



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.

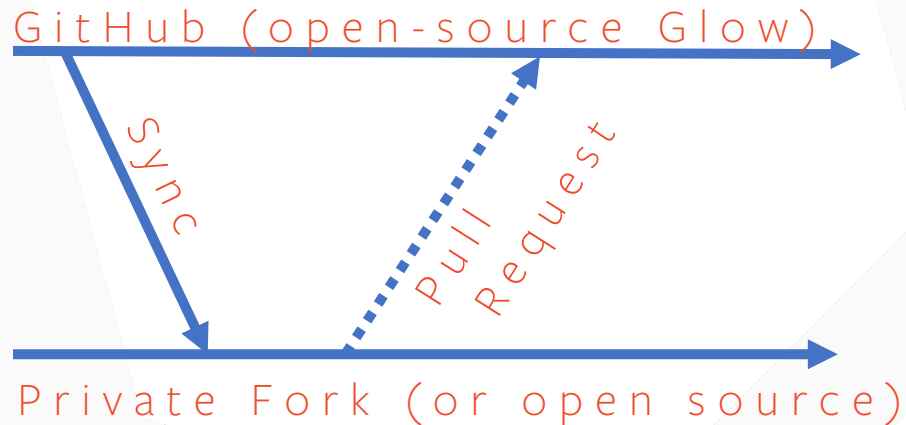
Vendors can develop their backend in a private fork.

Download and sync upstream changes.

Can submit features and bug fixes with pull requests.

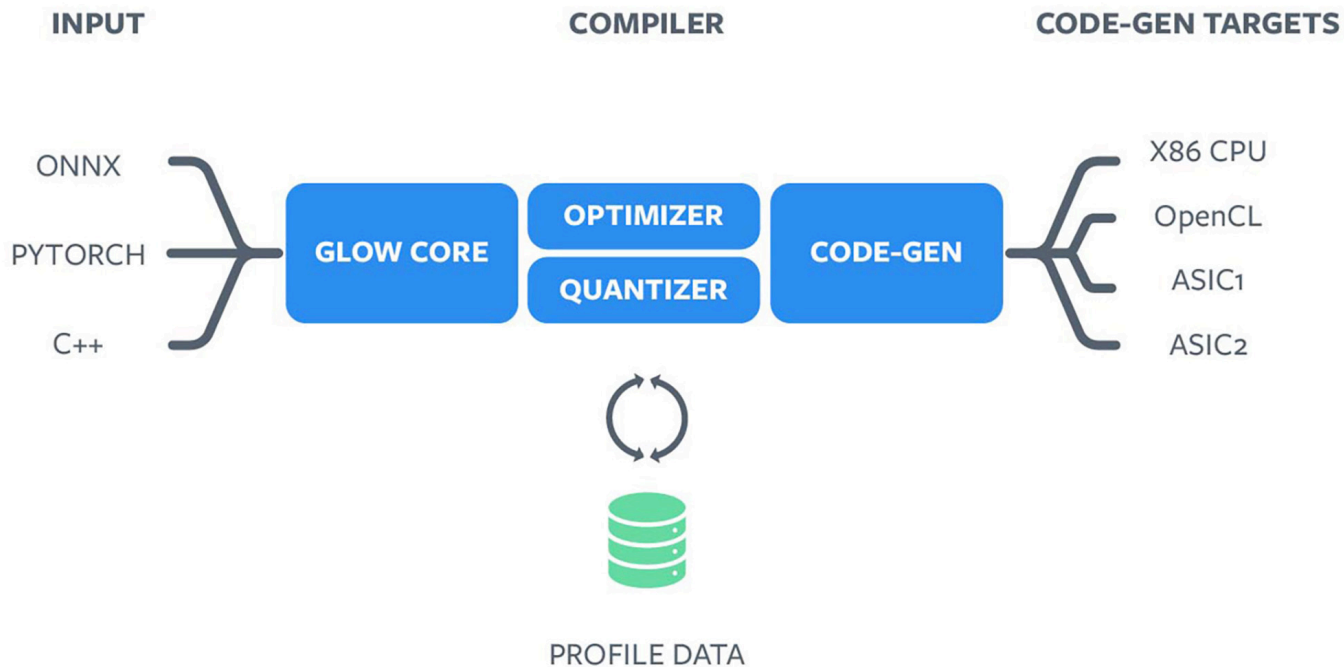
Open source PRs are reviewed by the community.

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





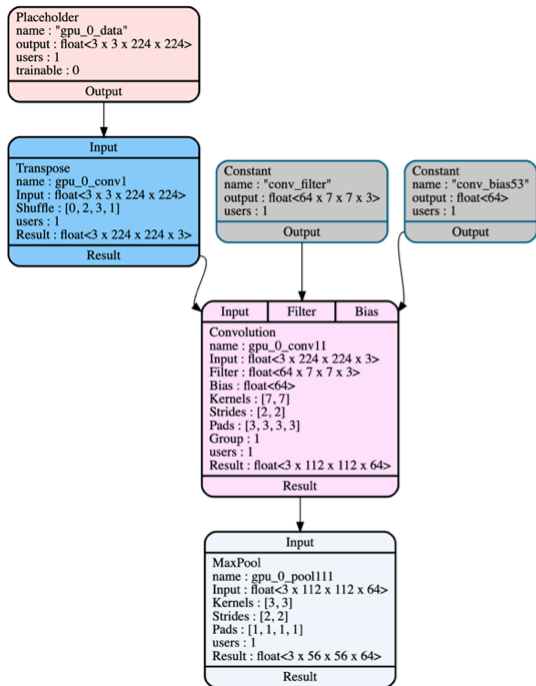
Glow Compiler & Runtime

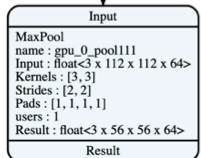
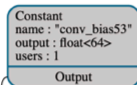
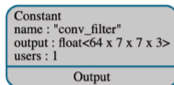
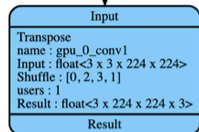
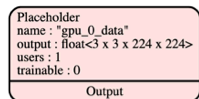




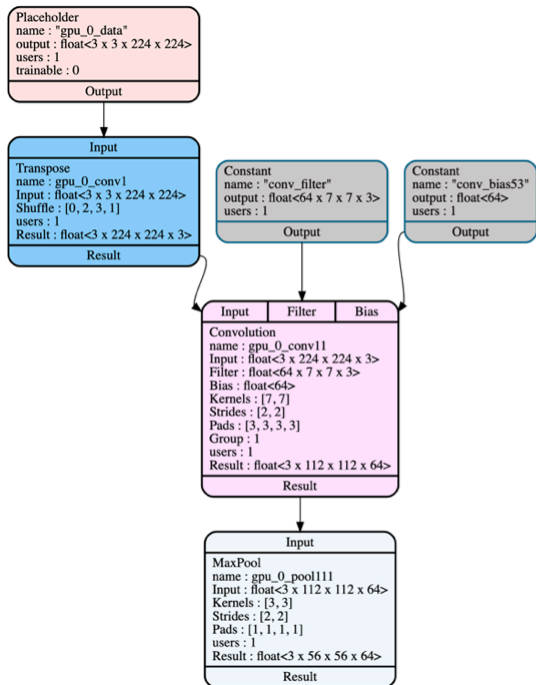
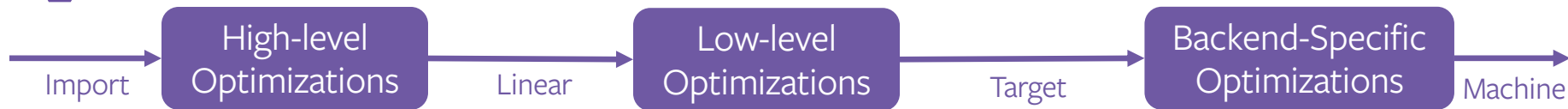
Import

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

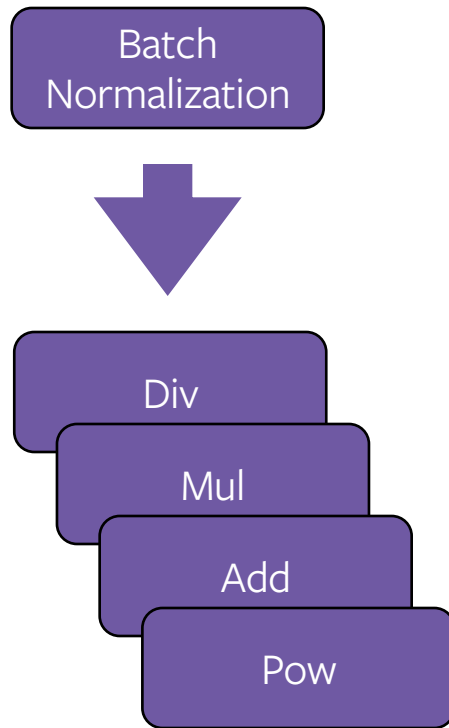
```
BB.newNode("Convolution")
    .addInput("Input")
    .addInput("Filter")
    .addInput("Bias")
    .addMember(MemberType::VectorUnsigned, "Kernels")
    .addMember(MemberType::VectorUnsigned, "Strides")
    .addMember(MemberType::VectorUnsigned, "Pads")
    .addMember(MemberType::Unsigned, "Group")
    .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

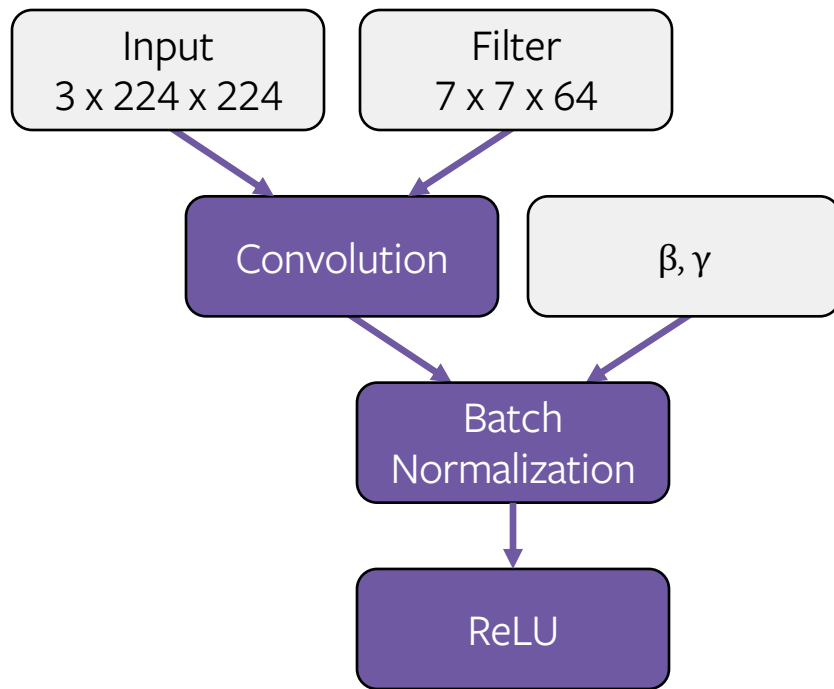
- ML frameworks have 100s of operators
 - ... times N accelerator backends
 - ... implemented in custom assembly
- Not scalable
- Instead, Lower complex ops to simple kernels





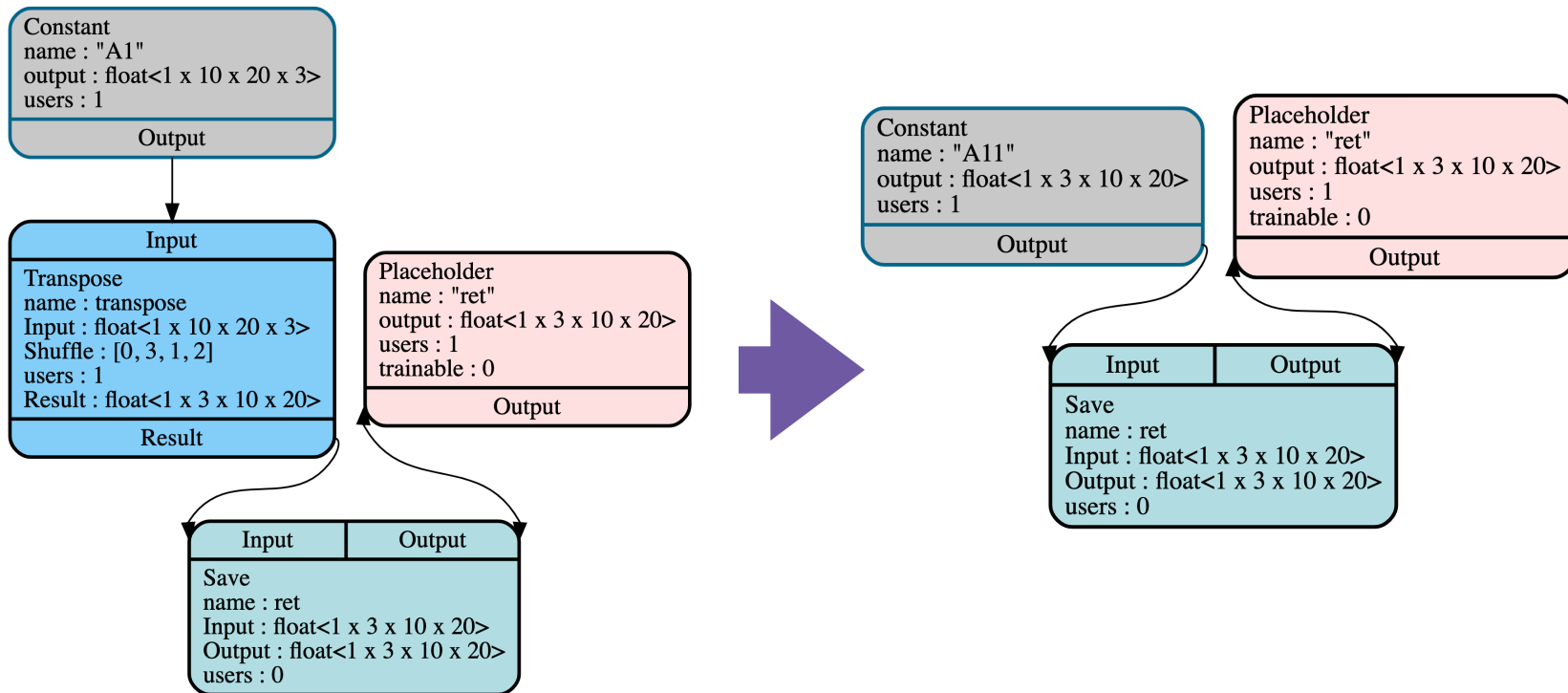
Graph Optimizations

- 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



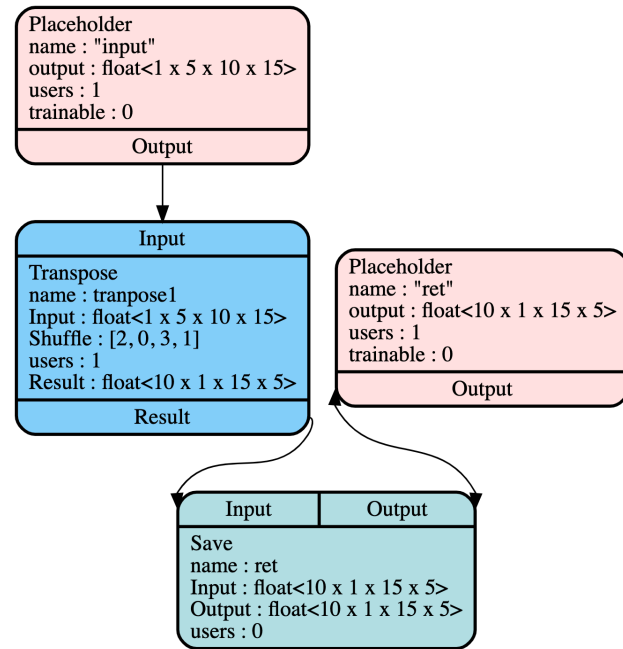
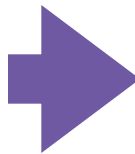
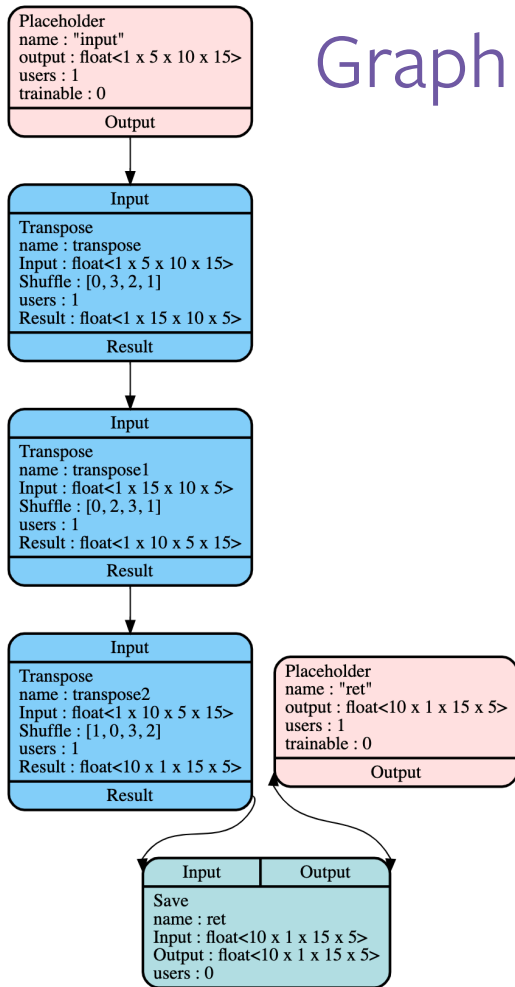


Graph Optimizations



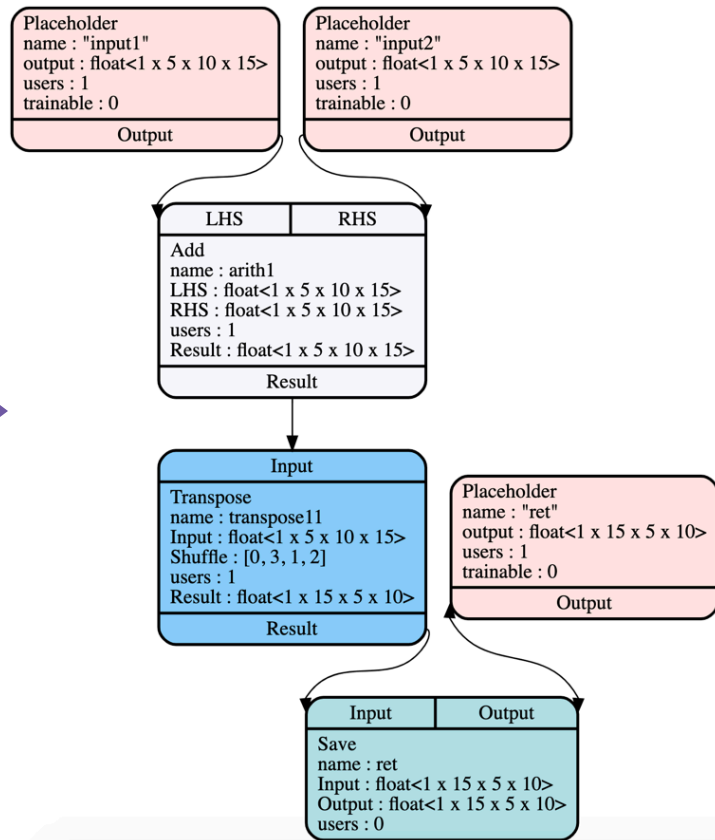
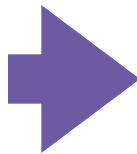
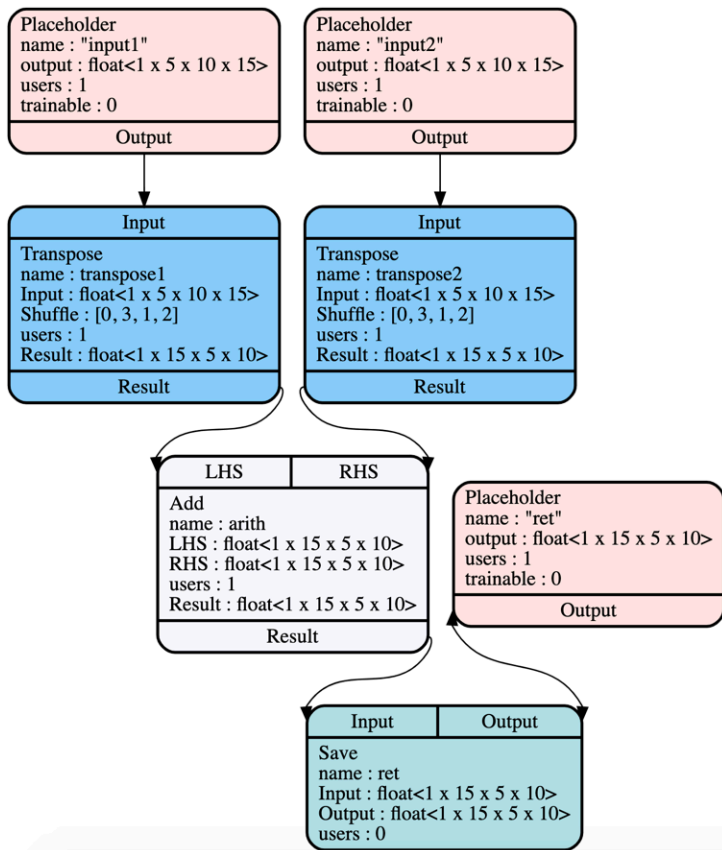


Graph Optimizations



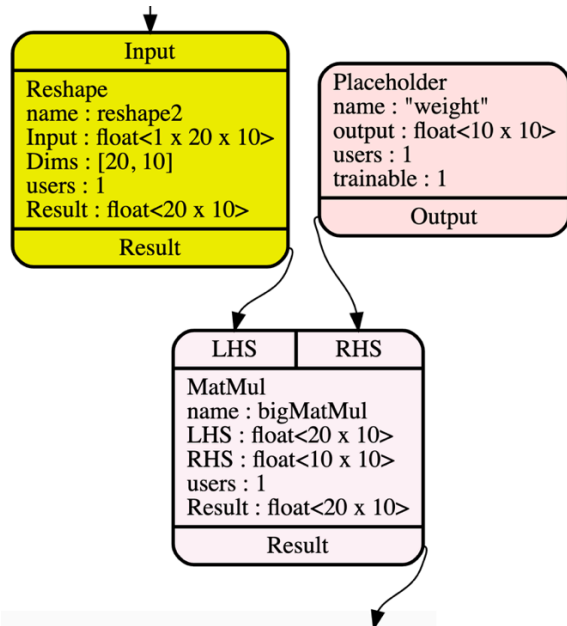
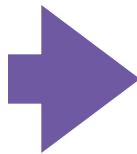
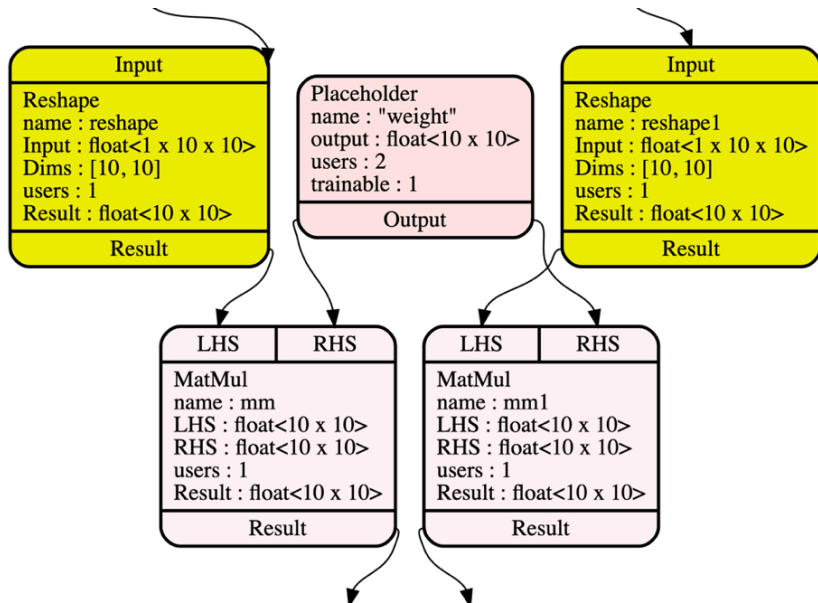


Graph Optimizations





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

```
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]");
```

Instruction-specific:

- generateInst()
- shouldShareBuffers()



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
}
```

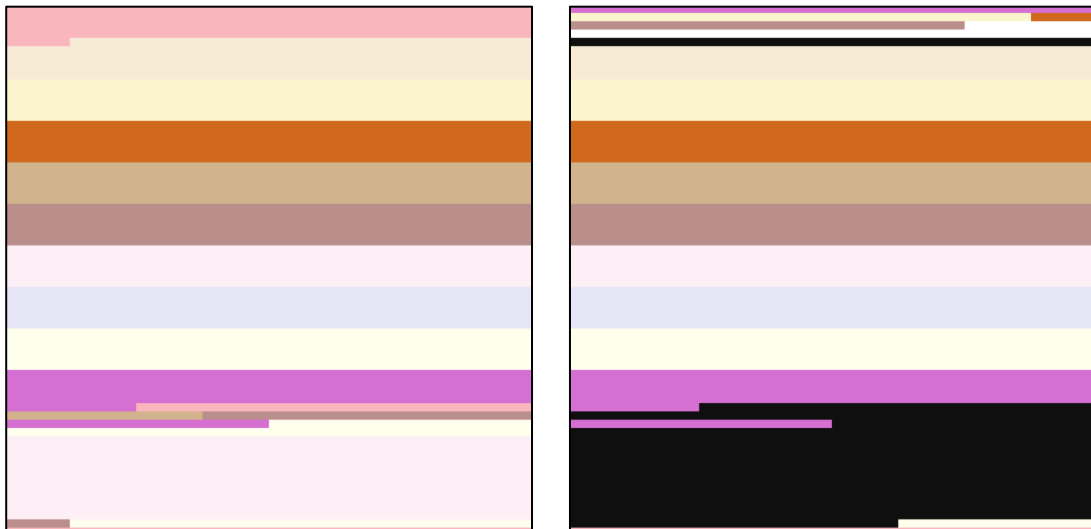
- Linear instruction-based address-only representation
- Operands are typed pointers to buffers
- Memory optimizations:
 - Instruction scheduling
 - Buffer sharing
 - Shortening buffer lifetime



Implementing a Backend

All backends:

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



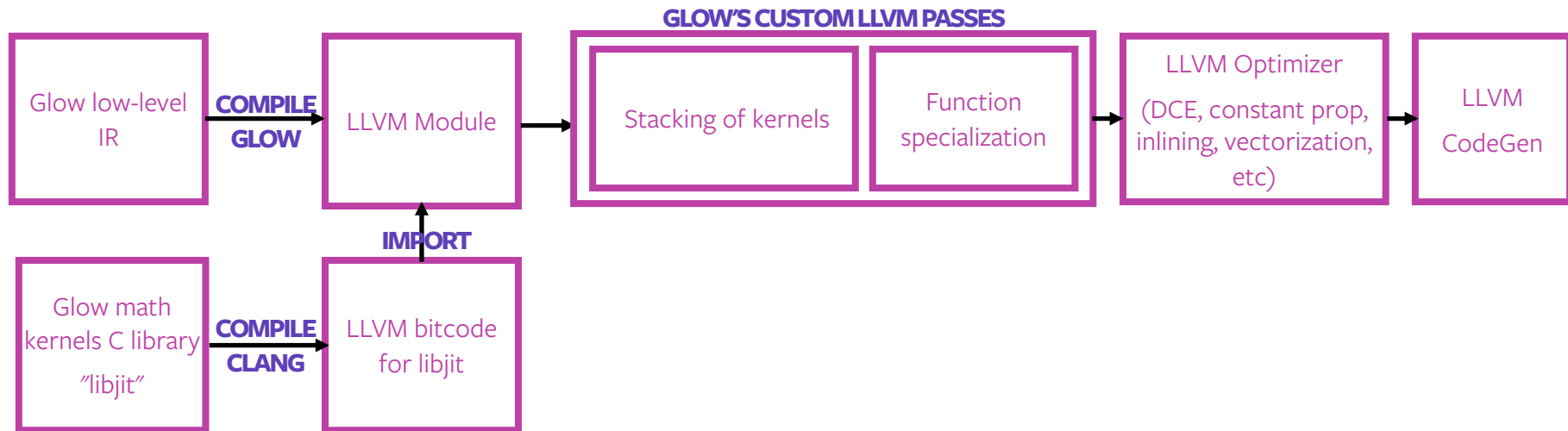
Instruction-specific:

- generateInst()
- **shouldShareBuffers()**

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



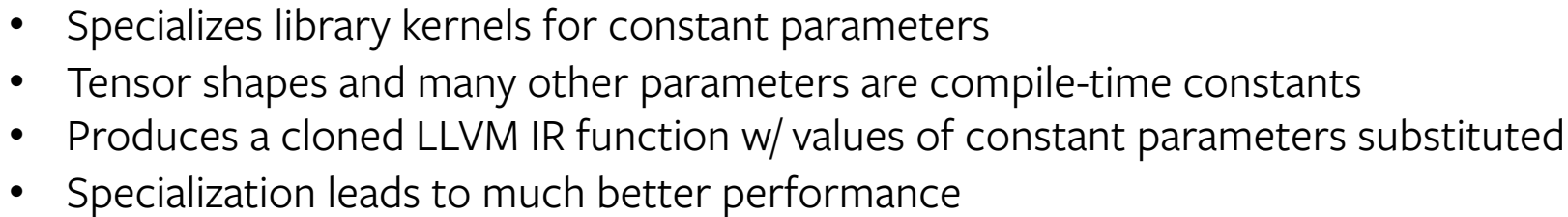
CPU/JIT backend design





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



```
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;
}
```

[illegible]

```

1  @param[in]  x0    # @param[in]  x0
2  @param[in]  dx    # @param[in]  dx
3  @param[in]  dx0    # @param[in]  dx0
4  @param[in]  dx1    # @param[in]  dx1
5  @param[in]  dx2    # @param[in]  dx2
6  @param[in]  dx3    # @param[in]  dx3
7  @param[in]  dx4    # @param[in]  dx4
8  @param[in]  dx5    # @param[in]  dx5
9  @param[in]  dx6    # @param[in]  dx6
10 @param[in]  dx7    # @param[in]  dx7
11 @param[in]  dx8    # @param[in]  dx8
12 @param[in]  dx9    # @param[in]  dx9
13 @param[in]  dx10   # @param[in]  dx10
14 @param[in]  dx11   # @param[in]  dx11
15 @param[in]  dx12   # @param[in]  dx12
16 @param[in]  dx13   # @param[in]  dx13
17 @param[in]  dx14   # @param[in]  dx14
18 @param[in]  dx15   # @param[in]  dx15
19 @param[in]  dx16   # @param[in]  dx16
20 @param[in]  dx17   # @param[in]  dx17
21 @param[in]  dx18   # @param[in]  dx18
22 @param[in]  dx19   # @param[in]  dx19
23 @param[in]  dx20   # @param[in]  dx20
24 @param[in]  dx21   # @param[in]  dx21
25 @param[in]  dx22   # @param[in]  dx22
26 @param[in]  dx23   # @param[in]  dx23
27 @param[in]  dx24   # @param[in]  dx24
28 @param[in]  dx25   # @param[in]  dx25
29 @param[in]  dx26   # @param[in]  dx26
30 @param[in]  dx27   # @param[in]  dx27
31 @param[in]  dx28   # @param[in]  dx28
32 @param[in]  dx29   # @param[in]  dx29
33 @param[in]  dx30   # @param[in]  dx30
34 @param[in]  dx31   # @param[in]  dx31
35 @param[in]  dx32   # @param[in]  dx32
36 @param[in]  dx33   # @param[in]  dx33
37 @param[in]  dx34   # @param[in]  dx34
38 @param[in]  dx35   # @param[in]  dx35
39 @param[in]  dx36   # @param[in]  dx36
40 @param[in]  dx37   # @param[in]  dx37
41 @param[in]  dx38   # @param[in]  dx38
42 @param[in]  dx39   # @param[in]  dx39
43 @param[in]  dx40   # @param[in]  dx40
44 @param[in]  dx41   # @param[in]  dx41
45 @param[in]  dx42   # @param[in]  dx42
46 @param[in]  dx43   # @param[in]  dx43
47 @param[in]  dx44   # @param[in]  dx44
48 @param[in]  dx45   # @param[in]  dx45
49 @param[in]  dx46   # @param[in]  dx46
50 @param[in]  dx47   # @param[in]  dx47
51 @param[in]  dx48   # @param[in]  dx48
52 @param[in]  dx49   # @param[in]  dx49
53 @param[in]  dx50   # @param[in]  dx50
54 @param[in]  dx51   # @param[in]  dx51
55 @param[in]  dx52   # @param[in]  dx52
56 @param[in]  dx53   # @param[in]  dx53
57 @param[in]  dx54   # @param[in]  dx54
58 @param[in]  dx55   # @param[in]  dx55
59 @param[in]  dx56   # @param[in]  dx56
60 @param[in]  dx57   # @param[in]  dx57
61 @param[in]  dx58   # @param[in]  dx58
62 @param[in]  dx59   # @param[in]  dx59
63 @param[in]  dx60   # @param[in]  dx60
64 @param[in]  dx61   # @param[in]  dx61
65 @param[in]  dx62   # @param[in]  dx62
66 @param[in]  dx63   # @param[in]  dx63
67 @param[in]  dx64   # @param[in]  dx64
68 @param[in]  dx65   # @param[in]  dx65
69 @param[in]  dx66   # @param[in]  dx66
70 @param[in]  dx67   # @param[in]  dx67
71 @param[in]  dx68   # @param[in]  dx68
72 @param[in]  dx69   # @param[in]  dx69
73 @param[in]  dx70   # @param[in]  dx70
74 @param[in]  dx71   # @param[in]  dx71
75 @param[in]  dx72   # @param[in]  dx72
76 @param[in]  dx73   # @param[in]  dx73
77 @param[in]  dx74   # @param[in]  dx74
78 @param[in]  dx75   # @param[in]  dx75
79 @param[in]  dx76   # @param[in]  dx76
80 @param[in]  dx77   # @param[in]  dx77
81 @param[in]  dx78   # @param[in]  dx78
82 @param[in]  dx79   # @param[in]  dx79
83 @param[in]  dx80   # @param[in]  dx80
84 @param[in]  dx81   # @param[in]  dx81
85 @param[in]  dx82   # @param[in]  dx82
86 @param[in]  dx83   # @param[in]  dx83
87 @param[in]  dx84   # @param[in]  dx84
88 @param[in]  dx85   # @param[in]  dx85
89 @param[in]  dx86   # @param[in]  dx86
90 @param[in]  dx87   # @param[in]  dx87
91 @param[in]  dx88   # @param[in]  dx88
92 @param[in]  dx89   # @param[in]  dx89
93 @param[in]  dx90   # @param[in]  dx90
94 @param[in]  dx91   # @param[in]  dx91
95 @param[in]  dx92   # @param[in]  dx92
96 @param[in]  dx93   # @param[in]  dx93
97 @param[in]  dx94   # @param[in]  dx94
98 @param[in]  dx95   # @param[in]  dx95
99 @param[in]  dx96   # @param[in]  dx96
100 @param[in]  dx97   # @param[in]  dx97
101 @param[in]  dx98   # @param[in]  dx98
102 @param[in]  dx99   # @param[in]  dx99
103 @param[in]  dx100  # @param[in]  dx100
104 @param[in]  dx101  # @param[in]  dx101
105 @param[in]  dx102  # @param[in]  dx102
106 @param[in]  dx103  # @param[in]  dx103
107 @param[in]  dx104  # @param[in]  dx104
108 @param[in]  dx105  # @param[in]  dx105
109 @param[in]  dx106  # @param[in]  dx106
110 @param[in]  dx107  # @param[in]  dx107
111 @param[in]  dx108  # @param[in]  dx108
112 @param[in]  dx109  # @param[in]  dx109
113 @param[in]  dx110  # @param[in]  dx110
114 @param[in]  dx111  # @param[in]  dx111
115 @param[in]  dx112  # @param[in]  dx112
116 @param[in]  dx113  # @param[in]  dx113
117 @param[in]  dx114  # @param[in]  dx114
118 @param[in]  dx115  # @param[in]  dx115
119 @param[in]  dx116  # @param[in]  dx116
120 @param[in]  dx117  # @param[in]  dx117
121 @param[in]  dx118  # @param[in]  dx118
122 @param[in]  dx119  # @param[in]  dx119
123 @param[in]  dx120  # @param[in]  dx120
124 @param[in]  dx121  # @param[in]  dx121
125 @param[in]  dx122  # @param[in]  dx122
126 @param[in]  dx123  # @param[in]  dx123
127 @param[in]  dx124  # @param[in]  dx124
128 @param[in]  dx125  # @param[in]  dx125
129 @param[in]  dx126  # @param[in]  dx126
130 @param[in]  dx127  # @param[in]  dx127
131 @param[in]  dx128  # @param[in]  dx128
132 @param[in]  dx129  # @param[in]  dx129
133 @param[in]  dx130  # @param[in]  dx130
134 @param[in]  dx131  # @param[in]  dx131
135 @param[in]  dx132  # @param[in]  dx132
136 @param[in]  dx133  # @param[in]  dx133
137 @param[in]  dx134  # @param[in]  dx134
138 @param[in]  dx135  # @param[in]  dx135
139 @param[in]  dx136  # @param[in]  dx136
140 @param[in]  dx137  # @param[in]  dx137
141 @param[in]  dx138  # @param[in]  dx138
142 @param[in]  dx139  # @param[in]  dx139
143 @param[in]  dx140  # @param[in]  dx140
144 @param[in]  dx141  # @param[in]  dx141
145 @param[in]  dx142  # @param[in]  dx142
146 @param[in]  dx143  # @param[in]  dx143
147 @param[in]  dx144  # @param[in]  dx144
148 @param[in]  dx145  # @param[in]  dx145
149 @param[in]  dx146  # @param[in]  dx146
150 @param[in]  dx147  # @param[in]  dx147
151 @param[in]  dx148  # @param[in]  dx148
152 @param[in]  dx149  # @param[in]  dx149
153 @param[in]  dx150  # @param[in]  dx150
154 @param[in]  dx151  # @param[in]  dx151
155 @param[in]  dx152  # @param[in]  dx152
156 @param[in]  dx153  # @param[in]  dx153
157 @param[in]  dx154  # @param[in]  dx154
158 @param[in]  dx155  # @param[in]  dx155
159 @param[in]  dx156  # @param[in]  dx156
160 @param[in]  dx157  # @param[in]  dx157
161 @param[in]  dx158  # @param[in]  dx158
162 @param[in]  dx159  # @param[in]  dx159
163 @param[in]  dx160  # @param[in]  dx160
164 @param[in]  dx161  # @param[in]  dx161
165 @param[in]  dx162  # @param[in]  dx162
166 @param[in]  dx163  # @param[in]  dx163
167 @param[in]  dx164  # @param[in]  dx164
168 @param[in]  dx165  # @param[in]  dx165
169 @param[in]  dx166  # @param[in]  dx166
170 @param[in]  dx167  # @param[in]  dx167
171 @param[in]  dx168  # @param[in]  dx168
172 @param[in]  dx169  # @param[in]  dx169
173 @param[in]  dx170  # @param[in]  dx170
174 @param[in]  dx171  # @param[in]  dx171
175 @param[in]  dx172  # @param[in]  dx172
176 @param[in]  dx173  # @param[in]  dx173
177 @param[in]  dx174  # @param[in]  dx174
178 @param[in]  dx175  # @param[in]  dx175
179 @param[in]  dx176  # @param[in]  dx176
180 @param[in]  dx177  # @param[in]  dx177
181 @param[in]  dx178  # @param[in]  dx178
182 @param[in]  dx179  # @param[in]  dx179
183 @param[in]  dx180  # @param[in]  dx180
184 @param[in]  dx181  # @param[in]  dx181
185 @param[in
```

SPECIALIZE FOR $N = 256$

- ✓ 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 $\text{sub}(z, \text{mul}(x, y))$
- 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()
.verify(VerifyKind::SameElementType, {"Dest", "Src"})
.autoVerify(VerifyKind::SameShape, {"Dest", "Src"})
.autoIRGen();
```

MULTIPLE KERNELS DOING TENSOR TRAVERSALS

```
for (unsigned i = 0; i < n; ++i) {
    dest1[i] = lhs1[i] * rhs[i];
}

for (unsigned i = 0; i < n; ++i) {
    dest2[i] = lhs2[i] - dest1[i];
}
```

KERNEL STACKING

SINGLE KERNEL DOING TENSOR TRAVERSAL

```
for (unsigned i = 0; i < n; ++i) {
    float tmp = lhs1[i] * rhs[i];
    dest1[i] = tmp;
    dest2[i] = lhs2[i] - tmp;
}
```




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

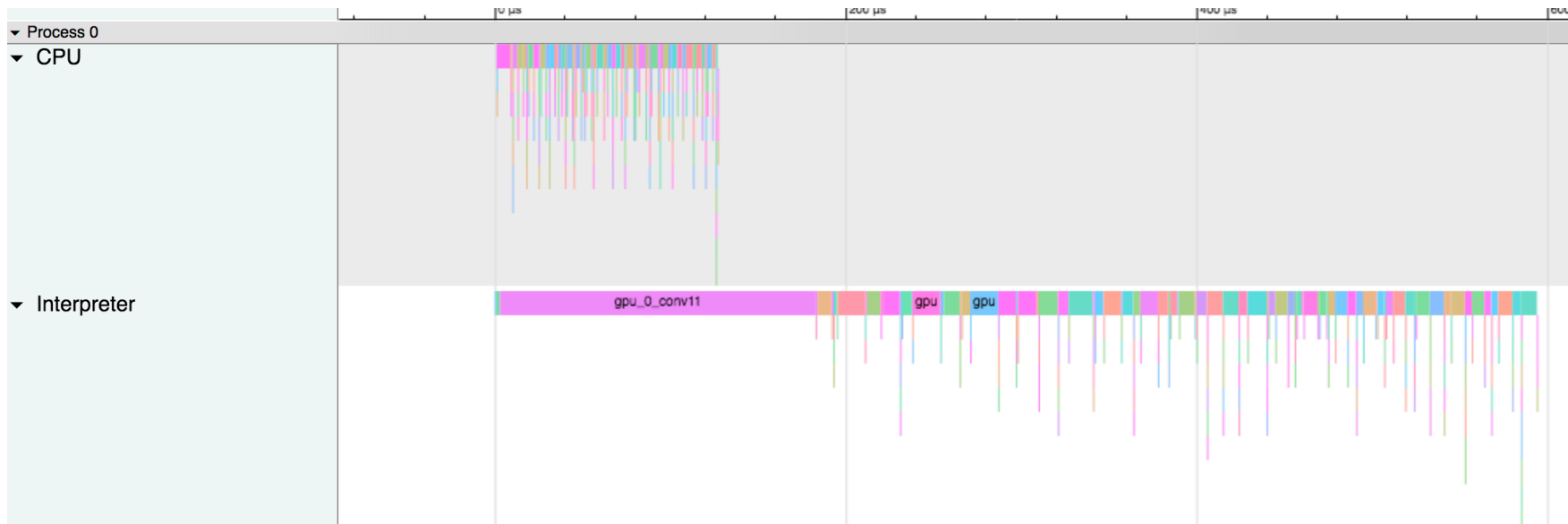
AN EXAMPLE OF A DEBUGGING SESSION USING LLDB

```
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 = cpuconvdckc8
    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 = cpuconvdckc8
    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 = 0x00000000100010060
(lldb) p &conv_filter__1025
(float (*)[32][3][3][256][8]) $1 = 0x000000001020f4210
(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



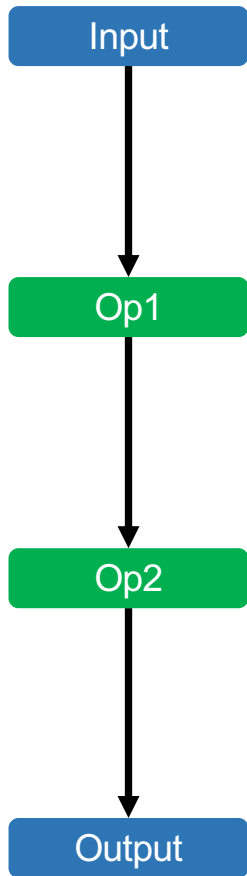


Profile-Guided Quantization Support

- **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
 - $\text{Float value} = (\text{qvalue} - \text{offset}) * \text{scale}$
- Very useful for **bringing up new unquantized models** on hardware requiring quantized ops



Profile-Guided Quantization Support

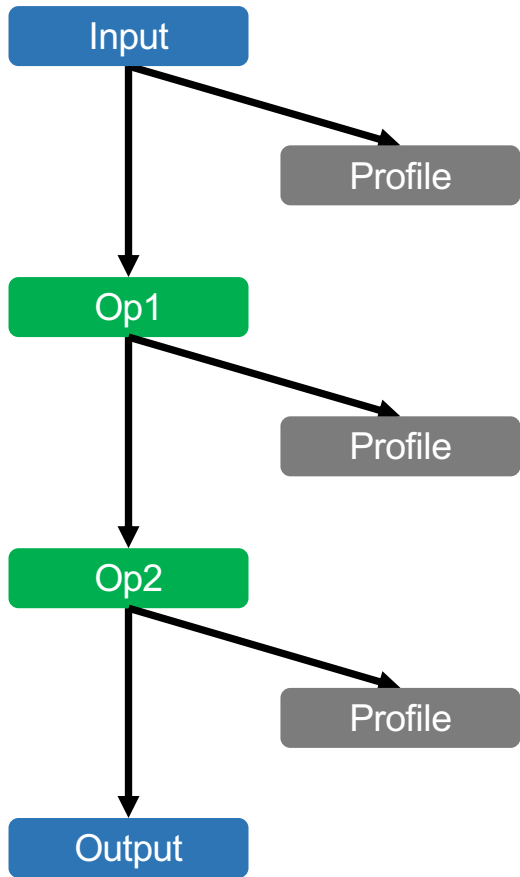


Profiling:

- 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

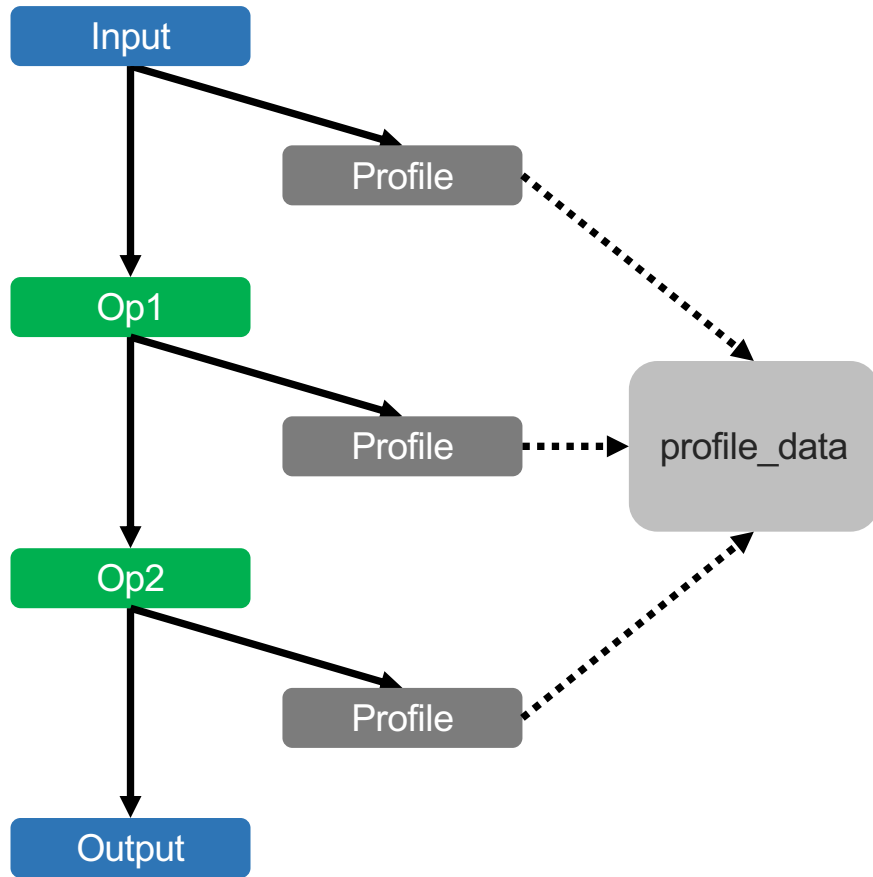


Profiling:

- 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

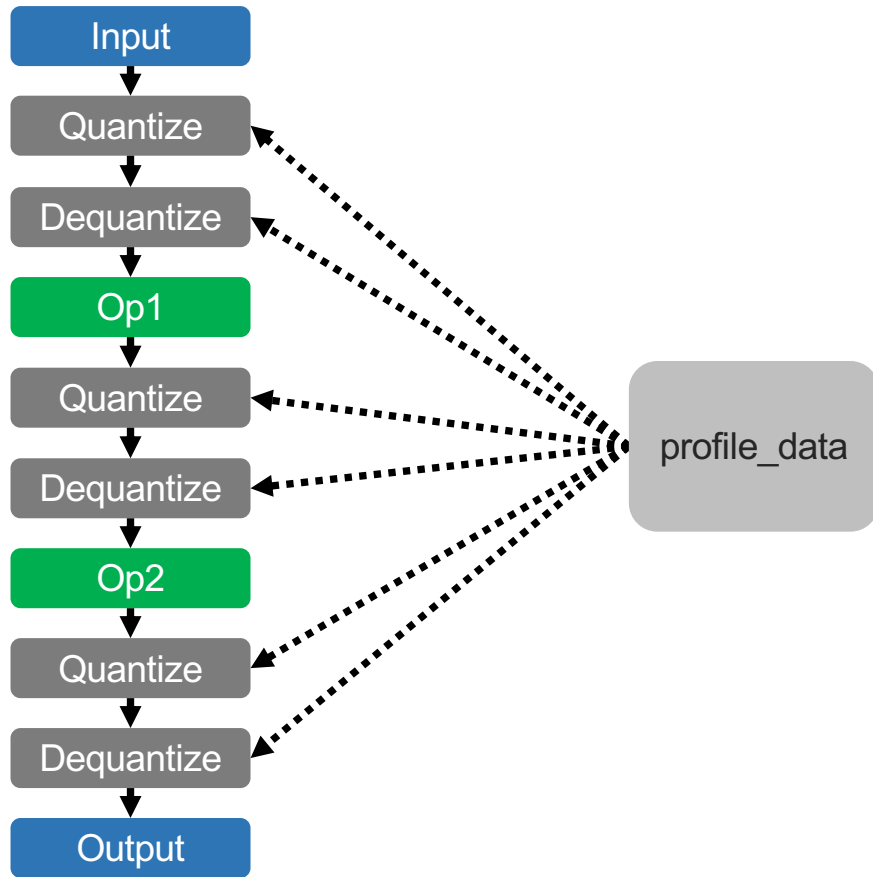


Profiling:

- 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

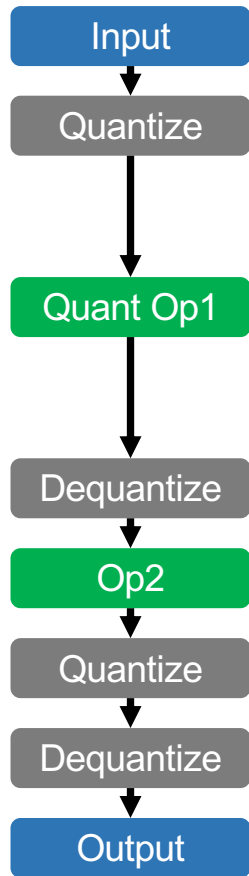


Quantization:

- 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

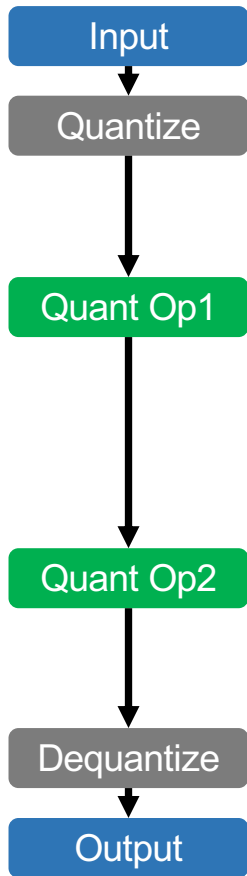


Quantization:

- 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

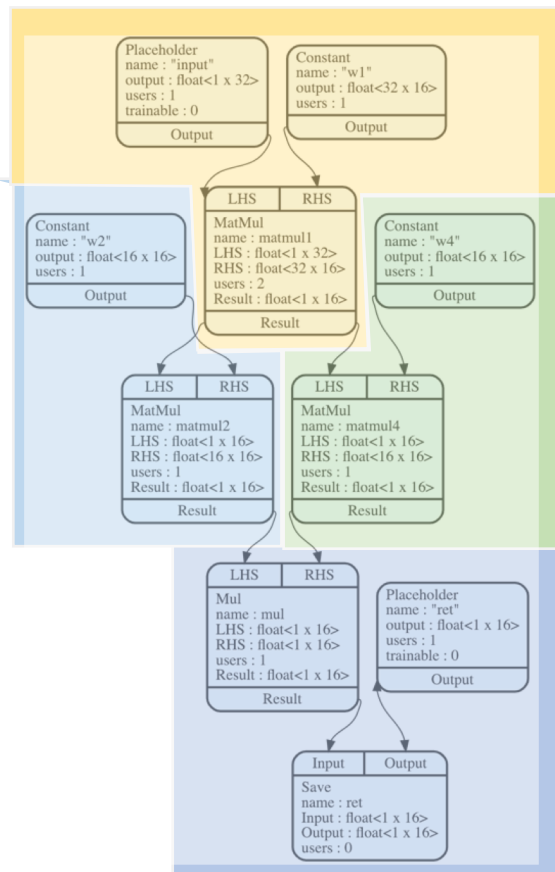


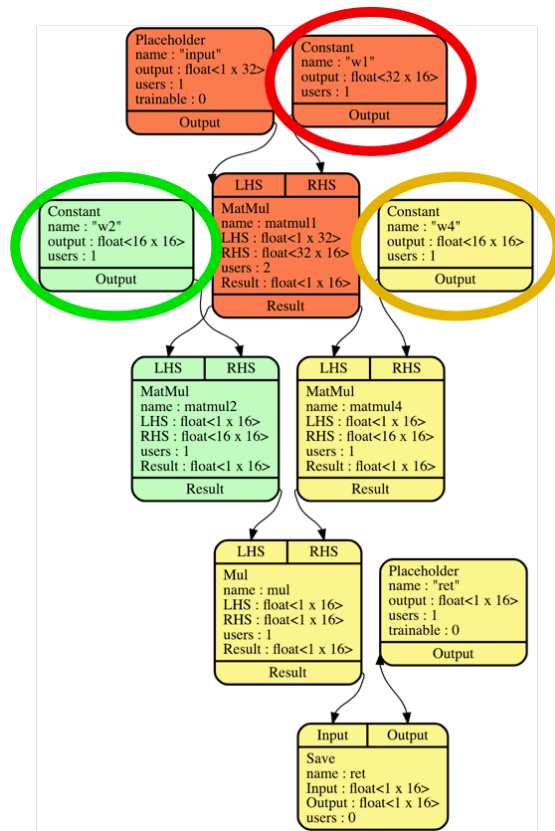
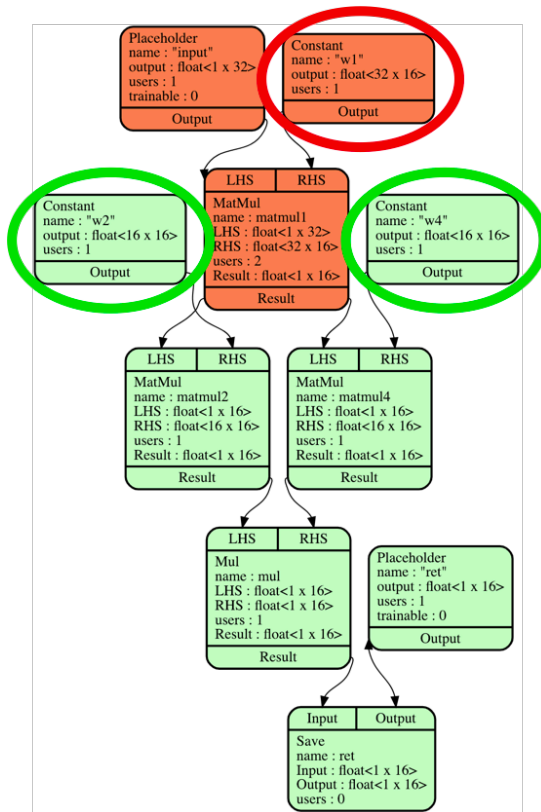
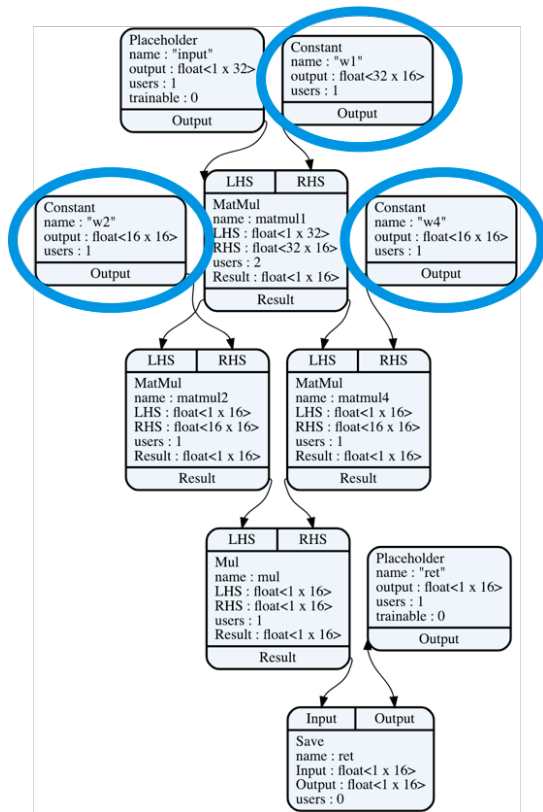
GLOW EXECUTION ENGINE

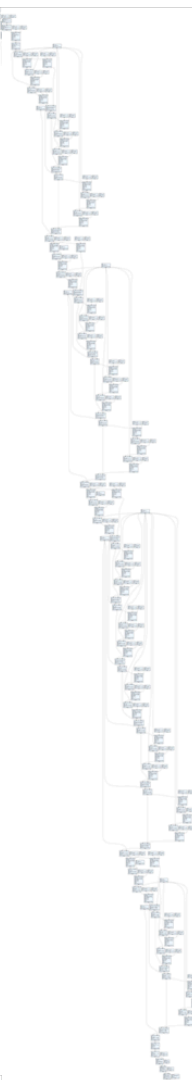
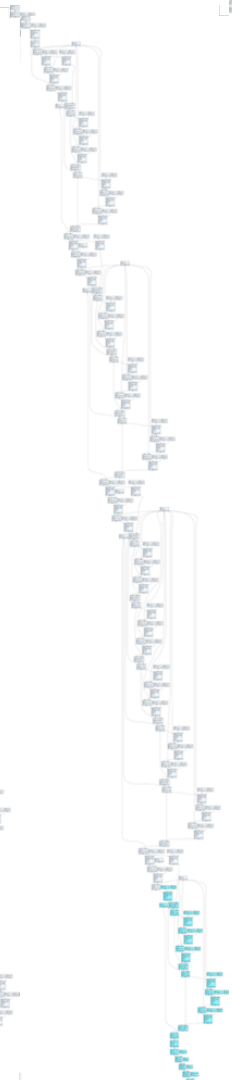
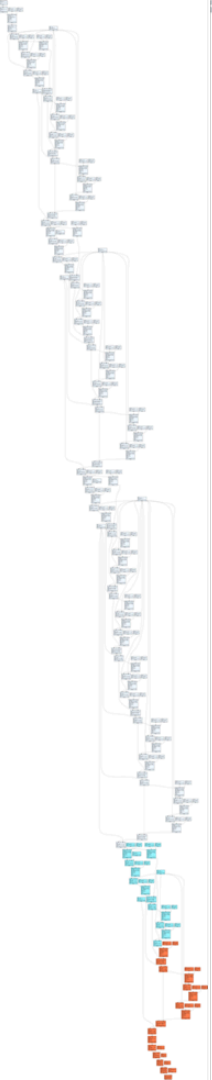
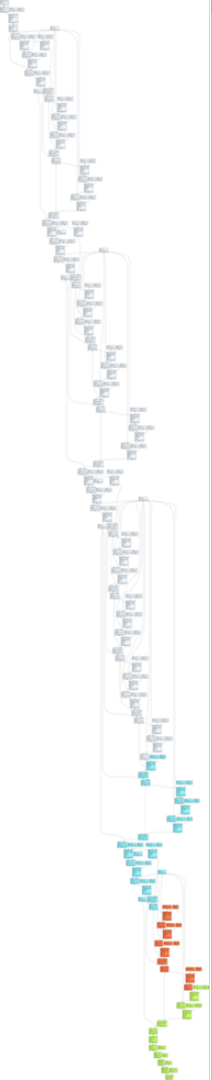
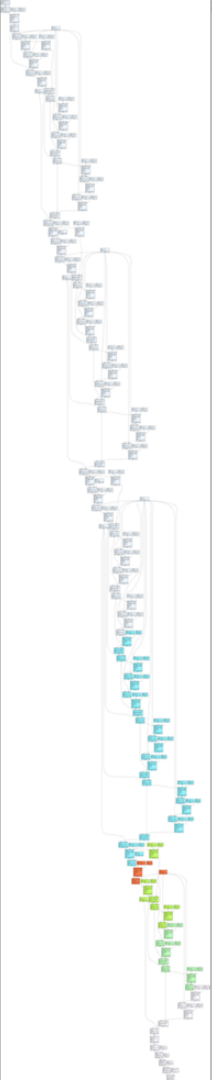
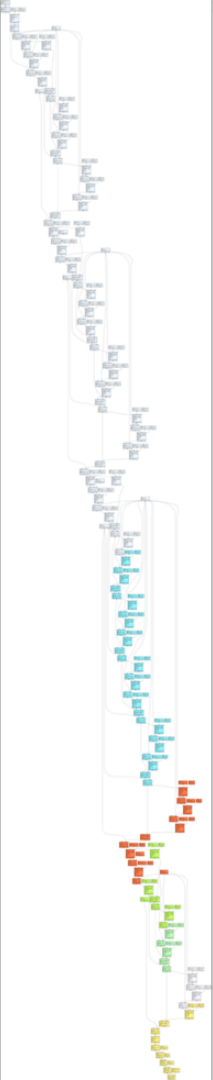
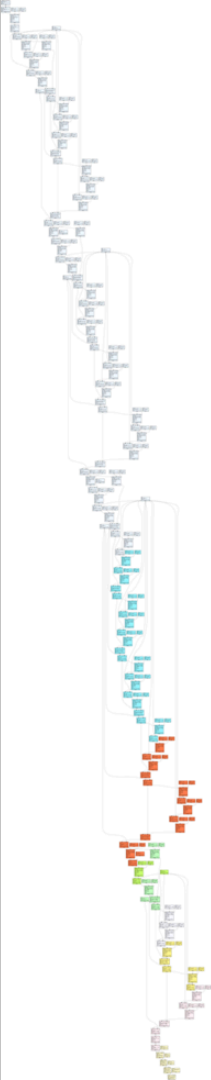
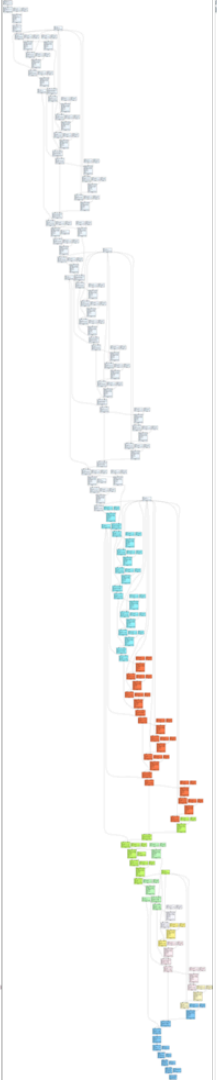
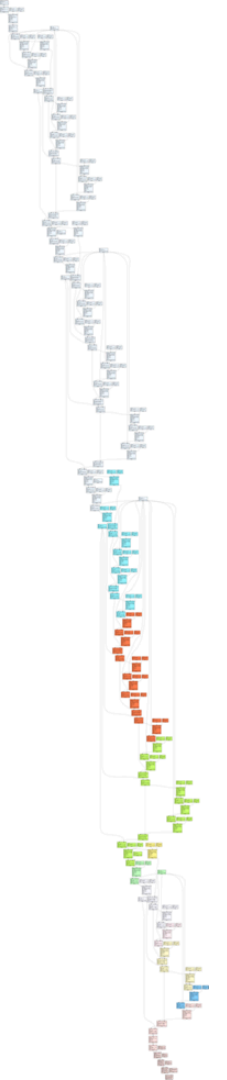
Partitioner

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









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



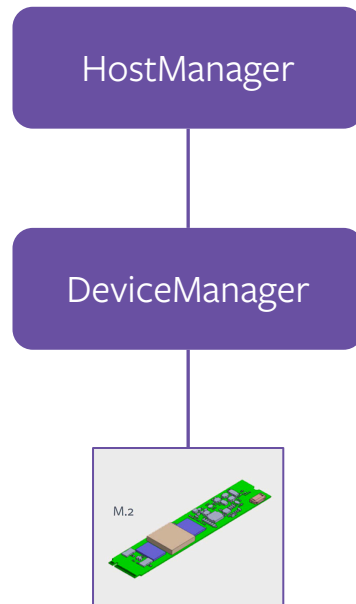
Implementing a Backend

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, IROGenVisitor &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:

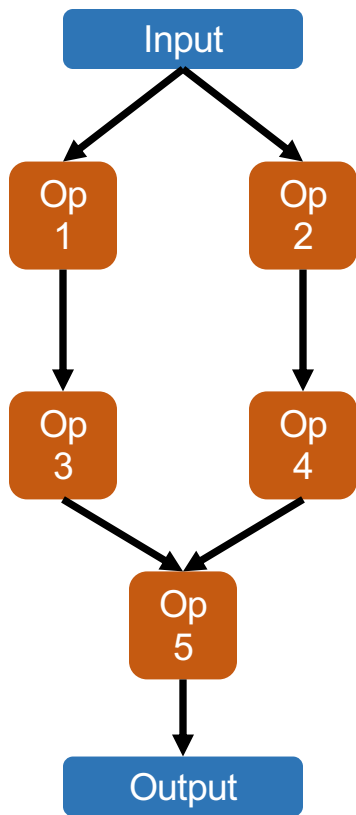
- generateInst()
- shouldShareBuffers()

```
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;
}
```



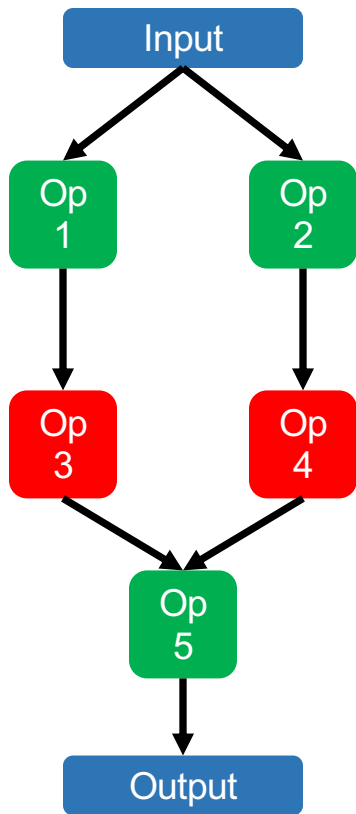
Initializing Glow Graphs with ONNXIFI



PyTorch network



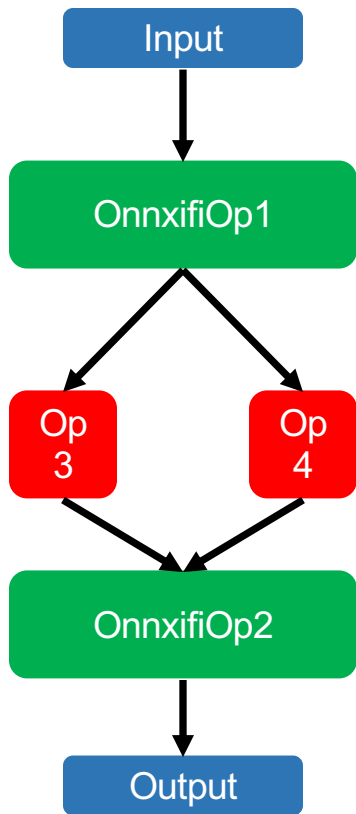
Initializing Glow Graphs with ONNXIFI



PyTorch network colored by
Glow operator support



Initializing Glow Graphs with ONNXIFI

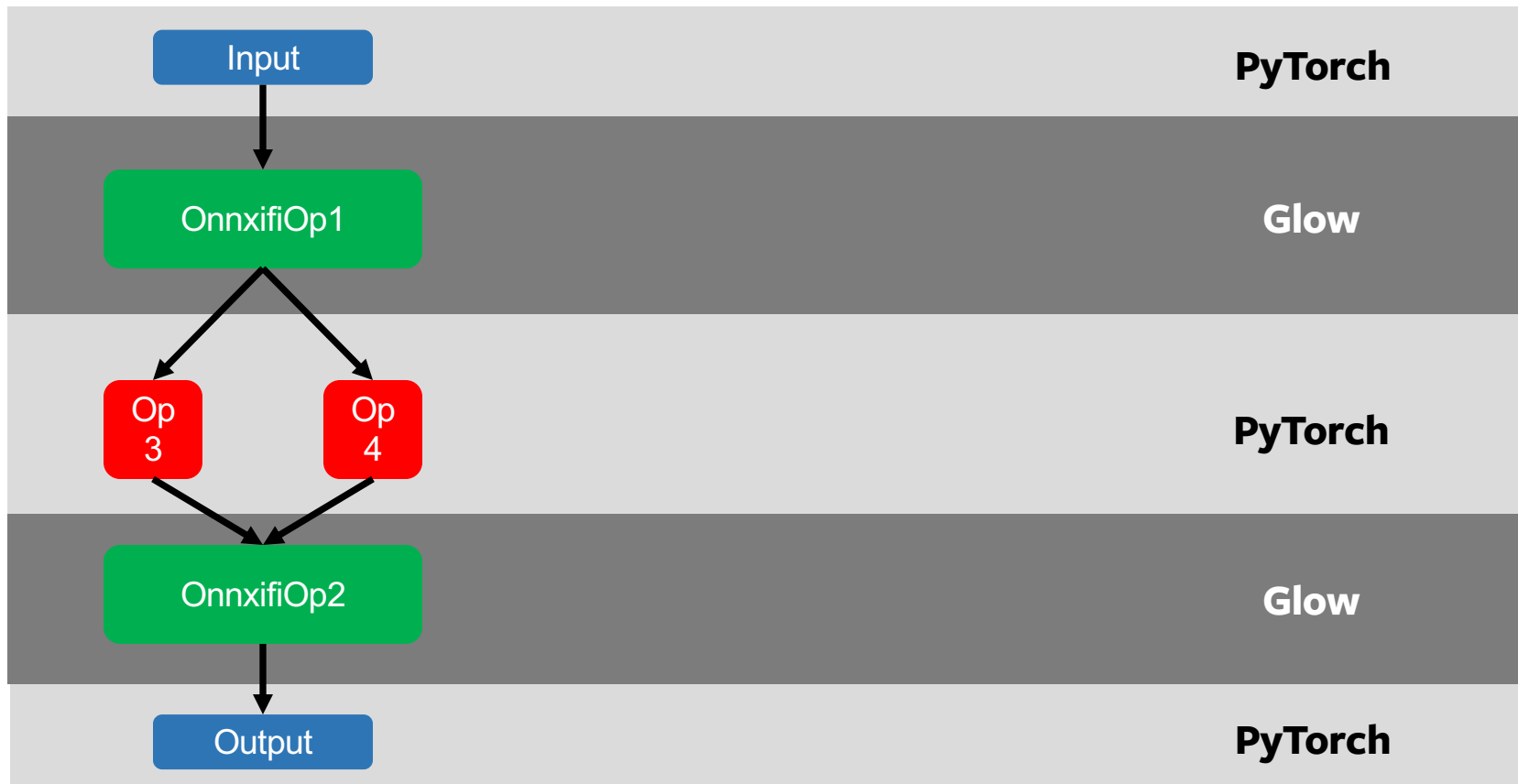


OnnxifiOp

- **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.