A Compiler for Deep Neural Network Accelerators to Generate Optimized Code for a Wide Range of Data Parameters from a Hand-crafted Computation Kernel

Eri Ogawa, Kazuaki Ishizaki, Hiroshi Inoue, Swagath Venkataramani, Jungwook Choi, Wei Wang, Vijayalakshmi Srinivasan, Moriyoshi Ohara, Kailash Gopalakrishnan

IBM Research
DNN accelerators are COOL!!

- Deep neural networks (DNNs) require a high performance engine due to their computational complexity
  - General purpose platforms, such as CPUs and GPUs, are not energy efficient

- DNN accelerators can dramatically improve the energy efficiency
  - increasing data and thread level parallelism per chip
  - limiting functionalities
    - e.g. no hardware-controlled cache memory,
      no complex branch instructions,
      no out-of-order processing

Google TPU v3 (https://cloud.google.com/tpu/)
Hierarchical software-controlled small scratchpads

Massive number of computation units and load/store units

Optimizing data transfers across computation units and scratchpad memories are critically important to achieve high performance and energy efficiency.

Our challenges in a compiler for the DNN accelerator

- Generating optimized data transfer code with appropriate address updates
  - Device memory To Lx scratchpad
  - Lx scratchpad To L0 scratchpad
  - L0 scratchpad To SFU/PE

- Generating code for all units
  - Each has a different program counter
  - Each has a different instruction set
Design principle of our compiler

- Use hand-crafted computation kernels
  - Maximize the utilization of a specific hardware

- Generate code for controlling loops to repeat executing each kernel
  - Adapt hand-crafted computation kernels for all parameters (e.g. input size)

- Enable standard as well as advanced optimizing-compiler techniques
  - Use intermediate representation (IR) to allow a compiler to analyze the code structure
Agenda

- Software stack for the target processor

- Compiler overview
  - How to compute a kernel
  - IR (Intermediate Representation)
  - Optimizer

- Evaluation

- Conclusion
Software stack for the target processor

- **TensorFlow**
  - Generating a computational graph from high-level DNN descriptions

- **Scheduler** [*]
  - Generating data partitioning parameters
    - Ones to partition and sequence computations across cores
    - Ones to orchestrate data staging and movement across memory hierarchy levels

- **Compiler**
  - Generating device code from the following two inputs
    - Data partitioning parameters
    - Annotated computation kernels

Compiler overview

- **IR generator**
  - Create a structured IR (Intermediate representation) from two inputs

- **Optimizer**
  - Optimize code by analyzing the IR

- **Device code generator**
  - Generate assembly code from the IR
How to compute a kernel

- Example: ReLU

Computation for each element in input images (max(0, x), x: input value)

Send output images back to device memory

Output images

Input images

Part of Input images

Part of Input images
IR generator (Lx load unit)

- Generate controlling loops to repeat executing the kernel

Kernel instructions written by library developers

3 dimensions of input data for L0-Lx level

Reg Init
R0(Lx addr) = imm

Loop (input channel - (Device memory size / Lx size))

Loop (minibatch - (Device memory size / Lx size))

Loop (image size - (Device memory size / Lx size))

Loop (input channel - (Lx size / L0 size))

Loop (minibatch - (Lx size / L0 size))

Loop (image size - (Lx size / L0 size))

Kernel Block
LOAD addr=R0(Lx addr) size=(L0 size) dst=L0SU

Reg Update
R0(Lx addr) += imm

Reg Update

Reg Update

Reg Update

Reg Update

Reg Update

Reg Update

3 dimensions of input data for Lx-Device memory level
IR generator (Device memory controller)

- Generate code for transferring data between the device memory and a scratchpad

Data transfer instructions which is automatically generated

ReLU IRs for DMC LU
(Load unit of Device Memory Controller)

Reg Init
R0(device mem addr) = imm
R1(Lx addr) = imm

Loop (input channel - (Device memory size / Lx size))
Loop (minibatch size - (Device memory size / Lx size))
Loop (image size - (Device memory size / Lx size))

Data Transfer
SEND addr=R0(device mem addr) addr=R1(Lx addr) size=(Lx size)

Reg Update
R0(device mem addr) += imm
R1(Lx addr) += imm

Reg Update

Reg Update

3 dimensions of input data for Lx-Device memory level
Optimizer

- Enable performance features
  - Eliminate unnecessary instructions (e.g. register += 0)
  - Fuse certain nested loops
  - Enable double buffering
  - Enable software pipelining

- Generate code of zero-padding for convolution operations
  - Align input data at device memory controller for convolution operators by filling additional zeros
Evaluation

- Execution time of the compiler generated code for five kernels on a cycle-accurate simulator of the target hardware
  - Our compiler generated high performance code as equally tuned as hand-written code
    - X axis: five kernels we evaluated
    - Y axis: the execution time of the generated code normalized by that of hand-written code

![Bar chart showing execution time normalized by handwritten code for different kernels. The kernels are relu, sigmoid, avgpool, softmax, and convolution. The X-axis represents the kernels, and the Y-axis shows the execution time normalized by handwritten code. The values range from 0.00 to 1.00. The chart indicates that the execution time for all kernels is close to 1.00, suggesting high performance.]
Conclusion

- A compiler for a DNN accelerator that has hierarchical software-controlled scratchpads
  - generated optimized code for a wide range of input data parameters from a single hand-crafted kernel

- Future works
  - Add more optimization techniques to increase the performance and energy efficiency
  - Support more DNN operators
  - Support more DL frameworks in addition to TensorFlow