Neural Networks

"Where is my cat?"  "Где моя кошка?"

"Thumbs up!!!11 :)"  fake account!
PyTorch

A machine learning framework that provides a seamless path from research to production.
def foo(x, t):
    y = x.mm(x)
    print(y)  # still works!
    return y + t

x = torch.Tensor([[1,2],[3,5]])
y = torch.Tensor([[3,7],[1,2]])
foo(x, y)
ML at scale

- Facebook has billions of active users.
- Many services at Facebook use AI.
- NNs require lots of compute power.
- CPUs and GPUs are not efficient.
CPUs and GPUs are inefficient

- CPUs and GPUs work hard to extract parallelism.
- Matrix operations are very regular and expose lots of parallelism. Easy to accelerate.
- No need to waste power/area on useless features.
Accelerators are efficient because they are specialized

- Have many arithmetic execution units.
- Use dedicated local memories.
- Reduce the arithmetic bit-widths.
- Use a specialized programming model.

Facebook is building ML hardware acceleration ecosystem with partners using the Glow compiler.
Glow Compiler Design

INPUTS
- ONNX
- PyTorch
- C++ API

COMPILER
- Glow Core
- Optimizer
- Quantizer
- CodeGen

TARGET-SPECIFIC CODE GENERATOR
- Backend A
- Backend B
- Backend C
- CPU Backend

PGO
**Compilation pipeline**

1. **Graph**
   - High-level optimizer
   - IR
   - Low-level optimizer
   - Target IR
   - Accelerator backend

**Performs**:

- **High-level graph optimizations. Focus on linear-algebra kind of optimizations.**
- **Low-level IR optimizations. Focus on buffer and memory reuse optimizations.**
- **Target-specific lowering and optimizations for specific accelerator.**
High-Level Graph

- Static-shaped data-flow graph.
- Enables high-level domain-specific optimizations.
- Example: Change the matrix layout, merge batchnorm into conv, eliminate numeric re-scale.
Low-Level Instructions

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

```
declare {
  %input = WeightVar float<10 x 5000> mutable
  %rnn_initial_state = WeightVar float<10 x 20> mutable
  %rnn_Whh = WeightVar float<20 x 20> mutable
  %result = WeightVar float<100 x 500> mutable
}
program {
  %X_0 = allocactivation { Ty: float<10 x 500>}
  %X_01 = extracttensor @out %X_0, @in %input { Offsets: [0, 0]}
  %mergeLHS11 = allocactivation { Ty: float<100 x 500>}
  %mergeLHS111 = splat @out %mergeLHS11 { Value: 0.000000e+00}
  %mergeLHS112 = inserttensor @inout %mergeLHS11, @in %X_0 { Offsets: [0, 0], Count: 1, Axis: 0}
  %bigMatMul1_res = allocactivation { Ty: float<100 x 20>}
  %bigMatMul1 = matmul @out %bigMatMul1_res, @in %mergeLHS11, @in %rnn_Wxh
  %dealloc10 = deallocactivation @out %mergeLHS11
  %mergedBA = batchedadd @out %bigMatMul1_res, @in %bigMatMul1
  %rnn_add_0_res = allocactivation { Ty: float<10 x 20>}
  %fc_dot = matmul @out %rnn_add_0_res, @in %rnn_initial_state, @in %rnn_Whh
  %fc_add_bias = batchedadd @out %rnn_add_0_res, @in %fc_dot
```
Graph Lowering

- ML frameworks support hundreds of op kinds, implemented in C and CUDA.
- Writing hundreds of ops for accelerators isn’t scalable.
- Glow lowers complex high-level nodes into primitive nodes.
Automatic Node Generation

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

- Neural networks are resilient and can work with reduced bit-width (i8, fp16 instead of fp32).
- Quantization is the process of converting the network to integer arithmetic.
- Represent a range of real numbers using integers.

\[ \text{real} = (\text{integer} - \text{offset}) \times \text{scale} \]
Profile Guided Quantization

- Glow uses profile-guided quantization to estimate the range of each edge in the graph.
- Compiler optimizations eliminate numeric scale-conversion between nodes.
Why JIT?

- Significantly better performance vs an interpreter
  - Eliminates the dispatching overhead
  - Performs target-specific optimizations

- Just-in-time information allows for better compiler decisions & optimizations
  - Vendor libraries already JIT (CUDA, dnnMKL, libxssm), but one operator at a time
  - JITting the whole graph gives the compiler even more opportunities

- Ability to specialize based on the concrete NN model
  - Most shapes, types and sometimes addresses of memory buffers are constants at the time of JIT compilation
  - JIT can produce a tailor-made optimized code for the specific values/shapes of some/all parameters
  - Easy for a JIT, but not possible to achieve using any general-purpose libraries!
Glow JIT implementation

- 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
CPU/JIT design

Glow low-level IR

COMPILE GLOW

LLVM Module

Stacking of kernels

Function specialization

LLVM Optimizer
(DCE, constant prop, inlining, vectorization, etc)

LLVM CodeGen

Glow math kernels C library “libjit”

COMPILE CLANG

LLVM bitcode for libjit

IMPORT

GLOW’S CUSTOM LLVM PASSES

COMPILE GLOW
Function specialization

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

**EXAMPLE OF AN OPERATION**

```c
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**

**SPECIALIZED CODE**

- Runtime checks are eliminated
- Control flow is simplified
- Smaller code
- Faster execution
Stacking of kernels

- 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
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
  - Interesting e.g. for deployments 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
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!
• 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
Glow is a machine learning compiler for accelerators.
We are working with hardware partners on opening ML acceleration.
We rely on LLVM in many parts of the compiler.
Lot’s of hard problems to solve. Facebook is hiring. ;}

Glow
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
GLOW SUMMIT 2019

Tuesday, March 19, 9:30am

RSVP
Thank you