction**()

instruction(*operation* o, *shape* r, std::vector<*instruction_ref*> args)

instruction(*operation* o, *shape* r, std::vector<*instruction_ref*> args, std::vector<module_ref> modules)

instruction(*literal* l)

void **replace**(*operation* o)

void **recompute_shape**()

void **clear_arguments**()

bool **valid**(*instruction_ref* start, bool check_order = false) const

bool **valid**() const

shape **get_shape**() const

const *literal* &**get_literal**() const

const *operation* &**get_operator**() const

std::string **name**() const

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

const std::vector<module_ref> &**module_inputs**() const

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

Where this instruction is used as an input to another instruction.

void **add_output**(*instruction_ref* ins)

template<class **T**>
inline void **remove_output**(const *T* &ins)

bool **can_eval**() const

bool **is_undefined**() const

argument **eval**(bool check_eval = true) const

void **finalize**(context &ctx)

void **set_normalized**(bool value = true) bool **is_normalized**() const

bool **need_normalization**() const

operation **normalized_operator**() const

std::size_t **get_target_id**() const

void **set_target_id**(std::size_t tid)

void **debug_print**() const

Public Static Functions

static void **replace_refs**(*instruction_ref* ins, const std::unordered_map<*instruction_ref*, *instruction_ref*> &map_insts, const std::unordered_map<module_ref, module_ref> &map_mods)

static void **backreference**(*instruction_ref* ref)

static void **replace_argument**(*instruction_ref* ins, *instruction_ref* old, *instruction_ref* new_ins)

static void **replace_mod_argument**(*instruction_ref* ins, module_ref old, module_ref new_mod)

static void **replace**(*instruction_ref* ins, *operation* o, const *shape* &r, std::vector<*instruction_ref*> args)

static void **replace**(*instruction_ref* ins, *operation* o, const *shape* &r, std::vector<*instruction_ref*> args, std::vector<module_ref> module_args)

static *instruction_ref* **get_output_alias**(*instruction_ref* ins, bool shallow = false)

static void **print**(std::ostream &os, *instruction_ref* ins, const std::unordered_map<*instruction_ref*, std::string> &names)

Friends

friend bool **operator==**(const *instruction* &i, *instruction_ref* ref)

friend bool **operator==**(const *instruction* &x, const *instruction* &y)

friend bool **operator!=**(const *instruction* &x, const *instruction* &y)

friend bool **operator==**(*instruction_ref* ref, const *instruction* &i)

friend bool **operator!=**(const *instruction* &i, *instruction_ref* ref)

friend bool **operator!=**(*instruction_ref* ref, const *instruction* &i)

5.5.2 instruction_ref

type migraphx::internal::**instruction_ref**

References an instruction in the program.

5.5.3 program

struct **program**

Stores the instruction stream.

Public Functions

program()

program(*program*&&) noexcept

program(const *program*&)

program &**operator=**(*program*)

~program() noexcept

std::vector<std::string> **get_parameter_names**() const

shape **get_parameter_shape**(std::string name) const

instruction_ref **get_parameter**(std::string name) const

std::unordered_map<std::string, *shape*> **get_parameter_shapes**() const

std::vector<*argument*> **eval**(parameter_map params, execution_environment exec_env = execution_environment{}) const

void **finish**() const

std::size_t **size**() const

std::vector<*shape*> **get_output_shapes**() const

context &**get_context**() const

instruction_ref **validate**() const

target_assignments **get_target_assignments**(const std::vector<*target*> &targets, assignment_options options = assignment_options{})

void **compile**(const *target* &t, compile_options options = compile_options{})

void **compile**(const std::vector<*target*> &targets, std::vector<compile_options> compile_opts = {})

bool **is_compiled**() const

void **finalize**()

void **perf_report**(std::ostream &os, std::size_t n, parameter_map params, std::size_t batch = 1) const void **mark**(const parameter_map &params, marker &&m)

value **to_value**() const

void **from_value**(const value &v)

void **debug_print**() const

void **debug_print**(*instruction_ref* ins) const

void **print**(std::unordered_map<*instruction_ref*, std::string> &names, const std::function<void(*instruction_ref*, std::unordered_map<*instruction_ref*, std::string>)> &print_func) const

void **print**(const std::function<void(*instruction_ref* ins, std::unordered_map<*instruction_ref*, std::string>)> &print_func) const

void **print_graph**(std::ostream &os, bool brief = false) const

void **print_py**(std::ostream &os) const

void **print_cpp**(std::ostream &os) const

void **dry_run**(parameter_map params) const

void **annotate**(std::ostream &os, const std::function<void(*instruction_ref*)> &a) const

program &**sort**()

module ***create_module**(const std::string &name)

module ***create_module**(const std::string &name, module m)

module ***get_module**(const std::string &name)

const module ***get_module**(const std::string &name) const

module ***get_main_module**()

const module ***get_main_module**() const

std::vector<const module*> **get_modules**() const

std::vector<module*> **get_modules**()

std::unordered_multimap<module_ref, module_ref> **get_module_tree**()

void **remove_module**(const std::string &name)

void **remove_unused_modules**()

Friends

friend std::ostream &**operator<<**(std::ostream &os, const *program* &p)

friend bool **operator==**(const *program* &x, const *program* &y)

inline friend bool **operator!=**(const *program* &x, const *program* &y)

5.5.4 parse_onnx

program migraphx::internal::**parse_onnx**(const std::string &name, const *onnx_options*& = *onnx_options*{})

Create a program from an onnx file.

5.5.5 parse_tf

program migraphx::internal::**parse_tf**(const std::string &name, const *tf_options* &options = *tf_options*{})

Create a program from a tf pb file (default is nhwc format)

5.5.6 onnx_options

struct **onnx_options**

struct to pass in onnx options to parser

5.5.7 tf_options

struct **tf_options**

struct to pass in tf options to parser

5.6 Targets

5.6.1 target

struct **target**

An interface for a compilation target.

Public Functions

std::string **name**() const

A unique name used to identify the target.

std::vector<*pass*> **get_passes**(context &ctx, const compile_options &options) const

The transformation pass to be run during compilation.

Parameters

- **ctx** – This is the target-dependent context that is created by `get_context`
- **options** – Compiling options passed in by the user

Returns

The passes to be ran

context **get_context**() const

Construct a context for the target.

Returns

The context to be used during compilation and execution.

supported_segments **target_is_supported**(T&, const_module_ref mod, support_metric metric) const

Get the ranges of instructions that are supported on a target.

Parameters

- **module** – Module to check for supported instructions
- **metric** – Used to define how the quality of the support should be measured

Returns

the supported segments of the graph

argument **copy_to**(const *argument* &arg) const

copy an argument to the current target.

Parameters

arg – Input argument to be copied to the target

Returns

Argument in the target.

argument **copy_from**(const *argument* &arg) const

copy an argument from the current target.

Parameters

arg – Input argument to be copied from the target

Returns

Argument in the host.

argument **allocate**(const *shape* &s) const

Allocate an argument based on the input shape.

Parameters

s – Shape of the argument to be allocated in the target

Returns

Allocated argument in the target.

5.6.2 gpu::target

struct **target**

Public Functions

std::string **name**() const

std::vector<*pass*> **get_passes**(migraphx::context &gctx, const compile_options &options) const

migraphx::context **get_context**() const

argument **copy_to**(const *argument* &arg) const

argument **copy_from**(const *argument* &arg) const

argument **allocate**(const *shape* &s) const

5.6.3 cpu::target

struct **target**

Public Functions

std::string **name**() const

std::vector<*pass*> **get_passes**(migraphx::context &gctx, const compile_options&) const

inline migraphx::context **get_context**() const

inline *argument* **copy_to**(const *argument* &arg) const

inline *argument* **copy_from**(const *argument* &arg) const

argument **allocate**(const *shape* &s) const

5.7 Quantization

5.7.1 quantize_fp16

void migraphx::internal::**quantize_fp16**(*program* &prog, const std::vector<std::string> &ins_names = {"all"})

5.7.2 quantize_int8

void `migraphx::internal::quantize_int8`(*program* &prog, const *target* &t, const std::vector<parameter_map> &calibration, const std::unordered_set<std::string> &ins_names = {"dot", "convolution"})

5.8 Passes

5.8.1 pass

struct **pass**

An interface for applying a transformation to the instructions in a `program`

Public Functions

std::string **name**() const

A unique name used to identify the pass.

void **apply**(module_pass_manager &mpm) const

Run the pass on the module.

void **apply**(module &m) const

void **apply**(*program* &p) const

Run the pass on the program.

5.8.2 dead_code_elimination

struct **dead_code_elimination**

Remove instructions where the output is not used.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

void **apply**(*program* &p) const

5.8.3 eliminate_common_subexpression

struct **eliminate_common_subexpression**

Remove identical instructions.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

5.8.4 eliminate_concat

struct **eliminate_concat**

Remove concat operators by having each operator can write to different chunk of memory.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

Public Members

concat_optimization **concat_opt**

5.8.5 eliminate_contiguous

struct **eliminate_contiguous**

Remove contiguous instructions by checking if the operator can use non-standard shapes.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

Public Members

std::string **op_name**

5.8.6 eliminate_identity

struct **eliminate_identity**

Remove identity instructions. Currently when used as the last pass, it will preserve the semantics of previous program state, therefore dead code elimination should not be used afterwards.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

5.8.7 eliminate_pad

struct **eliminate_pad**

Remove pads if they can be written as an attribute to another op (im2col, convolution, pooling)

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

5.8.8 propagate_constant

struct **propagate_constant**

Replace instructions which take all literals with a literal of the computation.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

Public Members

std::unordered_set<std::string> **skip_ops** = { }

5.8.9 rewrite_rnn

struct **rewrite_rnn**

Rewrite rnn to gemm and add.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

5.8.10 schedule

struct **schedule**

Schedule instructions for concurrent execution

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

Public Members

schedule_model **model** = { }

bool **enable** = true

5.8.11 simplify_algebra

struct **simplify_algebra**

Simplify many algebraic instructions to more efficient versions.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

5.8.12 simplify_reshapes

struct **simplify_reshapes**

Eliminate redundant reshapes.

Public Functions

inline std::string **name**() const

void **apply**(module &m) const

Public Members

size_t **depth** = 4

5.9 Matchers

5.9.1 Introduction

The matchers provide a way to compose several predicates together. A matcher such as `m(m1, m2)` first checks a match for `m` followed by a match for `m1` and `m2` subsequently.

The most commonly used matcher is the `name` matcher. It matches the instruction with the operator equal to the name specified:

```
auto match_sum = name("sum");
```

The above matcher finds `sum` operators. To find `sum` operators with the output `standard_shape`, use:

auto match_sum = name("sum")(standard_shape());

5.9.2 Arguments

To match arguments in the instructions, match each argument using the `arg` matcher:

```
auto match_sum = name("sum")(arg(0)(name("@literal"), arg(1)(name("@literal"))));
```

The above matcher matches a `sum` operator with two arguments that are literals. Note that the `args` matcher eliminates the need to write `arg(0)` and `arg(1)` everytime:

```
auto match_sum = name("sum")(args(name("@literal"), name("@literal")));
```

5.9.3 Binding

To reference other instructions encountered while traversing through the instructions, use `.bind`:

```
auto match_sum = name("sum")(args(
                                name("@literal").bind("one"),
                                name("@literal").bind("two")
                            )).bind("sum");
```

This associates the instruction to a name that can be read from the `matcher_result` when it matches.

5.9.4 Finding matches

To use the matchers to find instructions, write a callback object that contains the matcher and an `apply` function that takes the `matcher_result` when the match is found:

```
struct match_find_sum
{
    auto matcher() const { return name("sum"); }

    void apply(program& p, matcher_result r) const
    {
        // Do something with the result
    }
};

find_matches(prog, match_find_sum{});
```

5.9.5 Creating matchers

The macros `MIGRAPH_BASIC_MATCHER` and `MIGRAPH_PRED_MATCHER` help in the creation of the matchers. Here is how you can create a matcher for shapes that are broadcasted:

```
MIGRAPH_PRED_MATCHER(broadcasted_shape, instruction_ref ins)
{
    return ins->get_shape().broadcasted();
}
```

For parameters to the predicate, use `make_basic_pred_matcher` to create the matcher. Here is how you can create a matcher to check the number of dimensions of the shape:

```
inline auto number_of_dims(std::size_t n)
{
    return make_basic_pred_matcher([=](instruction_ref ins) {
        return ins->get_shape().lens().size() == n;
    });
}
```

5.10 Tools

5.10.1 roctx.py

You can use the *roctx* command with rocprof binary to get marker timing information for each MIGraphX operator. To process timing information, use `roctx.py` helper script.

```
Usage: roctx.py [-h] [--json-path json_path] [--out out]
[--study-name study-name] [--repeat repeat] [--parse]
[--run run] [--debug]
```

The `roctx.py` helper script provides two main functionalities: `run` and `parse`.

`--run`

Runs `migraphx-driver roctx` command with the given `migraphx-driver` knobs followed by the parsing of the result which provides GPU kernel timing information. You can pass the MIGraphX knobs via a string to *–run* knob. See the _roctx-examples for usage.

`--parse`

Parses JSON file in the given `--json-path` and provides GPU kernel timing information.

`--out`

Output folder

`--study-name`

Optional. Allows user to name a study for easy interpretation. Defaults to timestamp.

`--repeat`

Number of iterations. Sets to **2** by default.

`--debug`

Provides additional debug information related to data. Use for debugging purposes only.

Examples:

Running inference with rocTX for a given ONNX file:

```
python roctx.py --run '--onnx --gpu fcn-resnet50-11.onnx' --out output_folder --repeat 5
```

Example output:

```
*** RESULTS ***
                                                    SUM_avg  MIN_avg  MAX_avg  COUNT
Marker start: gpu::convolution                         1625        8      103     31
Marker start: gpu::conv_bias_relu                      1212        8      450     18
Marker start: gpu::add_relu                             155        1       11     23
Marker start: gpu::conv_bias                            112       12       43      4
Marker start: load                                      110        0        2    160
Marker start: gpu::triadd_relu                           90        7       11     10
Marker start: hip::hip_copy_literal                      77        0        2    110
Marker start: broadcast                                  41        0        2     53
Marker start: gpu::concat                                39        6        7      6
Marker start: gpu::mul_add                               27        2        8      7
Marker start: gpu::sub                                   22        2        5      8
Marker start: slice                                      12        0        1     16
Marker start: gpu::pooling                                7        7        7      1
Marker start: step                                        3        3        3      1
Marker start: multibroadcast                              2        1        1      2
Marker start: @param                                      2        0        1      3
Marker start: hip::hip_allocate_memory                    1        1        1      1
Marker start: check_context::migraphx::version_...        0        0        0      0

AVG TOTAL TIME: 3544 us

OUTPUT CSV FILE:        output2021_11_04-03:03:02_AM.csv
KERNEL TIMING DETAILS:  rocTX_kernel_timing_details.txt
ALL DATA FROM ALL RUNS: rocTX_runs_dataframe.csv
```

Hotspot kerel timing information:

```
MOST TIME CONSUMING KERNELS IN EACH ITERATION (EXPECTED TO BE SAME KERNEL):
KERNEL NAME: miopenSp3AsmConv_v21_1_2_gfx9_fp32_stride1        448
KERNEL NAME: miopenSp3AsmConv_v21_1_2_gfx9_fp32_stride1        450
KERNEL NAME: miopenSp3AsmConv_v21_1_2_gfx9_fp32_stride1        449
KERNEL NAME: miopenSp3AsmConv_v21_1_2_gfx9_fp32_stride1        451
KERNEL NAME: miopenSp3AsmConv_v21_1_2_gfx9_fp32_stride1        456
```

The output provides `SUM`, `MIN`, `MAX` and `COUNT` information for each kernel executed for a given model. It also provides the average total time. The following three files are provided for reference:

- OUTPUT CSV FILE: Provides a summary of the run which includes utilized MIGraphX knobs and related kernel timing information.
- KERNEL TIMING DETAILS: Provides the hotspot kernel timing information.
- ALL DATA FROM ALL RUNS: Provides all output data related to all iterations executed during a run.

Parsing an existing JSON file:

```
python roctx.py --parse --json-path ../trace.json
```

5.11 Environment Variables

5.11.1 For parsing

MIGRAPHX_TRACE_ONNX_PARSER

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints debugging traces for the ONNX parser. Prints: Initializers (if used), ONNX node operators, added MIGraphX instructions.

MIGRAPHX_DISABLE_FP16_INSTANCENORM_CONVERT

Set to "1", "enable", "enabled", "yes", or "true" to use. Disables the conversion from fp16 to fp32 for the `InstanceNormalization` ONNX operator that MIGX does as a workaround for accuracy issues with *reduce_mean/variance*. See `parse_instancenorm.cpp` for more details.

5.11.2 Matchers

MIGRAPHX_TRACE_MATCHES

Set to "1" to print the matcher that matches an instruction and the matched instruction. Set to "2" and use the `MIGRAPHX_TRACE_MATCHES_FOR` flag to filter out results.

MIGRAPHX_TRACE_MATCHES_FOR

Set to the name of any matcher to print the traces for that matcher only.

MIGRAPHX_VALIDATE_MATCHES

Set to "1", "enable", "enabled", "yes", or "true" to use. Validates the module after finding the matches (runs `module.validate()`).

5.11.3 Program Execution

MIGRAPHX_TRACE_EVAL

Set to "1", "2", or "3" to use. "1" prints the instruction run and the time taken. "2" prints everything in "1" and a snippet of the output argument and some output statistics (e.g. min, max, mean). "3" prints everything in "1" and all output buffers.

5.11.4 Program Verification

MIGRAPHX_VERIFY_ENABLE_ALLCLOSE

Set to "1", "enable", "enabled", "yes", or "true" to use. Uses `allclose` with the given `atol` and `rtol` for verifying ranges with `driver verify` or the tests that use `migraphx/verify.hpp`.

5.11.5 Pass debugging or Pass controls

MIGRAPHX_TRACE_ELIMINATE_CONTIGUOUS

Set to "1", "enable", "enabled", "yes", or "true" to use. Debug print the instructions that have input `contiguous` instructions removed.

MIGRAPHX_DISABLE_POINTWISE_FUSION

Set to "1", "enable", "enabled", "yes", or "true" to use. Disables the `fuse_pointwise` compile pass.

MIGRAPHX_DEBUG_MEMORY_COLORING

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints debug statements for the `memory_coloring` pass.

MIGRAPHX_TRACE_SCHEDULE

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints debug statements for the `schedule` pass.

MIGRAPHX_TRACE_PROPAGATE_CONSTANT

Set to "1", "enable", "enabled", "yes", or "true" to use. Traces instructions replaced with a constant.

MIGRAPHX_8BITS_QUANTIZATION_PARAMS

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints the quantization parameters in the main module only.

MIGRAPHX_DISABLE_DNNL_POST_OPS_WORKAROUND

Set to "1", "enable", "enabled", "yes", or "true" to use. Disables the DNNL post ops workaround.

MIGRAPHX_DISABLE_MIOPEN_FUSION

Set to "1", "enable", "enabled", "yes", or "true" to use. Disables MIOpen fusions.

MIGRAPHX_DISABLE_SCHEDULE_PASS

Set to "1", "enable", "enabled", "yes", or "true" to use. Disables the `schedule` pass.

MIGRAPHX_DISABLE_REDUCE_FUSION

Set to "1", "enable", "enabled", "yes", or "true" to use. Disables the `fuse_reduce` pass.

MIGRAPHX_ENABLE_NHWC

Set to "1", "enable", "enabled", "yes", or "true" to use. Enables the `layout_nhwc` pass.

MIGRAPHX_ENABLE_CK

Set to "1", "enable", "enabled", "yes", or "true" to use. Enables use of the Composable Kernels library. Use it in conjunction with `MIGRAPHX_DISABLE_MLIR=1`.

MIGRAPHX_DISABLE_MLIR*

Set to "1", "enable", "enabled", "yes", or "true" to use. Disables use of the rocMLIR library.

MIGRAPHX_ENABLE_EXTRA_MLIR

Set to "1", "enable", "enabled", "yes", or "true" to use. Enables additional opportunities to use MLIR for chances of an improved performance.

MIGRAPHX_COPY_LITERALS

Set to "1", "enable", "enabled", "yes", or "true" to use. Uses `hip_copy_to_gpu` with a new `literal` instruction rather than using `hip_copy_literal{}`.

5.11.6 Compilation traces

`MIGRAPHX_TRACE_FINALIZE`

Set to "1", "enable", "enabled", "yes", or "true" to use. Debug print instructions during the `module.finalize()` step.

`MIGRAPHX_TRACE_COMPILE`

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints trace information for the graph compilation process.

`MIGRAPHX_TRACE_PASSES`

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints the compile pass and the program after the pass.

`MIGRAPHX_TIME_PASSES`

Set to "1", "enable", "enabled", "yes", or "true" to use. Times the compile passes.

5.11.7 GPU kernels JIT compilation debugging

These environment variables are applicable for both hiprtc and hipclang.

`MIGRAPHX_TRACE_CMD_EXECUTE`

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints commands executed by the MIGraphX `process`.

`MIGRAPHX_TRACE_HIPRTC`

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints HIPRTC options and C++ file executed.

`MIGRAPHX_DEBUG_SAVE_TEMP_DIR`

Set to "1", "enable", "enabled", "yes", or "true" to use. Prevents deletion of the created temporary directories.

`MIGRAPHX_GPU_DEBUG`

Set to "1", "enable", "enabled", "yes", or "true" to use. Internally, this adds the option `-DMIGRAPHX_DEBUG` when compiling GPU kernels. It enables assertions and capture of source locations for the errors.

`MIGRAPHX_GPU_DEBUG_SYM`

Set to "1", "enable", "enabled", "yes", or "true" to use. Adds the option `-g` when compiling HIPRTC.

`MIGRAPHX_GPU_DUMP_SRC`

Set to "1", "enable", "enabled", "yes", or "true" to use. Dumps the compiled HIPRTC source files.

`MIGRAPHX_GPU_DUMP_ASM`

Set to "1", "enable", "enabled", "yes", or "true" to use. Dumps the hip-clang assembly.

`MIGRAPHX_GPU_OPTIMIZE`

Set the optimization mode for GPU compile (`-O` option). Defaults to `-O3`.

`MIGRAPHX_GPU_COMPILE_PARALLEL`

Set to the number of threads to use. Compiles GPU code in parallel with the given number of threads.

`MIGRAPHX_TRACE_NARY`

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints the `nary` device functions used.

MIGRAPHX_ENABLE_HIPRTC_WORKAROUNDS

Set to "1", "enable", "enabled", "yes", or "true" to use. Enables HIPRTC workarounds for bugs in HIPRTC.

MIGRAPHX_USE_FAST_SOFTMAX

Set to "1", "enable", "enabled", "yes", or "true" to use. Uses fast softmax optimization.

MIGRAPHX_ENABLE_NULL_STREAM

Set to "1", "enable", "enabled", "yes", or "true" to use. Allows using null stream for miopen and hipStream.

MIGRAPHX_NSTREAMS

Set to the number of streams to use. Defaults to 1.

MIGRAPHX_TRACE_BENCHMARKING

Set to "1" to print benchmarking trace. Set to "2" to print detailed benchmarking trace.

5.11.8 MLIR vars

MIGRAPHX_TRACE_MLIR

Set to "1" to trace MLIR and print any failures. Set to "2" to additionally print all MLIR operations.

MIGRAPHX_MLIR_USE_SPECIFIC_OPS

Set to the MLIR operations you want to always use regardless of the GPU architecture. Accepts a list of operators separated by commas (e.g. "fused", "convolution", "dot").

MIGRAPHX_MLIR_TUNING_DB

Set to the path of the MLIR tuning database to load.

MIGRAPHX_MLIR_TUNING_CFG

Set to the path of the tuning configuration. Appends to tuning cfg file that could be used with rocMLIR tuning scripts.

MIGRAPHX_MLIR_TUNE_EXHAUSTIVE

Set to "1", "enable", "enabled", "yes", or "true" to use. Performs exhaustive tuning for MLIR.

MIGRAPHX_MLIR_TUNE_LIMIT

Set to an integer greater than 1. Limits the number of solutions available to MLIR for tuning.

5.11.9 CK vars

MIGRAPHX_LOG_CK_GEMM

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints composable kernels GEMM traces.

MIGRAPHX_CK_DEBUG

Set to "1", "enable", "enabled", "yes", or "true" to use. Mandatorily adds `-DMIGRAPHX_CK_CHECK=1` for compiling composable kernel operators.

MIGRAPHX_TUNE_CK

Set to "1", "enable", "enabled", "yes", or "true" to use. Performs tuning for composable kernels.

5.11.10 Testing

`MIGRAPHX_TRACE_TEST_COMPILE`

Set to the target whose compilation you want to trace (e.g. "gpu", "cpu"). Prints the compile trace for verify tests on the given target. Don't use this flag in conjunction with `MIGRAPHX_TRACE_COMPILE`.

`MIGRAPHX_TRACE_TEST`

Set to "1", "enable", "enabled", "yes", or "true" to use. Prints the reference and target programs even if the verify tests pass.

`MIGRAPHX_DUMP_TEST`

Set to "1", "enable", "enabled", "yes", or "true" to use. Dumps verify tests to `.mxr` files.

CHAPTER

SIX

C++ API REFERENCE

6.1 shape

enum **migraphx_shape_datatype_t**

An enum to represent the different data type inputs.

Values:

enumerator **migraphx_shape_tuple_type**

enumerator **migraphx_shape_bool_type**

enumerator **migraphx_shape_half_type**

enumerator **migraphx_shape_float_type**

enumerator **migraphx_shape_double_type**

enumerator **migraphx_shape_uint8_type**

enumerator **migraphx_shape_int8_type**

enumerator **migraphx_shape_uint16_type**

enumerator **migraphx_shape_int16_type**

enumerator **migraphx_shape_int32_type**

enumerator **migraphx_shape_int64_type**

enumerator **migraphx_shape_uint32_type**

enumerator **migraphx_shape_uint64_type** enumerator **migraphx_shape_fp8e4m3fnuz_type**

template<class **Lens**, class **Strides**>

struct **shape** : public migraphx::handle_base<>

Describe shape of tensor.

A shape consists of a data type, lengths of multi-dimension tensor, and strides

Public Types

using **shape_type** = *shape*

using **index_array** = typename *Lens*::base_array

Public Functions

inline **shape**()

inline **shape**(const migraphx_shape *p)

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **shape**(*HandleType* *p, *Lifetime* lifetime)

inline **shape**(*migraphx_shape_datatype_t* type)

Construct a scalar shape.

inline **shape**(*migraphx_shape_datatype_t* type, std::vector<size_t> plengths)

Construct a shape with its type and lengths. The strides are automatically computed assumming a packed layout.

inline **shape**(*migraphx_shape_datatype_t* t, std::initializer_list<std::size_t> d)

inline **shape**(*migraphx_shape_datatype_t* type, std::vector<size_t> plengths, std::vector<size_t> pstrides)

inline **shape**(*migraphx_shape_datatype_t* type, const dynamic_dimensions &dyn_dims)

inline std::vector<size_t> **lengths**() const

inline std::vector<size_t> **strides**() const

inline dynamic_dimensions **dyn_dims**() const

Get the dynamic dimensions of the shape.

inline *migraphx_shape_datatype_t* **type**() const

inline size_t **elements**() const

inline size_t **bytes**() const

inline bool **standard**() const

inline bool **dynamic**() const

Is the shape dynamic.

inline size_t **index**(size_t i) const

constexpr **shape**() = default

inline constexpr **shape**(*Lens* l, *Strides* s)

inline constexpr auto **elements**() const

inline constexpr auto **element_space**() const

inline constexpr auto **packed**() const

inline constexpr auto **broadcasted**() const

inline constexpr auto **transposed**() const

inline constexpr auto **skips**() const

inline constexpr auto **standard**() const

inline constexpr index_int **index**(*index_array* x) const

inline constexpr index_int **index**(index_int i) const

inline constexpr index_int **compute_index**(index_int i) const

inline constexpr *index_array* **multi**(index_int idx) const

Convert single index into a multi-index.

inline constexpr index_int **single**(*index_array* idx) const

Convert multi-index into a single index.

inline constexpr *shape* **get_shape**() const

Public Members

Lens **lens** = {}

Strides **strides** = {}

Friends

inline friend bool **operator==**(const *shape* &px, const *shape* &py)

inline friend bool **operator!=**(const *shape* &px, const *shape* &py)

template<class **Stream**>
inline friend constexpr const *Stream* &**operator<<**(const *Stream* &ss, const *shape* &s)

6.2 argument

struct **argument** : public migraphx::handle_base<>

Arguments to be passed to an migraphx arguments.

An `argument` represents a raw buffer of data with a shape.

Public Functions

inline **argument**()

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **argument**(*HandleType* *p, *Lifetime* lifetime)

inline **argument**(const migraphx_argument *p)

inline **argument**(*shape* pshape)

inline **argument**(*shape* pshape, void *pbuffer)

inline *shape* **get_shape**() const

inline char ***data**() const

template<typename **T**>
inline std::vector<*T*> **as_vector**() const

Public Static Functions

static inline *argument* **generate**(*shape* ps, size_t pseed = 0)

Generate an argument using random data.

Friends

inline friend bool **operator==**(const *argument* &px, const *argument* &py)

inline friend bool **operator!=**(const *argument* &px, const *argument* &py)

6.3 target

struct **target** : public migraphx::handle_base<>

A target for compilation.

Public Functions

inline **target**()

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **target**(*HandleType* *p, *Lifetime* lifetime)

inline **target**(const char *name)

Construct a target from its name.

6.4 program

struct **program_parameter_shapes** : public migraphx::handle_base<>

Public Functions

inline **program_parameter_shapes**()

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **program_parameter_shapes**(*HandleType* *p, *Lifetime* lifetime)

inline size_t **size**() const

inline *shape* **operator[]**(const char *pname) const

inline std::vector<const char*> **names**() const

struct **program_parameters** : public migraphx::handle_base<>

A class to construct the inputs parameters for a program.

Public Functions

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **program_parameters**(*HandleType* *p, *Lifetime* lifetime)

inline **program_parameters**(migraphx_program_parameters *p)

inline **program_parameters**()

inline **program_parameters**(std::initializer_list<std::pair<std::string, *argument*>> l)

Construct the parameters from initializer_list.

inline void **add**(const char *pname, const *argument* &pargument) const

Add a new parameter.

struct **migraphx_compile_options**

Public Functions

template<class ...**Ts**>
inline **migraphx_compile_options**(*Ts*&&... xs)

Public Members

migraphx::compile_options **object**

struct **program** : public migraphx::handle_base<>

A program represents the all computation graphs to be compiled and executed.

Public Functions

inline **program**()

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **program**(*HandleType* *p, *Lifetime* lifetime)

inline void **compile**(const *target* &ptarget, const compile_options &poptions) const

Compile the program for a specific target to be ran on.

inline void **compile**(const *target* &ptarget) const

Compile the program for a specific target to be ran on.

inline *program_parameter_shapes* **get_parameter_shapes**() const

Return the shapes for the input parameters.

inline shapes **get_output_shapes**() const

Get the shapes of all the outputs returned by this program.

inline arguments **eval**(const *program_parameters* &pparams) const

Run the program using the inputs passed in.

template<class **Stream**>
inline arguments **run_async**(const *program_parameters* &pparams, *Stream* *s) const

Overloaded to allow for execution_environment input.

inline void **print**() const

inline *program* **sort**()

inline module **get_main_module**()

inline context **experimental_get_context**()

inline module **create_module**(const std::string &name)

Friends

inline friend bool **operator==**(const *program* &px, const *program* &py)

inline friend bool **operator!=**(const *program* &px, const *program* &py)

6.5 quantize

struct **quantize_op_names** : public migraphx::handle_base<>

Public Functions

inline **quantize_op_names**()

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **quantize_op_names**(*HandleType* *p, *Lifetime* lifetime)

inline void **add**(const std::string &name)

inline void migraphx::**quantize_fp16**(const *program* &prog)

Quantize program to use fp16.

inline void migraphx::**quantize_fp16**(const *program* &prog, const *quantize_op_names* &names)

Quantize program to use fp16.

struct **quantize_int8_options** : public migraphx::handle_base<>

Options to be passed when quantizing for int8.

Public Functions

inline **quantize_int8_options**()

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **quantize_int8_options**(*HandleType* *p, *Lifetime* lifetime)

inline void **add_op_name**(const std::string &name)

Add an operator that should be quantized.

inline void **add_calibration_data**(const *program_parameters* &pp)

Add calibrartion data to be used for quantizing.

Public Members

std::vector<parameter_map> **calibration** = {}

std::unordered_set<std::string> **op_names** = {}

6.6 parse_onnx

struct **onnx_options** : public migraphx::handle_base<>

Options for parsing onnx options.

Public Functions

inline **onnx_options**()

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **onnx_options**(*HandleType* *p, *Lifetime* lifetime)

inline void **set_input_parameter_shape**(const std::string &name, std::vector<std::size_t> dim)

Make onnx parser treat an inputs with a certain dimensions.

inline void **set_dyn_input_parameter_shape**(const std::string &name, const dynamic_dimensions &dyn_dims)

inline void **set_default_dim_value**(unsigned int value)

When there is a dimension parameter, then use this default value.

inline void **set_default_dyn_dim_value**(const dynamic_dimension &dd)

inline void **set_default_loop_iterations**(int64_t value)

Set default max iteration number for the loop operator.

inline void **set_limit_loop_iterations**(int64_t value)

Set max iteration limit for the loop operator.

inline *program* migraphx::**parse_onnx**(const char *filename)

Parse an onnx file into a migraphx program.

inline *program* migraphx::**parse_onnx**(const char *filename, const migraphx::*onnx_options* &options)

Parse an onnx file into a migraphx program.

inline *program* migraphx::**parse_onnx_buffer**(const std::string &buffer)

Parse a buffer of memory as an onnx file.

inline *program* migraphx::**parse_onnx_buffer**(const std::string &buffer, const migraphx::*onnx_options* &options)

Parse a buffer of memory as an onnx file.

inline *program* migraphx::**parse_onnx_buffer**(const void *data, size_t size)

Parse a buffer of memory as an onnx file.

inline *program* `migraphx::parse_onnx_buffer`(const void *data, size_t size, const migraphx::*onnx_options* &options)

Parse a buffer of memory as an onnx file.

6.7 load

struct **file_options** : public migraphx::handle_base<>

Public Functions

template<class **HandleType**, class **Lifetime**, class = typename std::enable_if<std::is_convertible<*HandleType**, handle_type*>{}>::type>
inline **file_options**(*HandleType* *p, *Lifetime* lifetime)

inline **file_options**()

inline void **set_file_format**(const char *format)

inline *program* `migraphx::load`(const char *filename)

Load a saved migraphx program from a file.

inline *program* `migraphx::load`(const char *filename, const *file_options* &options)

Load a saved migraphx program from a file.

6.8 save

inline void `migraphx::save`(const *program* &p, const char *filename)

Save a program to a file.

inline void `migraphx::save`(const *program* &p, const char *filename, const *file_options* &options)

Save a program to a file.

CHAPTER

SEVEN

PYTHON API REFERENCE

7.1 shape

class migraphx.**shape**(*type*, *lens*, *strides=None*, *dyn_dims*)

Describes the shape of a tensor. This includes size, layout, and data type. Use dyn_dims for a dynamic shape.

migraphx.**type**()

An integer that represents the type.

Return type

int

migraphx.**lens**()

A list of the lengths of the shape.

Return type

list[int]

migraphx.**strides**()

A list of the strides of the shape.

Return type

list[int]

migraphx.**elements**()

The number of elements in the shape.

Return type

int

migraphx.**dyn_dims**()

The dynamic dimensions of the shape.

Return type

list[*dynamic_dimension*]

migraphx.**bytes**()

The number of bytes the shape uses.

Return type

int

migraphx.**type_size**()

The number of bytes one element uses

Return type
int

`migraphx.ndim()`

The number of dimensions for the shape.

Return type
int

`migraphx.packed()`

Returns true if the shape is packed.

Return type
bool

`migraphx.transposed()`

Returns true if the shape is transposed.

Return type
bool

`migraphx.broadcasted()`

Returns true if the shape is broadcasted.

Return type
bool

`migraphx.dynamic()`

Returns true if the shape is dynamic.

Return type
bool

`migraphx.standard()`

Returns true if the shape is a standard shape. That is, the shape is both packed and not transposed.

Return type
bool

`migraphx.scalar()`

Returns true if all strides are equal to 0 (scalar tensor).

Return type
bool

7.2 dynamic_dimension

`class migraphx.dynamic_dimension(`*min*, *max*, *optimals*`)`

Constructs a *dynamic_dimension* from a minimum, a maximum, and optionally a set of optimals.

`migraphx.is_fixed()`

Returns true if the *dynamic_dimension* is fixed.

:rtype : int

7.3 argument

`class migraphx.argument(data)`

Constructs an argument from a python buffer. This can include numpy arrays.

`migraphx.data_ptr()`

Returns the address to the underlying argument data.

Return type
int

`migraphx.get_shape()`

Returns the shape of the argument.

Return type
shape

`migraphx.tolist()`

Converts the elements of the argument to a python list.

Return type
list

`migraphx.generate_argument(s, seed=0)`

Generates an argument with random data.

Parameters

- **s** (shape) – Shape of argument to generate.
- **seed** (*int*) – The seed used for random number generation.

Return type
argument

`migraphx.fill_argument(s, value)`

Fills argument of shape *s* with the given value.

Parameters

- **s** (shape) – Shape of argument to fill.
- **value** (*int*) – Value to fill in the argument.

Return type
argument

`migraphx.create_argument(s, values)`

Creates an argument of shape *s* with a set of values.

Parameters

- **s** (shape) – Shape of argument to create.
- **values** (*list*) – Values to put in the argument. Must be the same number of elements as the shape.

Return type
argument

`migraphx.argument_from_pointer(shape, address)`

Creates argument from data stored in given address without copy.

Parameters

- **shape** (shape) – Shape of the data stored in address.
- **address** (*long*) – Memory address of data from another source

Return type

argument

7.4 target

`class migraphx.target`

This represents the compilation target.

`migraphx.get_target(name)`

Constructs the target.

Parameters

name (*str*) – The name of the target to construct. This can either be 'gpu' or 'ref'.

Return type

target

7.5 module

`migraphx.print()`

Prints the contents of the module as list of instructions.

`migraphx.add_instruction(op, args, mod_args=[])`

Adds instruction into the module.

Parameters

- **op** (*operation*) – 'migraphx.op' to be added as instruction.
- **args** (*list[instruction]*) – list of inputs to the op.
- **mod_args** (*list[module]*) – optional list of module arguments to the operator.

:rtype instruction

`migraphx.add_literal(data)`

Adds constant or literal data of provided shape into the module from python buffer which includes numpy array.

Parameters

data (*py::buffer*) – Python buffer or numpy array

:rtype instruction

`migraphx.add_parameter(name, shape)`

Adds a parameter to the module with the provided name and shape.

Parameters

- **name** (*str*) – name of the parameter.

- **shape** (shape) – shape of the parameter.

:rtype instruction

`migraphx.add_return(args)`

Adds a return instruction into the module.

Parameters

args (*list[instruction]*) – instruction arguments which need to be returned from the module.

:rtype instruction

7.6 program

`class migraphx.program`

Represents the computation graph to be compiled and run.

`migraphx.clone()`

Makes a copy of the program.

Return type

program

`migraphx.get_parameter_names()`

Gets all the input argument's or parameter's names to the program as a list.

:rtype list[str]

`migraphx.get_parameter_shapes()`

Gets the shapes of all the input parameters in the program.

Return type

dict[str, *shape*]

`migraphx.get_output_shapes()`

Gets the shapes of the final outputs of the program.

Return type

list[*shape*]

`migraphx.compile(t, offload_copy=True, fast_math=True, exhaustive_tune=False)`

Compiles the program for the target and optimizes it.

Parameters

- **t** (target) – Compilation target for the program.
- **offload_copy** (*bool*) – For targets with offloaded memory(such as the gpu), this will insert instructions during compilation to copy the input parameters to the offloaded memory and to copy the final result from the offloaded memory back to main memory.
- **fast_math** (*bool*) – Optimize math functions to use faster approximate versions. There may be slight accuracy degredation when enabled.
- **exhaustive_tune** – Flag to enable exhaustive search to find the fastest version of generated kernels for selected backend.

migraphx.**get_main_module**()

Gets main module of the program.

:rtype module

migraphx.**create_module**(*name*)

Creates and adds a module with the provided name into the program.

:param str name : name of the new module. :rtype module

migraphx.**run**(*params*)

Runs the program.

Parameters
params (*dict[str, argument]*) – Map of the input parameters to be used when running the program.

Returns
The result of the last instruction.

Return type
list[*argument*]

migraphx.**sort**()

Sorts the modules of the program for the instructions to appear in topologically sorted order.

migraphx.**quantize_fp16**(*prog*, *ins_names=['all']*)

Quantizes the program to use fp16.

Parameters

- **prog** (program) – Program to quantize.
- **ins_names** (*list[str]*) – List of instructions to quantize.

migraphx.**quantize_int8**(*prog*, *t*, *calibration=[]*, *ins_names=['dot', 'convolution']*)

Quantizes the program to use int8.

Parameters

- **prog** (program) – Program to quantize.
- **t** (target) – Target to be used to run the calibration data.
- **calibration** (*list[dict[str, argument]]*) – Calibration data used to decide the parameters to the int8 optimization.
- **ins_names** (*list[str]*) – List of instructions to quantize.

7.7 op

7.8 parse_onnx

migraphx.**parse_onnx**(*filename*, *default_dim_value=1*, *map_input_dims={}*, *skip_unknown_operators=false*, *print_program_on_error=false*, *max_loop_iterations=10*, *limit_max_iterations=65535*)

Loads and parses an ONNX file.

Parameters

- **filename** (*str*) – Path to file.

- **default_dim_value** (*str*) – default dimension to use (if not specified in onnx file).
- **default_dyn_dim_value** (dynamic_dimension) – default dynamic_dimension value to use.
- **map_input_dims** (*str*) – Explicitly specify the dims of an input.
- **map_dyn_input_dims** (*list*[dynamic_dimension]) – Explicitly specify the dynamic_dimensions of an input.
- **skip_unknown_operators** (*str*) – Continue parsing onnx file if an unknown operator is found.
- **print_program_on_error** (*str*) – Print program if an error occurs.
- **max_loop_iterations** (*int*) – Maximum iteration number for the loop operator if trip count is not set.
- **limit_max_iterations** (*int*) – Maximum iteration limit for the loop operator.

Return type
program

7.9 parse_tf

migraphx.**parse_tf**(*filename*, *is_nhwc=True*, *batch_size=1*, *map_input_dims=dict()*, *output_names=[]*)

Loads and parses a tensorflow protobuf file.

Parameters

- **filename** (*str*) – Path to file.
- **is_nhwc** (*bool*) – Use nhwc as default format.
- **batch_size** (*str*) – default batch size to use (if not specified in protobuf).
- **map_input_dims** (*dict[str, list[int]]*) – Optional arg to explictly specify dimensions of the inputs.
- **output_names** (*list[str]*) – Optional argument specify names of the output nodes.

Return type
program

7.10 load

migraphx.**load**(*filename*, *format='msgpack'*)

Loads a MIGraphX program.

Parameters

- **filename** (*str*) – Path to file.
- **format** (*str*) – Format of file. Valid options are msgpack or json.

Return type
program

7.11 save

`migraphx.save(p, filename, format='msgpack')`

Saves a MIGraphX program.

Parameters

- **p** (program) – Program to save.
- **filename** (*str*) – Path to file.
- **format** (*str*) – Format of file. Valid options are msgpack or json.

CHAPTER

EIGHT

LICENSE

The MIT License (MIT)

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

Q

R

S

T