Glow

Machine Learning Compiler
for Hardware Accelerators

Glow Is A Compiler

Compilers can handle multiple hardware targets.

Compilers optimize intermediate representations.

Compilers can vectorize and fuse loops.

Compilers knew how to do all of these things in the 80s!

Design Principles

Use well known compiler techniques to solve engineering problems.

Support a wide range of hardware acceleration platforms.

Make it easy to develop new hardware targets.

Enable powerful optimizations.

What You Get With Glow

Compiler framework that takes you most of the way from PyTorch to the hardware.

Reusable set of components, such as scheduler, memory allocator, JIT and code generator.

Conformance test suite and reference implementation to validate the stack.

Utilities such as quantizer and profiler.

Execution engine that enables multi-device support.

Glow Open Source Process

Glow is developed in open source on GitHub.

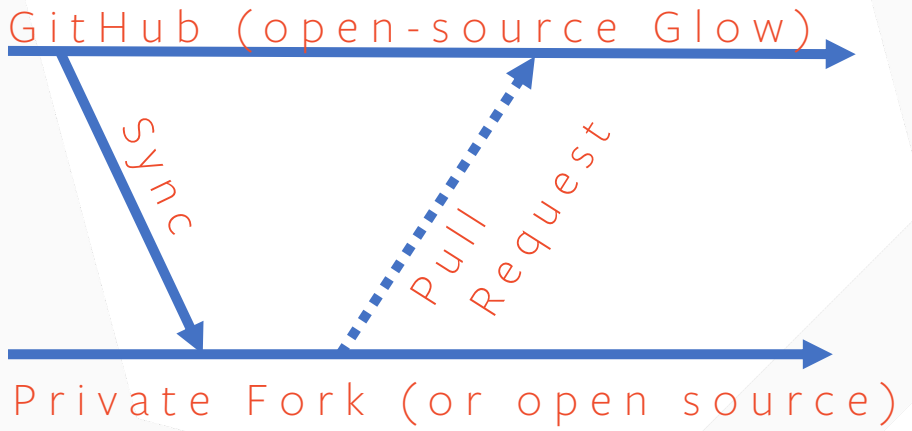
Vendors can develop their backend in a private fork.

Download and sync upstream changes.

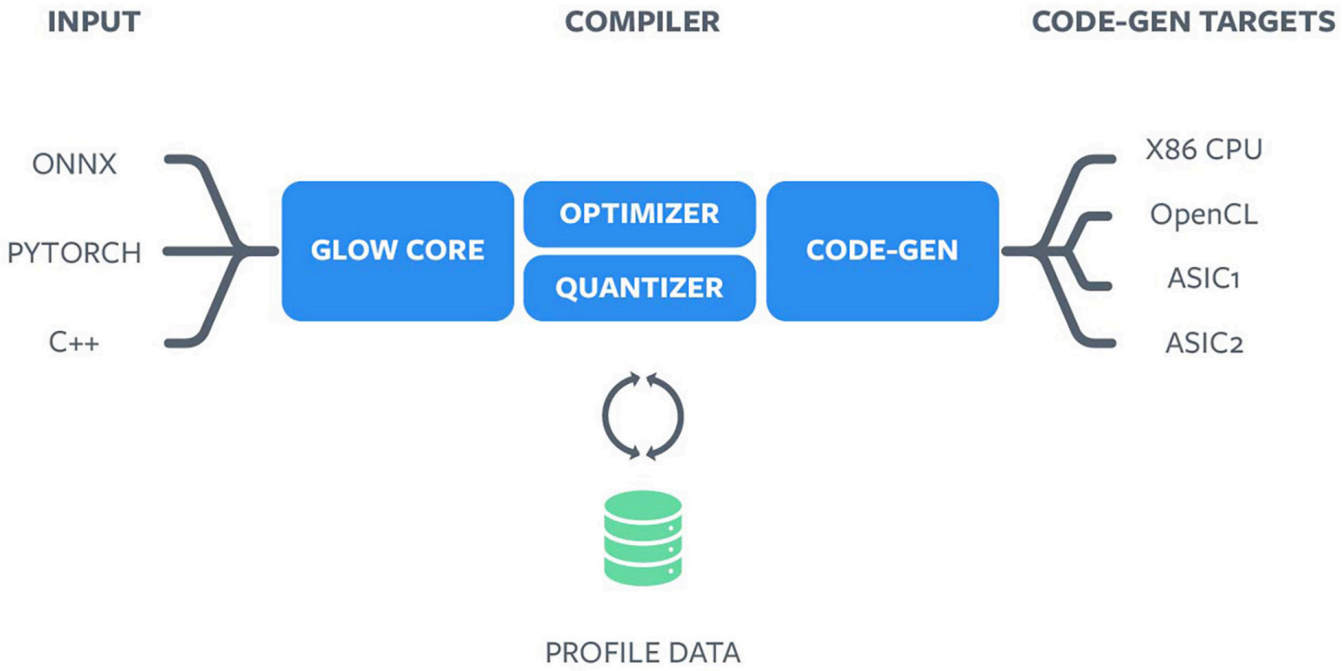
Can submit features and bug fixes with pull requests.

Open source PRs are reviewed by the community.

Vendors generally will own updates to their backends and if open-sourced, will manage governance as well.

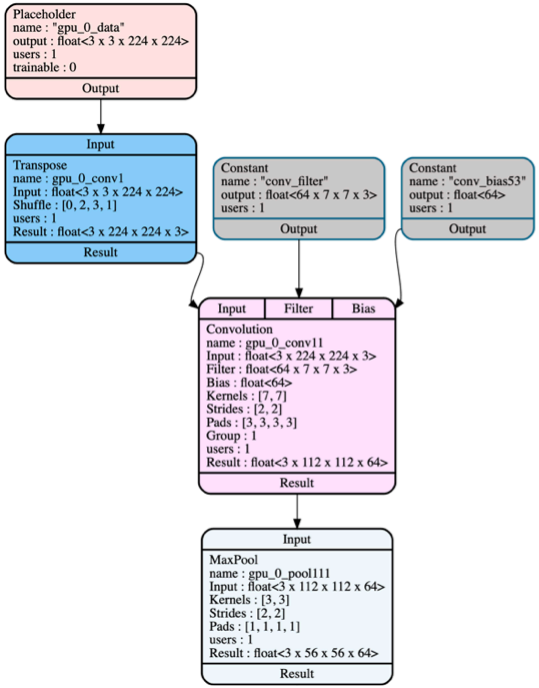


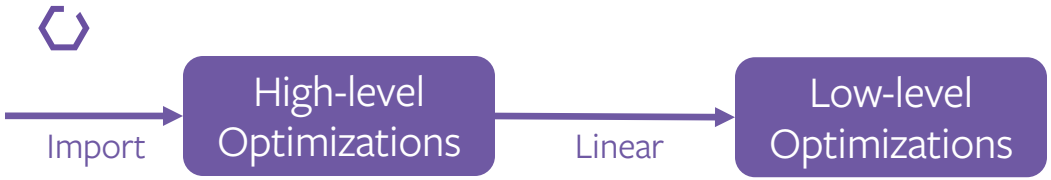
Glow Compiler & Runtime

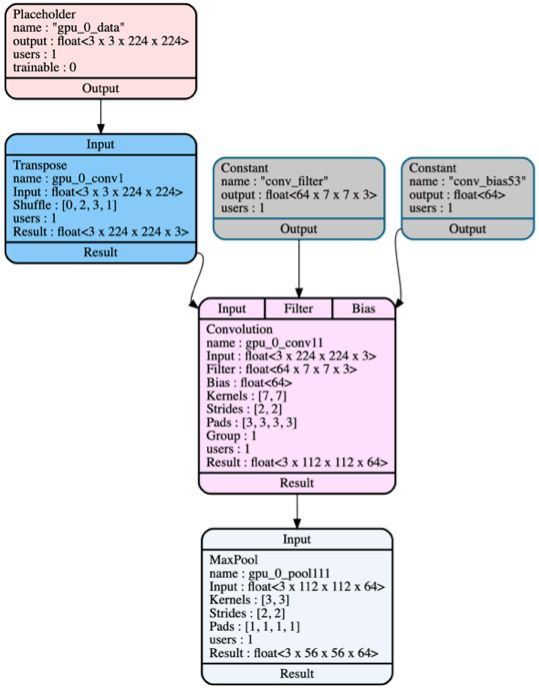


Import

High-level Optimizations

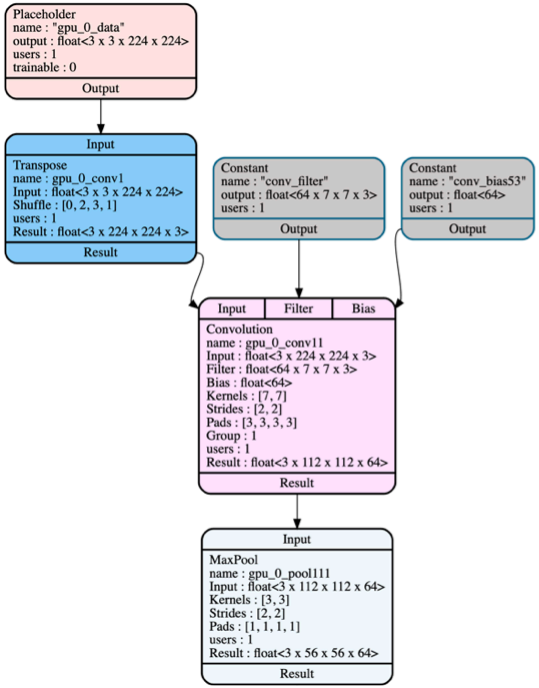







```
declare {
 %input = weight float<8 x 28 x 28 x 1>, broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>, xavier, 25.0
 %filter0 = weight float<16>, broadcast, 0.100
 %weights = weight float<10 x 144>, xavier, 144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo, @in %input,
     @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in %allo0,
     @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```




```
declare {
 %input = weight float<8 x 28 x 28 x 1>, broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>, xavier, 25.0
 %filter0 = weight float<16>, broadcast, 0.100
 %weights = weight float<10 x 144>, xavier, 144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo, @in %input,
     @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in %allo0,
     @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

```
LBB14_1:
  vmovaps 3211264(%rcx,%rax,4), %ymm1
  vmovaps 3211296(%rcx,%rax,4), %ymm2
  vmovaps 3211328(%rcx,%rax,4), %ymm3
  vaddps 6422528(%rcx,%rax,4), %ymm1, %ymm1
  vaddps 6422560(%rcx,%rax,4), %ymm2, %ymm2
  vmovaps 3211360(%rcx,%rax,4), %ymm4
  vaddps 6422592(%rcx,%rax,4), %ymm3, %ymm3
  vaddps 6422624(%rcx,%rax,4), %ymm4, %ymm4
  vmaxps %ymm0, %ymm1, %ymm1
  vmaxps %ymm0, %ymm2, %ymm2
  vmaxps %ymm0, %ymm3, %ymm3
  vmovaps %ymm1, 6422528(%rcx,%rax,4)
  vmovaps %ymm2, 6422560(%rcx,%rax,4)
  vmaxps %ymm0, %ymm4, %ymm1
  vmovaps %ymm3, 6422592(%rcx,%rax,4)
  vmovaps %ymm1, 6422624(%rcx,%rax,4)
  addq $32, %rax
```

Automatic IR Generation

- Graph Nodes and Instructions are classes with many methods: ctor, hash, set/get, compare, clone, print, verification, etc.
- Instead of writing the methods in C++ we auto-generate them
- Annotate with Properties

Automatic IR Generation

- Graph Nodes and Instructions are classes with many methods: ctor, hash, set/get, compare, clone, print, verification, etc.
- Instead of writing the methods in C++ we auto-generate them
- Annotate with Properties

```
BB.newNode("Convolution")
    .addInput("Input")
    .addInput("Filter")
    .addInput("Bias")
    .addMember(MemberType::VectorUnsigned, "Kernels")
    .addMember(MemberType::VectorUnsigned, "Strides")
    .addMember(MemberType::VectorUnsigned, "Pads")
    .addMember(MemberType::Unsigned, "Group")
    .addResultFromCtorArg()
    .addGradient()
    .setDocstring("Performs 2D Convolution using a given Input, Filter, and "
                  "Bias tensors, as well as provided Kernels, Strides, Pads, "
                  "and Group.");
```

Automatic IR Generation

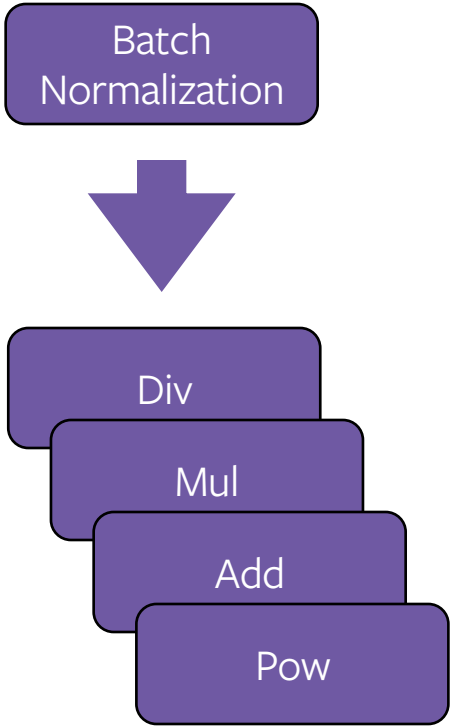
- Graph Nodes and Instructions are classes with many methods: ctor, hash, set/get, compare, clone, print, verification, etc.
- Instead of writing the methods in C++ we auto-generate them
- Annotate with Properties

```
BB.newNode("Convolution")
    .addInput("Input")
    .addInput("Filter")
    .addInput("Bias")
    .addMember(MemberType::VectorUnsigned, "Kernels")
    .addMember(MemberType::VectorUnsigned, "Strides")
    .addMember(MemberType::VectorUnsigned, "Pads")
    .addMember(MemberType::Unsigned, "Group")
    .addResultFromCtorArg()
    .addGradient()
    .setDocstring("Performs 2D Convolution using a given Input, Filter, and "
                  "Bias tensors, as well as provided Kernels, Strides, Pads, "
                  "and Group.");
```

```
BB.newInstr("Tanh")
    .addOperand("Dest", OperandKind::Out)
    .addOperand("Src", OperandKind::In)
    .inplaceOperand({
        "Dest",
        "Src",
    })
    .dataParallel()
    .autoVerify(VerifyKind::SameElementType, {"Dest", "Src"})
    .autoVerify(VerifyKind::SameShape, {"Dest", "Src"})
    .autoIRGen();
```

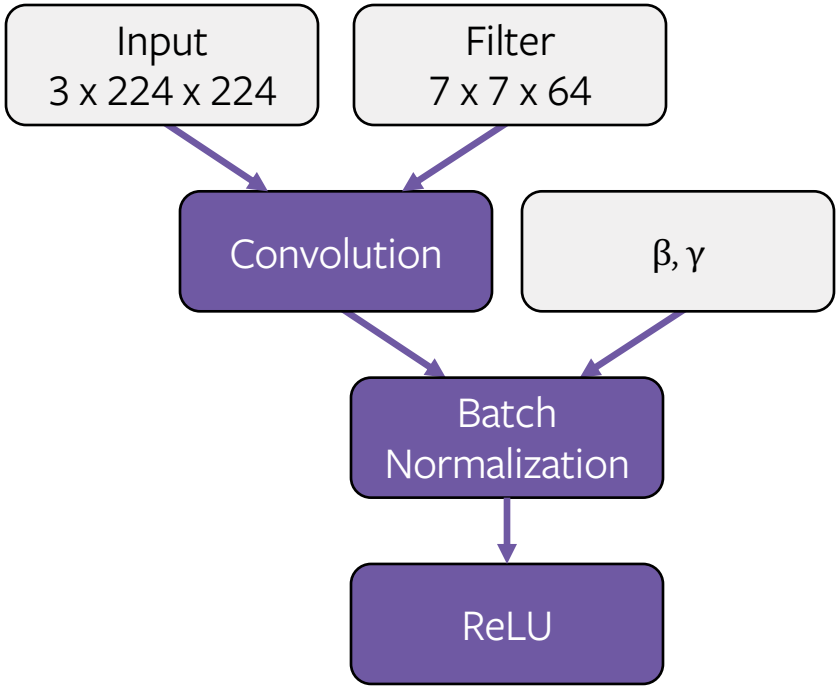
Graph Lowering

- ML frameworks have 100s of operators
  - … times N accelerator backends
  - … implemented in custom assembly
- Not scalable
- Instead, Lower complex ops to simple kernels

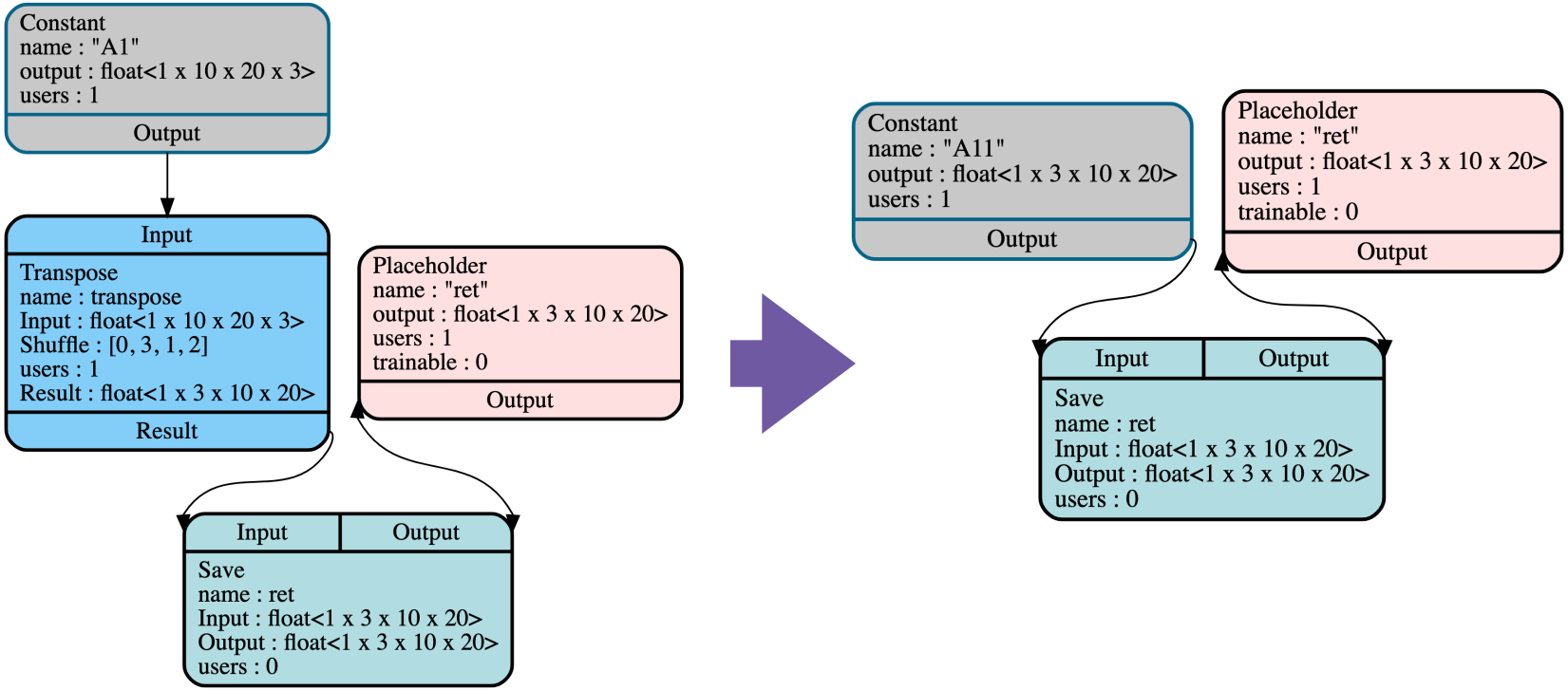


Graph Optimizations

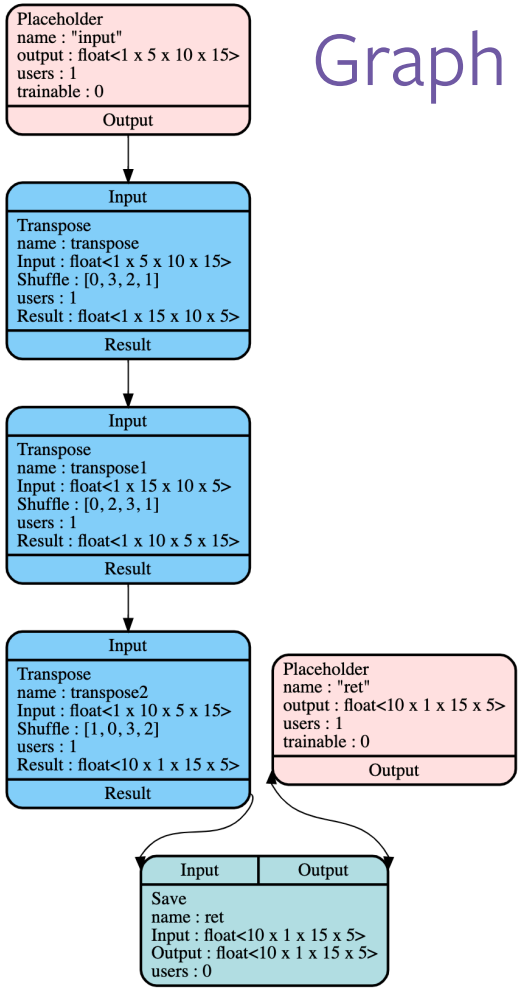
- Pure Data-flow Graph
- Static Types
- Domain-specific Optimizations
  - Fuse BatchNorm + Conv
  - Transpose Elimination
- Classical Compiler Optimizations
  - Dead Code Elimination
  - Common Subexpression Elimination
  - Constant Propagation

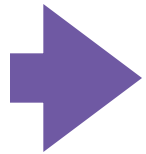


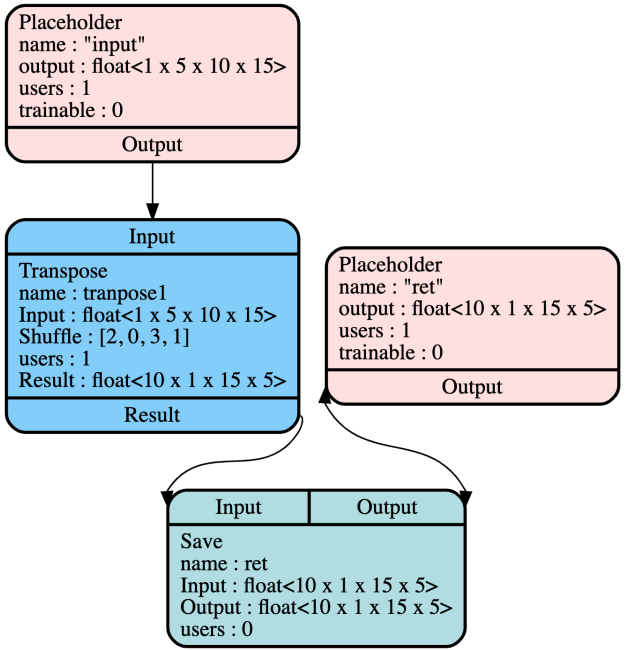
Graph Optimizations



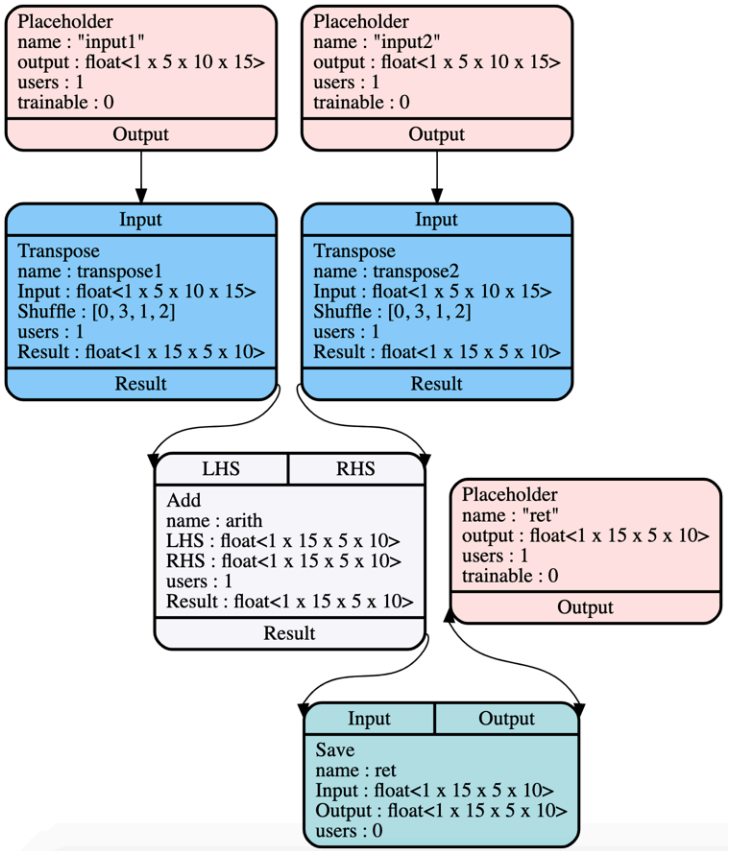
Graph Optimizations

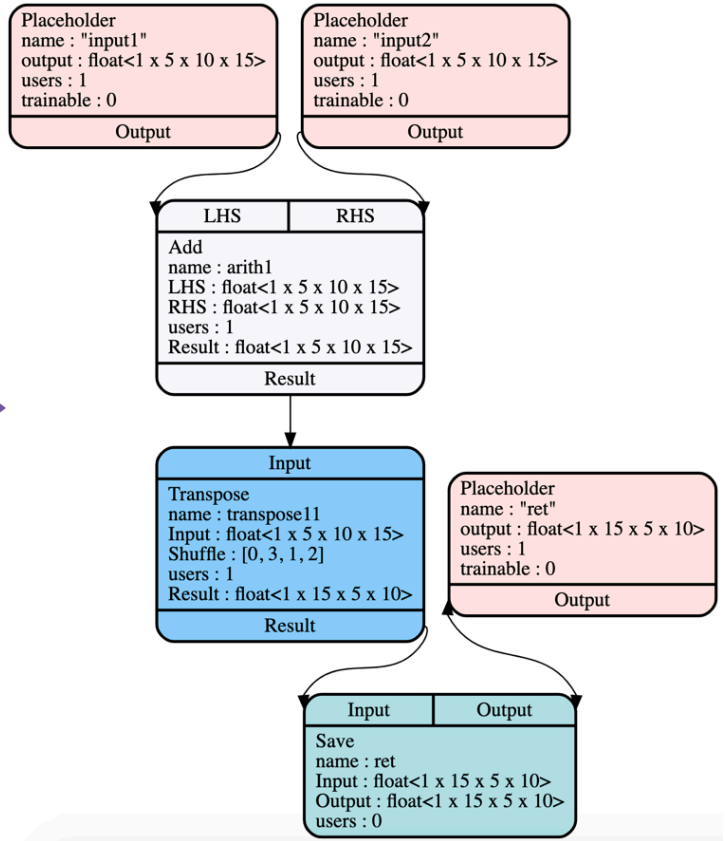




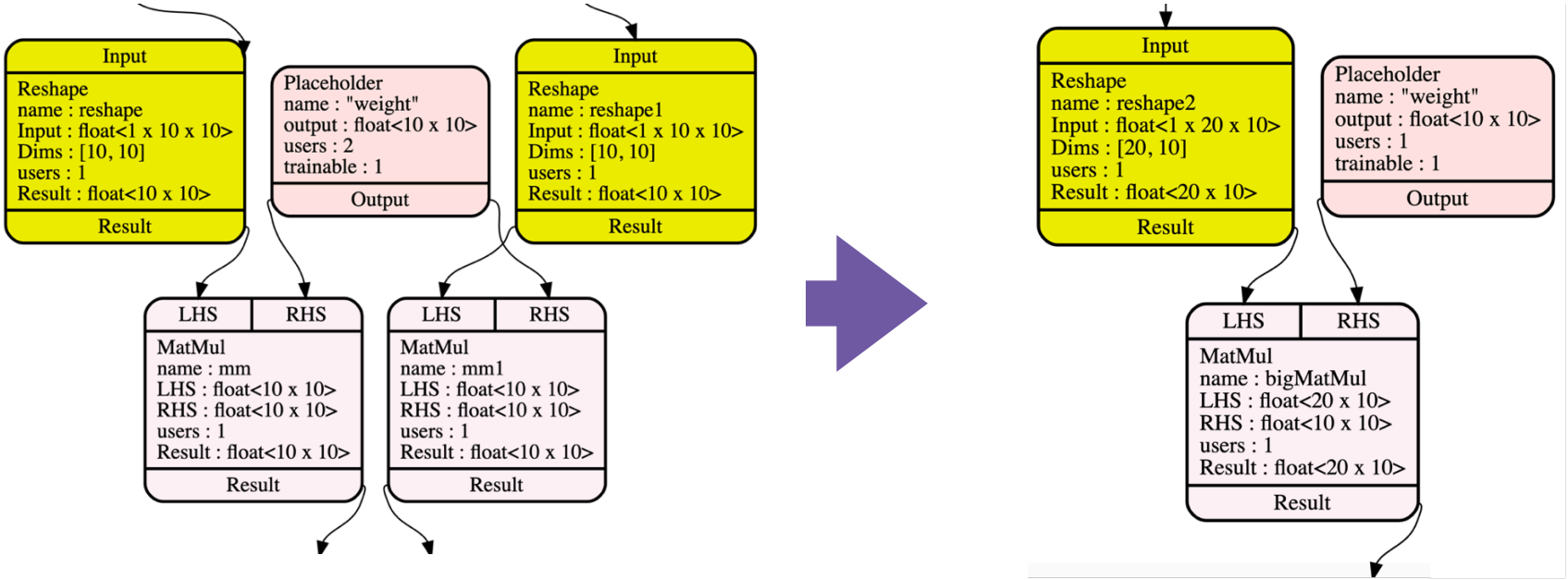


Graph Optimizations





Graph Optimizations



Implementing a Backend

All backends:

- `isOpSupported()`
- `shouldLower()`
- `transformPostLowering()`
- `compile()`
- `DeviceManager`

Instruction-specific:

- `generateInst()`
- `shouldShareBuffers()`

Implementing a Backend

All backends:

- isOpSupported()
- shouldLower()
- transformPostLowering()
- compile()
- DeviceManager

Instruction-specific:

- generateInst()
- shouldShareBuffers()

```
bool Interpreter::isOpSupported(const NodeInfo &NI) const {
  switch (NI.getKind()) {
  case Kinded::Kind::AddNodeKind:
  case Kinded::Kind::SubNodeKind:
  case Kinded::Kind::MulNodeKind:
  case Kinded::Kind::MaxNodeKind:
  case Kinded::Kind::MinNodeKind:
  case Kinded::Kind::AvgPoolNodeKind:
  case Kinded::Kind::MaxPoolNodeKind:
  case Kinded::Kind::MatMulNodeKind:
  case Kinded::Kind::BatchedReduceAddNodeKind:
  case Kinded::Kind::LogNodeKind:
  case Kinded::Kind::TanhNodeKind:
  case Kinded::Kind::SigmoidNodeKind:
    return NI.allInputsAndOutputsHaveSameElemKind(
        {ElemKind::FloatTy, ElemKind::Float16Ty, ElemKind::Int8QTy});
```

Implementing a Backend

All backends:

- isOpSupported()
- shouldLower()
- transformPostLowering()
- compile()
- DeviceManager

```
bool Interpreter::shouldLower(const Node *N) const {
  if (N->getKind() == Kinded::Kind::ConvolutionNodeKind) {
    return false;
  }
  return true;
}
```

Instruction-specific:

- generateInst()
- shouldShareBuffers()

Implementing a Backend

All backends:

- isOpSupported()
- shouldLower()
- transformPostLowering()
- compile()
- DeviceManager

Instruction-specific:

- generateInst()
- shouldShareBuffers()

```
BB.newBackendSpecificNode("CPUConvDKKC8")
    .addInput("Input")
    .addInput("Filter")
    .addInput("Bias")
    .addMember(MemberType::VectorUnsigned, "Kernels")
    .addMember(MemberType::VectorUnsigned, "Strides")
    .addMember(MemberType::VectorUnsigned, "Pads")
    .addMember(MemberType::Unsigned, "Group")
    .addResultFromCtorArg()
    .setDocstring("This is a cpu-specific convolution"
                  "implementation where the filter is"
                  "transposed to the shape [D/8, K, K, C, 8]");
```

Implementing a Backend

All backends:

- `isOpSupported()`
- `shouldLower()`
- `transformPostLowering()`
- `compile()`
- `DeviceManager`

Instruction-specific:

- `generateInst()`
- `shouldShareBuffers()`

```
declare {
 %input = weight float<8 x 28 x 28 x 1>, broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>, xavier, 25.0
 %filter0 = weight float<16>, broadcast, 0.100
 %weights = weight float<10 x 144>, xavier, 144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo, @in %input,
     @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in %allo0,
     @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

- Linear instruction-based address-only representation
- Operands are typed pointers to buffers
- Memory optimizations:
  - Instruction scheduling
  - Buffer sharing
  - Shortening buffer lifetime

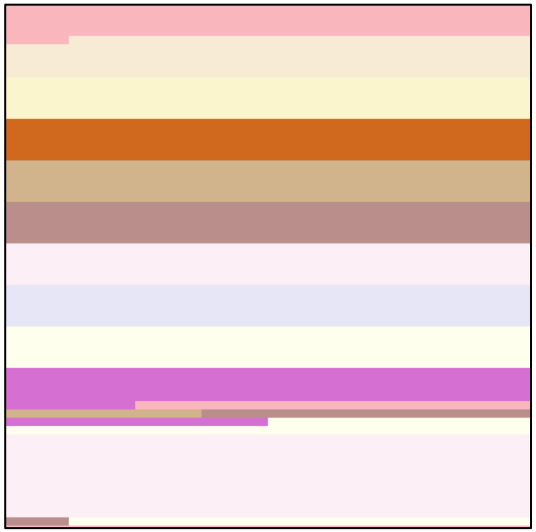
Implementing a Backend

All backends:

- `isOpSupported()`
- `shouldLower()`
- `transformPostLowering()`
- `compile()`
- `DeviceManager`

Instruction-specific:

- `generateInst()`
- `shouldShareBuffers()`



- Linear instruction-based address-only representation
- Operands are typed pointers to buffers
- Memory optimizations:
  - Instruction scheduling
  - Buffer sharing
  - Shortening buffer lifetime

CPU/JIT backend design

CPU/JIT backend design

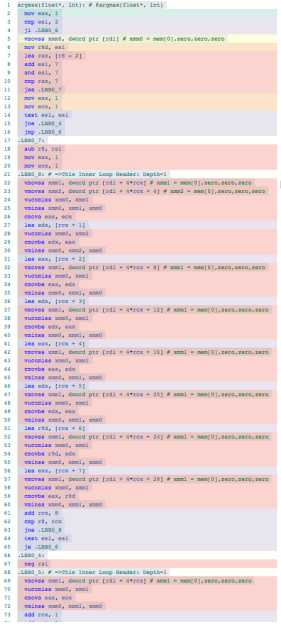
- CPU backend/JIT is LLVM-based
  - Leverages the LLVM's optimizer, code generator and ORC JIT APIs
  - Glow low-level IR is translated into LLVM IR and then LLVM backend produces optimized executable code
- A set of specialized optimized math kernels is needed for good performance
  - Generating such kernels manually by creating the LLVM IR is very time-consuming and error-prone
  - Instead, Glow uses a library of math kernels ("libjit") written in C and pre-compiled into LLVM bitcode
  - Easy to extend, saves a lot of time and effort

- Specializes library kernels for constant parameters
- Tensor shapes and many other parameters are compile-time constants
- Produces a cloned LLVM IR function w/ values of constant parameters substituted
- Specialization leads to much better performance

EXAMPLE OF AN OPERATION

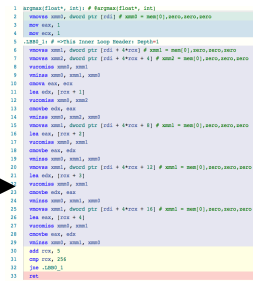
```
int argmax(float *arr, int n) {
  float maxVal = arr[0];
  int maxPos = 1;
  for (int i = 1; i < n; ++i) {
     if (arr[i] < maxVal) {
        maxVal = arr[i];
        maxPos = i;
     }
  }
  return maxPos;
}
```

UNSPECIALIZED CODE



SPECIALIZE FOR N = 256

SPECIALIZED CODE



✓Runtime checks are eliminated
✓Control flow is simplified
✓Smaller code
✓Faster execution

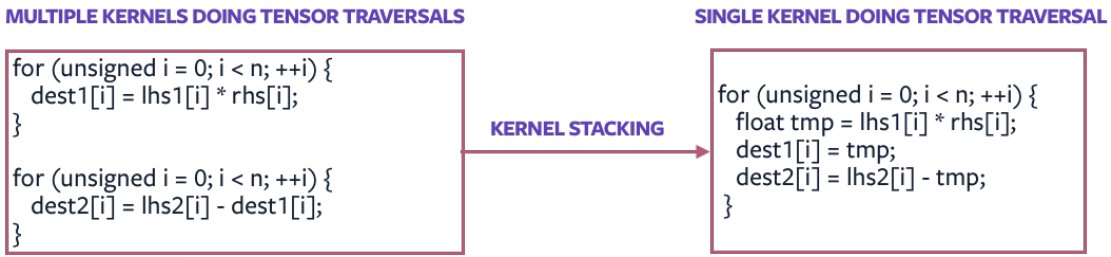
Stacking of kernels (vertical fusion)

- Many tensor operations are element-wise, e.g. add, max, mul
  - There are often chains of such operations like sub(z, mul(x, y))

- Sequential execution of these operations traverses the whole tensor every time and trashes the cache

- Instead, Glow generates LLVM IR for a stacked kernel where all those operations are performed on each element of a tensor during a single tensor traversal

```
BB.newInstr("Tanh")
    .addOperand("Dest", OperandKind::Out)
    .addOperand("Src", OperandKind::In)
    .inplaceOperand({
        "Dest",
        "Src",
    })
    .dataParallel()
    .autoVerify(VerifyKind::SameElementType, {"Dest", "Src"})
    .autoVerify(VerifyKind::SameShape, {"Dest", "Src"})
    .autoIRGen();
```



AOT and debugging support

- Ahead-of-time (AOT) compilation
  - Save the LLVM-generated machine code for a NN model as a self-contained object file
  - Deploy on mobile and memory-constrained devices

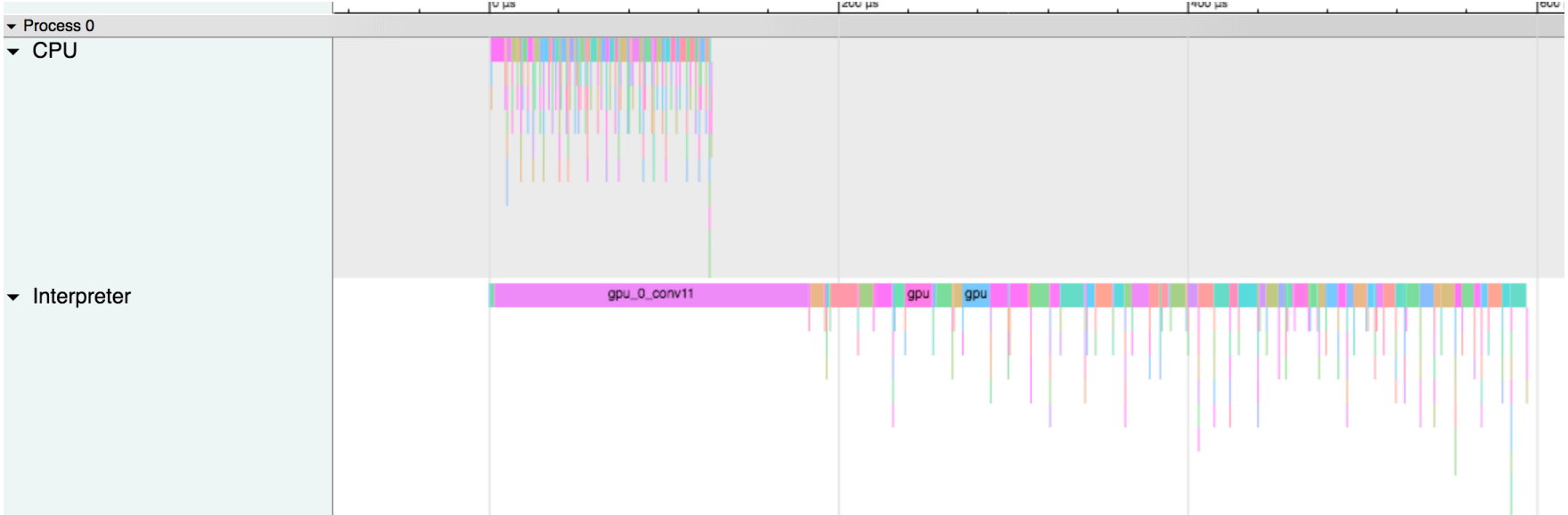
- Debugging support
  - Glow emits LLVM debug information for NN models
  - Debugging is done in terms of Glow IR instead of machine code

AN EXAMPLE OF A DEBUGGING SESSION USING LLDB

```
Process 95176 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason
    frame #0: 0x00000001000015f8 resnet50`resnet50 [inline
   282    162 %gpu_0/res4_4_branch2a__1024 = cpuconvdkkc8
 Kernel: 1, Stride: 1, Pad: 0, Depth: 256}
   283    163 %relu__1089 = cpumaxsplat @out %gpu_0_res4_4
   284    164 %gpu_0_res4_4_branch2b__1026_res = allocacti
-> 285    165 %gpu_0/res4_4_branch2b__1026 = cpuconvdkkc8
ernel: 3, Stride: 1, Pad: 1, Depth: 256}
   286    166 %dealloc37 = deallocactivation @out %gpu_0_r
   287    167 %relu__1090 = cpumaxsplat @out %gpu_0_res4_4
   288    168 %gpu_0_res4_4_branch2c_bn__786_res = allocac
, @in 173, @out 171
Target 0: (resnet50) stopped.
(lldb) p &gpu_0_res4_4_branch2a__1024_res
(float (*)[1][14][14][256]) $0 = 0x0000000100010060
(lldb) p &conv_filter__1025
(float (*)[32][3][3][256][8]) $1 = 0x00000001020f4210
(lldb) p conv_filter__1025[1][2][2][100][5]
(float) $2 = 0.15751867
(lldb) p conv_filter__1025[1][2][2][100]
(float [8]) $3 = ([0] = 0.032105349, [1] = -0.0109192021,
(lldb)
```

Trace Event Profiling

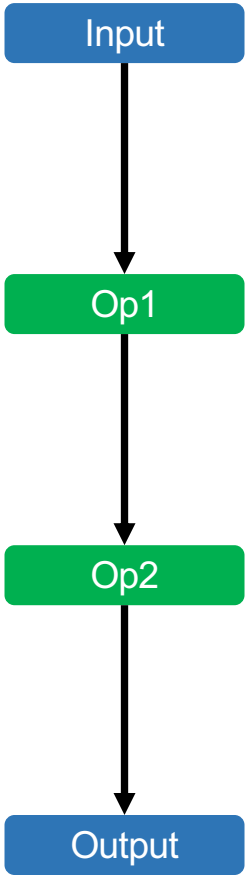
- **Timestamps** before and after operations
- Two options for tracing, **auto profile** all nodes or **manually insert trace events**
- End result is profiling data in the form of **Chrome trace events JSON** files



Profile-Guided Quantization Support

- **Problem**:
  - Not all inference hardware supports fp32 operations
  - Not all models are quantized
- Glow provides some simple **profile guided quantization** tools that can be used to quantize networks during initialization
- Supports int8 and int16 symmetric and asymmetric linear quantization
  - Float value = (qvalue – offset) * scale
- Very useful for **bringing up new unquantized models** on hardware requiring quantized ops

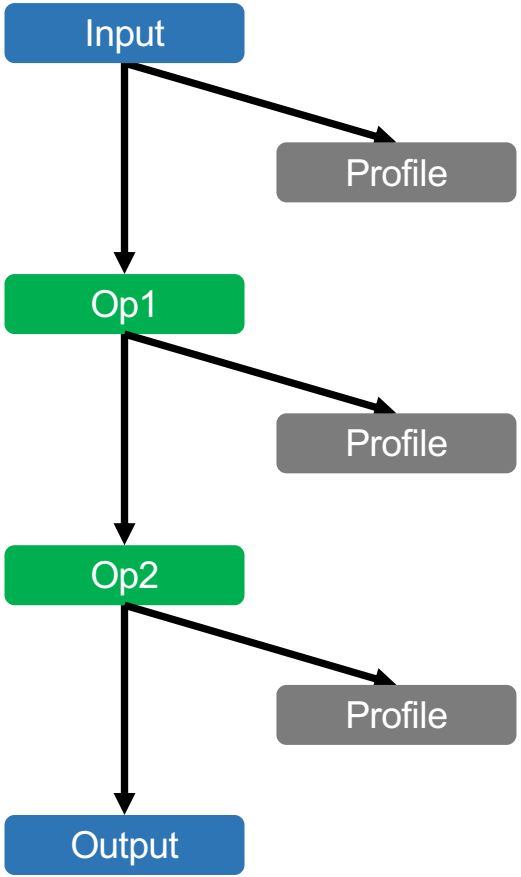
Profile-Guided Quantization Support



Profiling:

- 1: Add profiling operators to all intermediary values in the Glow graph
- 2: Run unquantized inference using the Glow interpreter reference Backend
- 3: Store profiles of intermediary values Glow graph

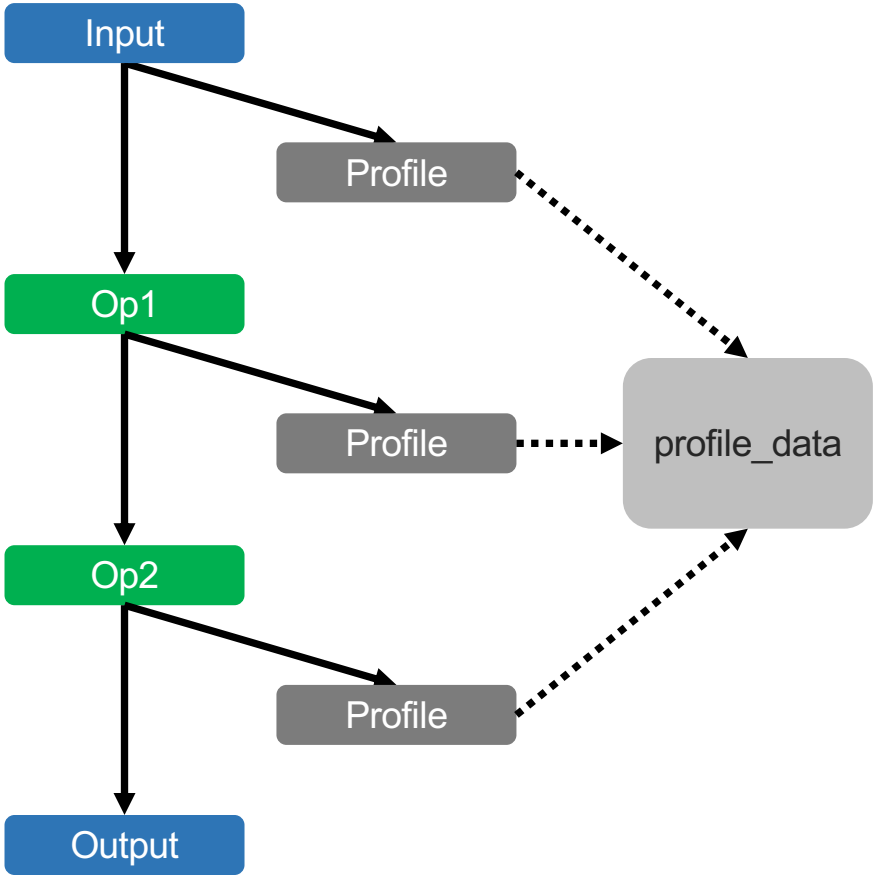
Profile-Guided Quantization Support



Profiling:

- 1: Add profiling operators to all intermediary values in the Glow graph
- 2: Run unquantized inference using the Glow interpreter reference Backend
- 3: Store profiles of intermediary values Glow graph

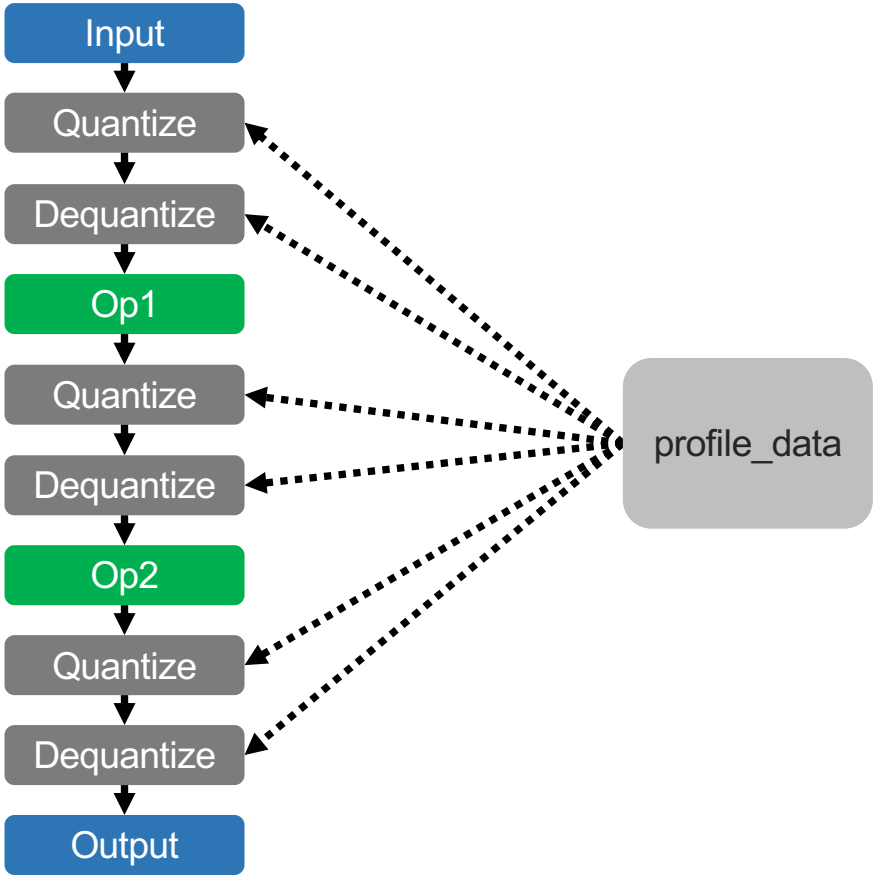
Profile-Guided Quantization Support



Profiling:

- 1: Add profiling operators to all intermediary values in the Glow graph
- 2: Run unquantized inference using the Glow interpreter reference Backend
- 3: Store profiles of intermediary values Glow graph

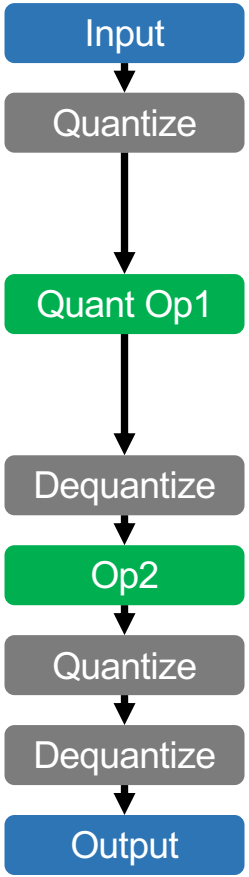
Profile-Guided Quantization Support



Quantization:

- 1: Add dequantization operators before and quantization operators after
- 2: Replace all ops with quantized ops where supported by the hardware

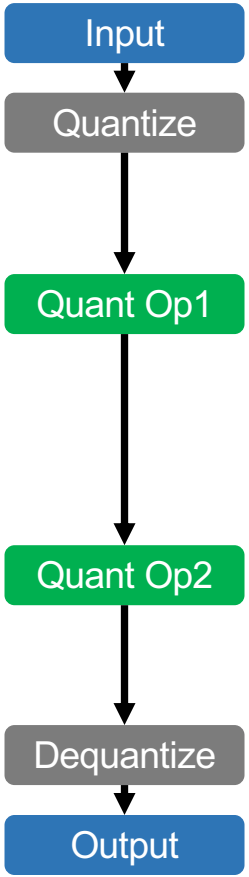
Profile-Guided Quantization Support



Quantization:

- 1: Add dequantization operators before and quantization operators after
- 2: Replace all ops with quantized ops where supported by the hardware

Profile-Guided Quantization Support



Quantization:

- 1: Add dequantization operators before and quantization operators after
- 2: Replace all ops with quantized ops where supported by the hardware

Thank you!

Participate on GitHub

Glow: Compiler for Neural Network Hardware Accelerators

https://github.com/pytorch/glow

arXiv publication

Glow: Graph Lowering Compiler Techniques for Neural Networks

https://arxiv.org/abs/1805.00907

@Scale 2018 Keynote

Glow: A community-driven approach to AI

https://atscaleconference.com/videos/scale-2018-keynote-glow-a-community-driven-approach-to-ai/

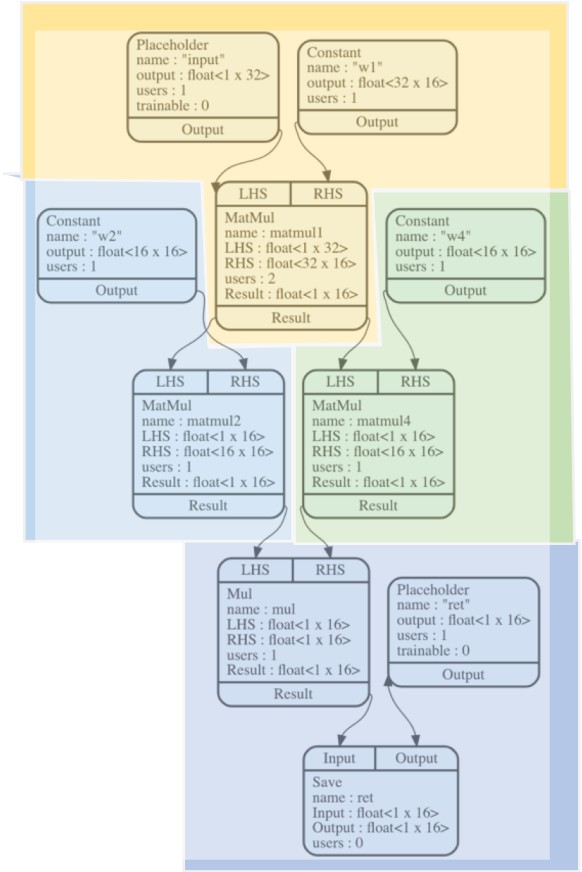
Backup Slides

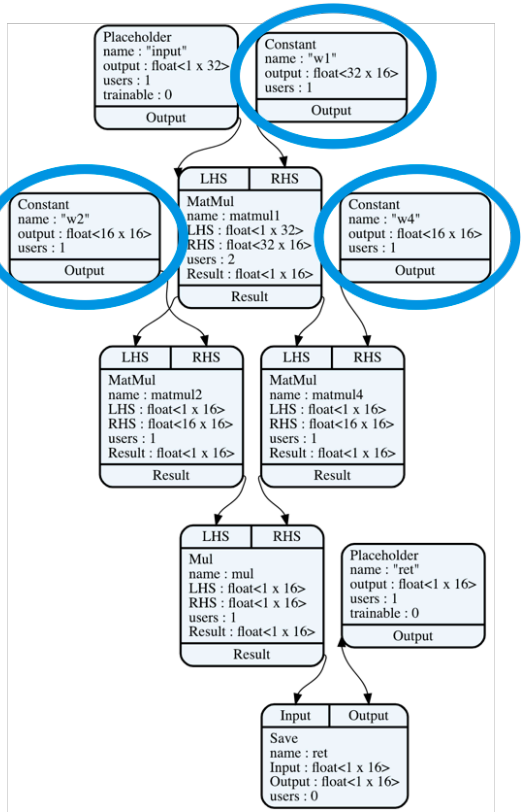
GLOW EXECUTION ENGINE

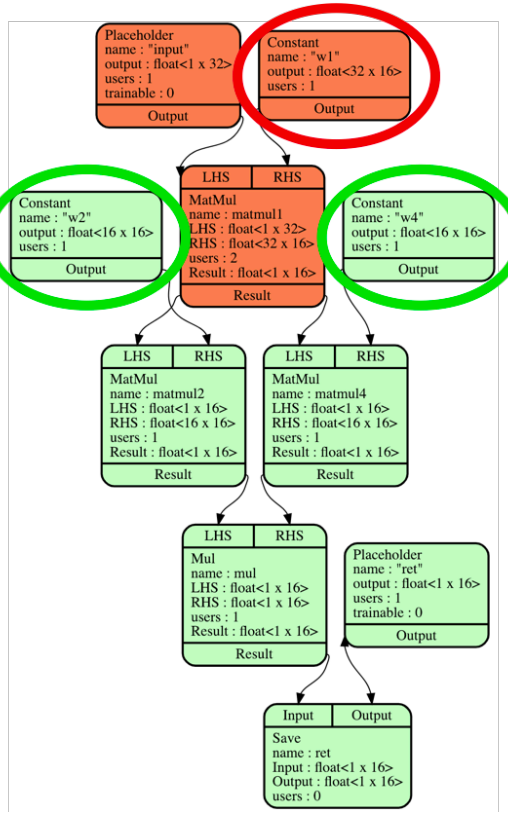
Partitioner

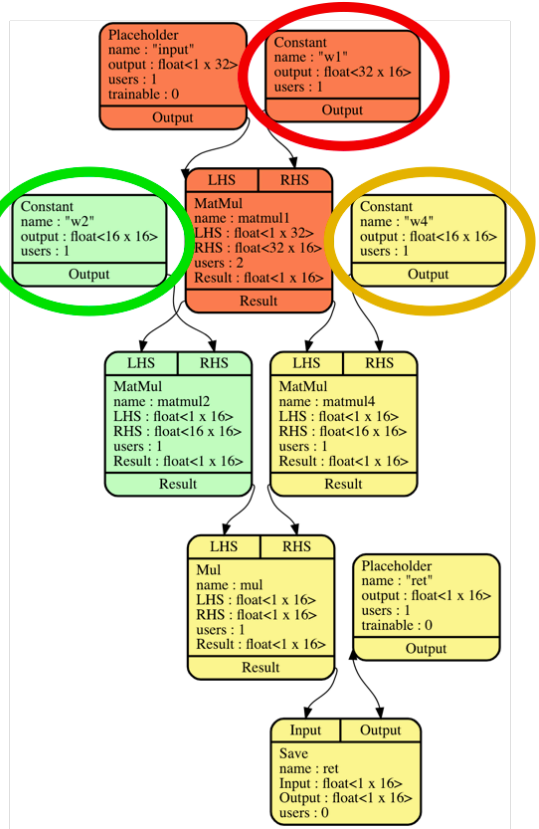
We need to support very large models (up to 80 gb) and want to saturate each accelerator, so we divide the input network into sub-graphs:

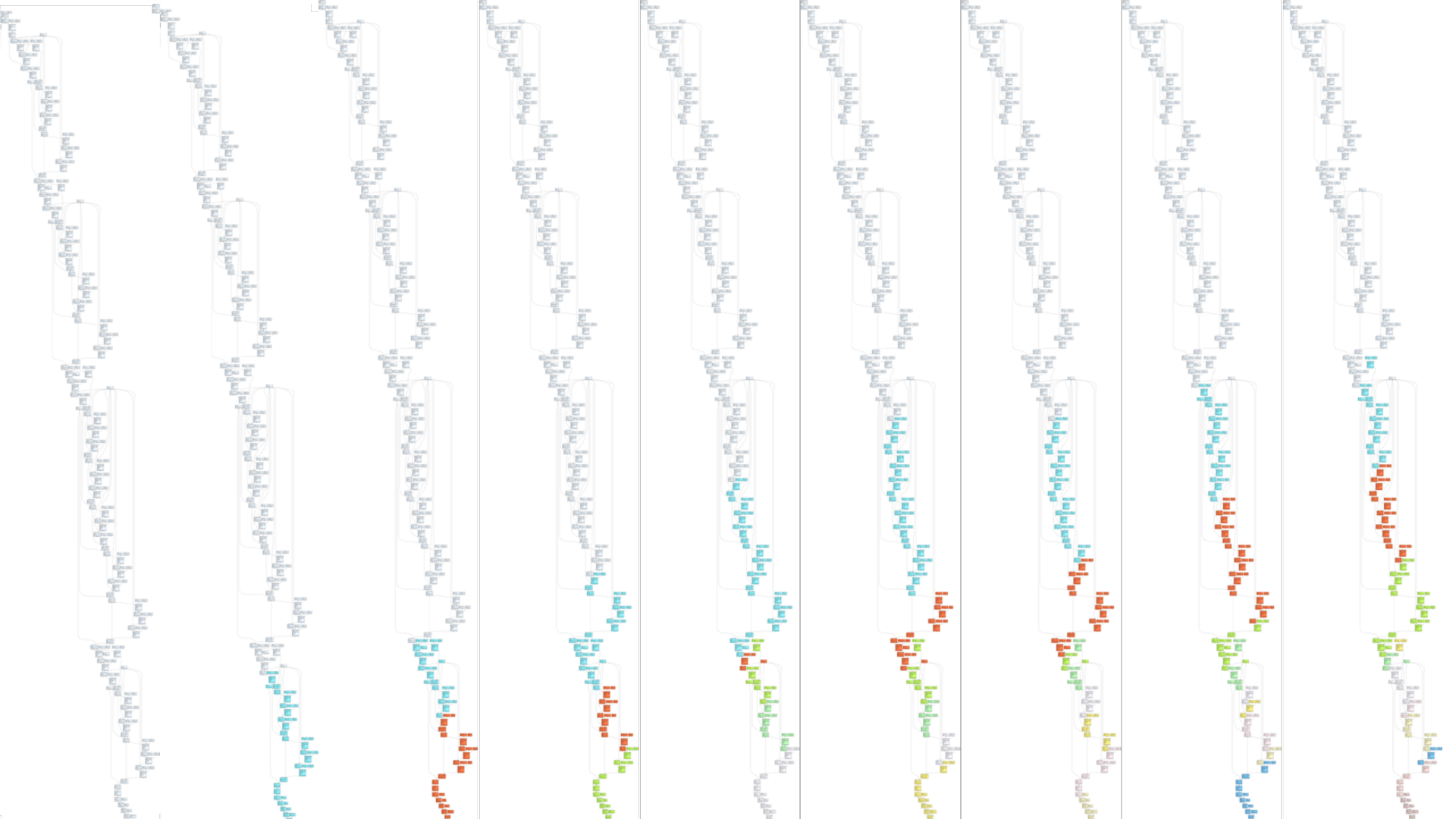
- **The Partitioner**:
  - Divides the graph into sub-graphs
  - Based on different criteria
    - Memory constraints
    - Estimated time cost
    - Communication cost between devices




Placeholder
name : "input"
output : float<1 x 32>
users : 1
trainable : 0
Output
Constant
name : "w1"
output : float<32 x 16>
users : 1
Output
Constant
name : "w2"
output : float<16 x 16>
users : 1
Output
LHS
RHS
MatMul
name : matmul1
LHS : float<1 x 32>
RHS : float<32 x 16>
users : 2
Result : float<1 x 16>
Result
Constant
name : "w4"
output : float<16 x 16>
users : 1
Output
LHS
RHS
MatMul
name : matmul2
LHS : float<1 x 16>
RHS : float<16 x 16>
users : 1
Result : float<1 x 16>
Result
LHS
RHS
MatMul
name : matmul4
LHS : float<1 x 16>
RHS : float<16 x 16>
users : 1
Result : float<1 x 16>
Result
LHS
RHS
Mul
name : mul
LHS : float<1 x 16>
RHS : float<1 x 16>
users : 1
Result : float<1 x 16>
Result
Placeholder
name : "ret"
output : float<1 x 16>
users : 1
trainable : 0
Output
Input
Output
Save
name : ret
Input : float<1 x 16>
Output : float<1 x 16>
users : 0


Placeholder
name : "input"
output : float<1 x 32>
users : 1
trainable : 0
Output
Constant
name : "w1"
output : float<32 x 16>
users : 1
Output
Constant
name : "w2"
output : float<16 x 16>
users : 1
Output
LHS
RHS
MatMul
name : matmul1
LHS : float<1 x 32>
RHS : float<32 x 16>
users : 2
Result : float<1 x 16>
Result
Constant
name : "w4"
output : float<16 x 16>
users : 1
Output
LHS
RHS
MatMul
name : matmul2
LHS : float<1 x 16>
RHS : float<16 x 16>
users : 1
Result : float<1 x 16>
Result
LHS
RHS
MatMul
name : matmul4
LHS : float<1 x 16>
RHS : float<16 x 16>
users : 1
Result : float<1 x 16>
Result
LHS
RHS
Mul
name : mul
LHS : float<1 x 16>
RHS : float<1 x 16>
users : 1
Result : float<1 x 16>
Result
Placeholder
name : "ret"
output : float<1 x 16>
users : 1
trainable : 0
Output
Input
Output
Save
name : ret
Input : float<1 x 16>
Output : float<1 x 16>
users : 0


Placeholder
name : "input"
output : float<1 x 32>
users : 1
trainable : 0
Output
Constant
name : "w1"
output : float<32 x 16>
users : 1
Output
Constant
name : "w2"
output : float<16 x 16>
users : 1
Output
LHS
RHS
MatMul
name : matmul1
LHS : float<1 x 32>
RHS : float<32 x 16>
users : 2
Result : float<1 x 16>
Result
Constant
name : "w4"
output : float<16 x 16>
users : 1
Output
LHS
RHS
MatMul
name : matmul2
LHS : float<1 x 16>
RHS : float<16 x 16>
users : 1
Result : float<1 x 16>
Result
LHS
RHS
MatMul
name : matmul4
LHS : float<1 x 16>
RHS : float<16 x 16>
users : 1
Result : float<1 x 16>
Result
LHS
RHS
Mul
name : mul
LHS : float<1 x 16>
RHS : float<1 x 16>
users : 1
Result : float<1 x 16>
Result
Placeholder
name : "ret"
output : float<1 x 16>
users : 1
trainable : 0
Output
Input
Output
Save
name : ret
Input : float<1 x 16>
Output : float<1 x 16>
users : 0



Static Shapes

- Glow and inference hardware need all tensor shapes upfront (**no dynamic allocation**)
- Any unbound shapes (like **batch size**) need to be bound before compiling
- Intermediary result shapes will be **inferred** by PyTorch at compile time

```
{10, 224, 224, 3}
```

N H W C

Function Families

- Glow's **solution to the constraints of static shapes**: Function Families
- **One network** can be provided to Glow via ONNXIFI with **multiple inputs of different size**
- Function family members are "**compiled together**" and passed together to backends
- Glow partitions the graph so with **memory constraints of the largest function** in mind
- Important that **Backends share constants** across networks to reduce memory footprint

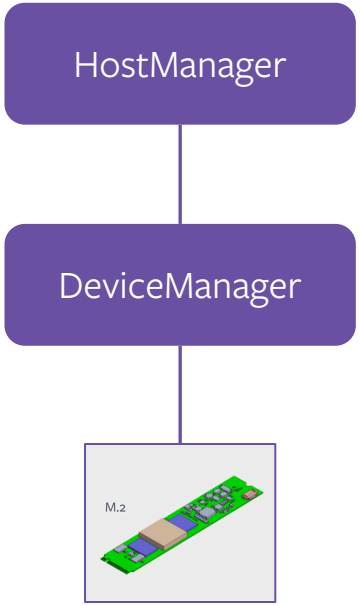
Implementing a Backend

All backends:

- `isOpSupported()`
- `shouldLower()`
- `transformPostLowering()`
- `compile()`
- `DeviceManager`

Instruction-specific:

- `generateInst()`
- `shouldShareBuffers()`



Implementing a Backend

All backends:

- isOpSupported()
- shouldLower()
- transformPostLowering()
- compile()
- DeviceManager

Instruction-specific:

- generateInst()
- shouldShareBuffers()

```
bool CPU::generateInst(Node *N, IRGenVisitor &irgen) const override {
  if (auto *N = llvm::dyn_cast<CPUConvDKKC8Node>(node)) {
    auto *I = valueForNode(N->getInput());
    auto *F = valueForNode(N->getFilter());
    auto *B = valueForNode(N->getBias());
    auto resTy = N->getResult().getType();
    auto *dest = builder_.createAllocActivationInst("alloc", resTy);
    auto *X =
        builder_.createCPUConvDKKC8Inst("N", dest, I, F, B,
                                        N->getKernels(), N->getStrides(),
                                        N->getPads(), N->getGroup());
    registerIR(N->getResult(), X->getDest());
    return true;
  }
  return false;
}
```

Implementing a Backend

All backends:

- isOpSupported()
- shouldLower()
- transformPostLowering()
- compile()
- DeviceManager

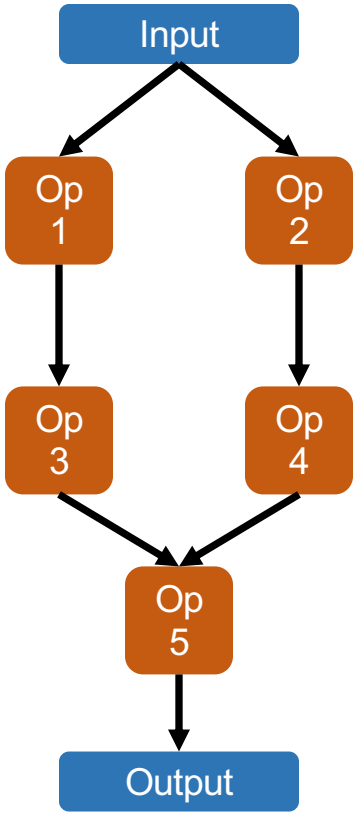
Instruction-specific:

- generateInst()
- shouldShareBuffers()

```
BB.newBackendSpecificInstr("CPUConvDKKC8")
  .addOperand("Dest", OperandKind::Out)
  .addOperand("Src", OperandKind::In)
  .addOperand("Filter", OperandKind::In)
  .addOperand("Bias", OperandKind::In)
  .addMember(MemberType::VectorUnsigned, "Kernels")
  .addMember(MemberType::VectorUnsigned, "Strides")
  .addMember(MemberType::VectorUnsigned, "Pads")
  .addMember(MemberType::Unsigned, "Group")
  .autoIRGen();
```

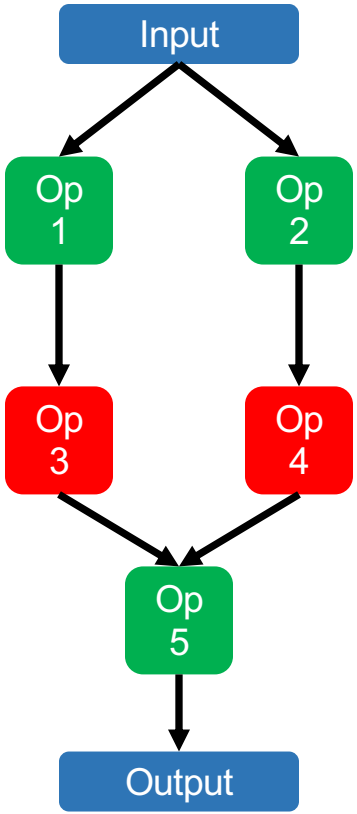
```
bool CPU::generateInst(Node *N, IRGenVisitor &irgen) const override {
  if (auto *N = llvm::dyn_cast<CPUConvDKKC8Node>(node)) {
    auto *I = valueForNode(N->getInput());
    auto *F = valueForNode(N->getFilter());
    auto *B = valueForNode(N->getBias());
    auto resTy = N->getResult().getType();
    auto *dest = builder_.createAllocActivationInst("alloc", resTy);
    auto *X =
        builder_.createCPUConvDKKC8Inst("N", dest, I, F, B,
                                        N->getKernels(), N->getStrides(),
                                        N->getPads(), N->getGroup());
    registerIR(N->getResult(), X->getDest());
    return true;
  }
  return false;
}
```

Initializing Glow Graphs with ONNXIFI



PyTorch network

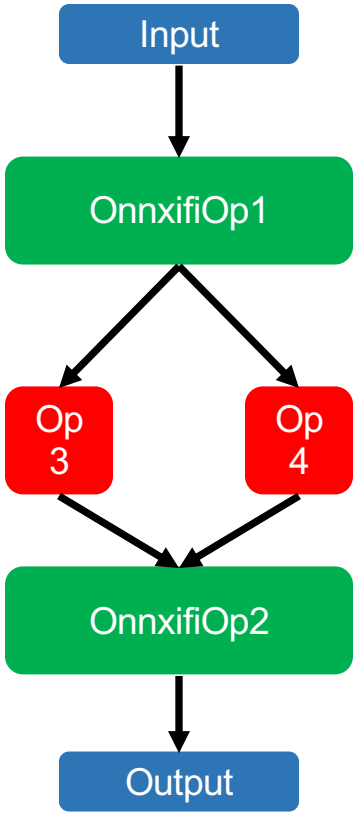
Initializing Glow Graphs with ONNXIFI



PyTorch network colored by

Glow operator support

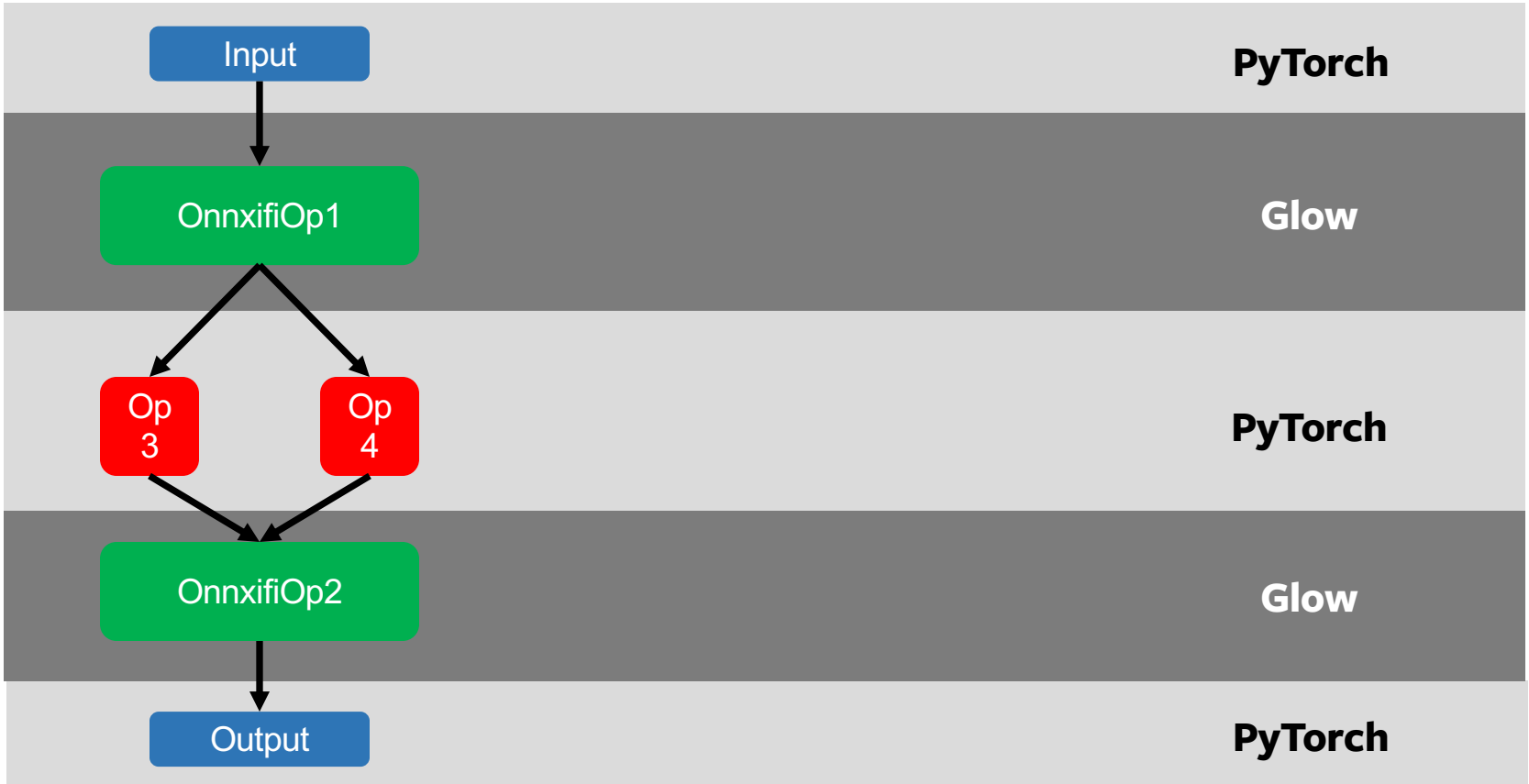
Initializing Glow Graphs with ONNXIFI



OnnxifiOp

- **Wraps the subgraph** that has been delegated to Glow
- Manages **IO with Glow** via ONNXIFI, waits for results

Initializing Glow Graphs with ONNXIFI



Memory management for HW accelerators

- Accelerators have many processing elements (PEs)
- Usually no caches, no out-of-order execution
- Accelerators have multiple memory banks with different properties in terms of size and access speed: DRAM, SRAM, scratchpads, etc
  - Memory in all memory banks needs to be managed explicitly
  - Data transfers between some memory banks is possible only by means of explicit DMA commands
- Instructions may have requirements on the memory banks to be used for their operands and on the memory layout of their operands

Static memory allocation

- Memory can't be 'malloc'-ed on the HW accelerator
- Glow compiler has to manage and allocate the on-device memory statically
  - Allocation is performed for each memory bank
- Live buffers are allocated and freed.
- Scheduler and IR optimizer reduce memory pressure and shorten buffers lifetime.

Memory management strategy

- Ensure that buffer operands are loaded into the required memory banks, usually into fast scratchpads
- Try to keep data in fast memory banks as long as possible
- Evict data from fast memory banks only if it cannot be avoided
  - Often involves explicit DMA transfers
- Minimize the cost of evictions, i.e. slow data transfers between memory banks

Sounds familiar???

Yes, it is rather similar to **register allocation**!!!

- The analogy is: buffers == virtual registers, fast memory banks == physical registers, eviction from fast memory banks == register spilling
- But there are differences:
  - Buffers have different varying sizes
  - Evicting buffers from fast memory banks is expensive, often involves DMA data transfers
    - Cost of eviction is proportional to the amount of data to be transferred!

How to improve performance?

- Keep the processing elements always busy
- Hide latency of memory accesses
  - Use pre-fetching
  - Intermix data-fetching and computation
- Partition data to fit into accelerator's memory banks and process it in parallel
  - e.g. scatter/gather approach
- Reduce the amount of data transfers
  - Between accelerator RAM and fast memory banks
  - Between the host and accelerator RAM
- Use a good scheduling algorithm

EXAMPLE: POSSIBLE CODE TO PERFORM CONVOLUTION

Set DMA mode to stride [0x0, 0x0, 0x400, 0x400, 0x0, 0x0]
Start DMA request for block #1 into SRAM address 0xA000
Start DMA request for block #2 into SRAM address 0x0B800
Start DMA request for block #3 into SRAM address 0xFF0000
Configure the state of activation unit to 'RELU'
Wait for #1 and #2
Start Matrix Multiplication on #1 and #2 to SRAM 0xD0000
Wait for #3
Start Matrix Multiplication on #1 and #3 to SRAM 0xD0400
Start DMA request for block #4 into address SRAM 0xFF0000
Wait for #4
Start Matrix Multiplication on #1 and #4 to SRAM 0xA0400
Configure the state of activation unit to lookup table from address SRAM 0xB0400

Why We Built Glow

Existing non-compiler frameworks “node visitor” design could not support our vision.

Gaps:

- Frameworks were not designed to support dozens of different architectures.
- No facilities for optimizing graphs – just protobufs.
- No way to reuse utilities, such as register allocator, across different devices.
- Missing features, such as quantization, memory optimizer, etc.
- No way to scale up to servers with multiple acceleration devices.
- No way to scale down to IoT devices with ahead-of-time compilation.