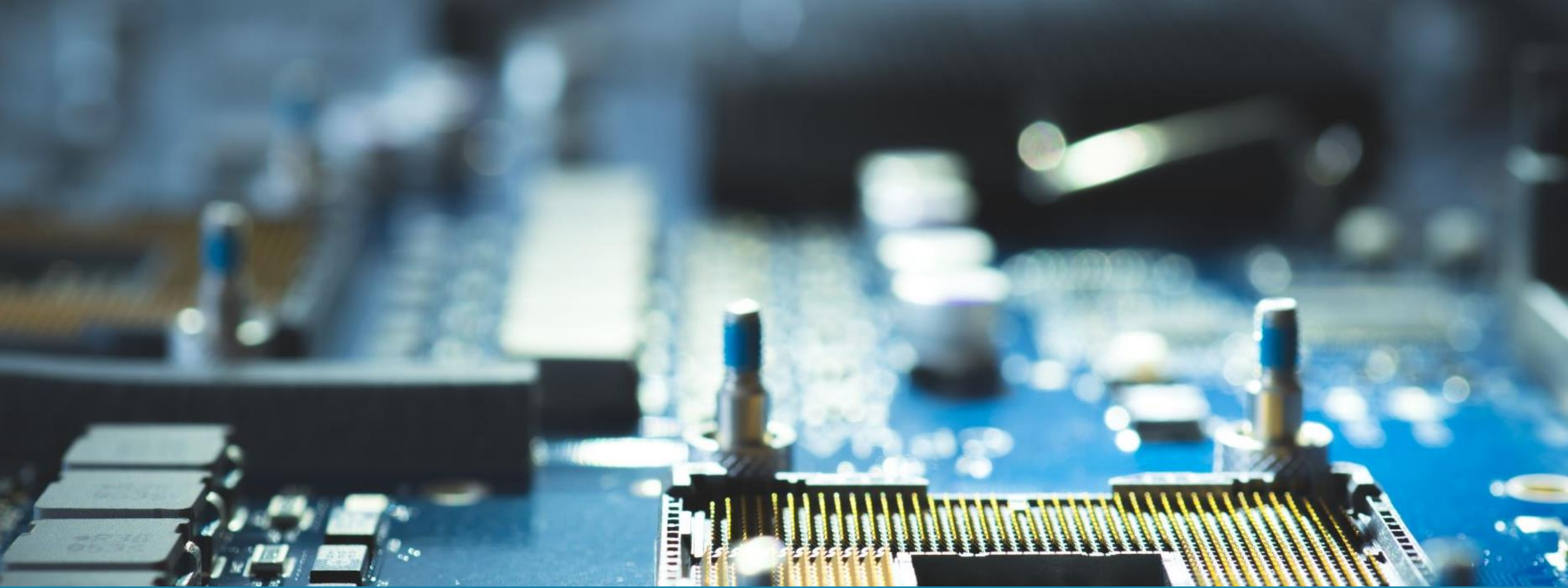


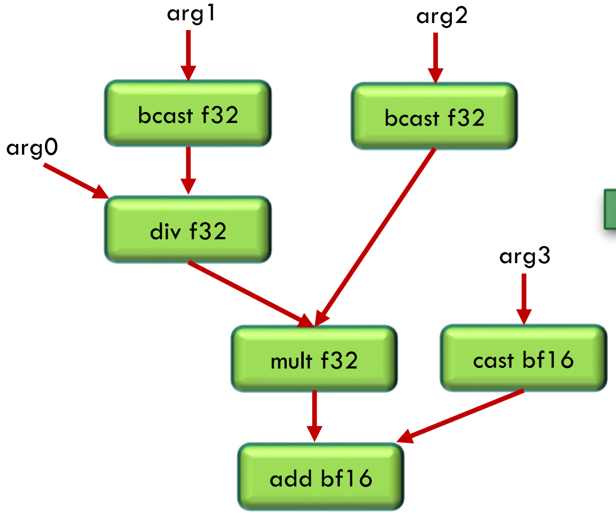
ML-BASED HARDWARE COST MODEL FOR HIGH-LEVEL MLIR

Dibyendu Das, Sandya Mannarswamy,

Intel

PROBLEM STATEMENT

- To predict machine/hardware characteristics from a high-level description of a dataflow graph (as shown below) via MLIR ops




```
func.func @main(%arg0: tensor<640x30522xf32>, %arg1: tensor<640x1xf32>,
                    %arg2: tensor<640x1xf32>, %arg3: tensor<640x30522xbf16>)
                    → tensor<640x30522xbf16> {
  %0 = xpu.broadcast %arg1 : tensor<640x1xf32> to tensor<640x30522xf32>
  %1 = xpu.div %arg0, %0 : tensor<640x30522xf32>
  %2 = xpu.broadcast %arg2 : tensor<640x1xf32> to tensor<640x30522xf32>
  %3 = xpu.mult %1, %2 : tensor<640x30522xf32>
  %4 = xpu.cast %3 : tensor<640x30522xf32> to tensor<640x30522xbf16>
  %5 = xpu.add %arg3, %4 : tensor<640x30522xbf16>
  return %5 : tensor<640x30522xbf16>
}
```

PROBLEM STATEMENT (CONTD…)

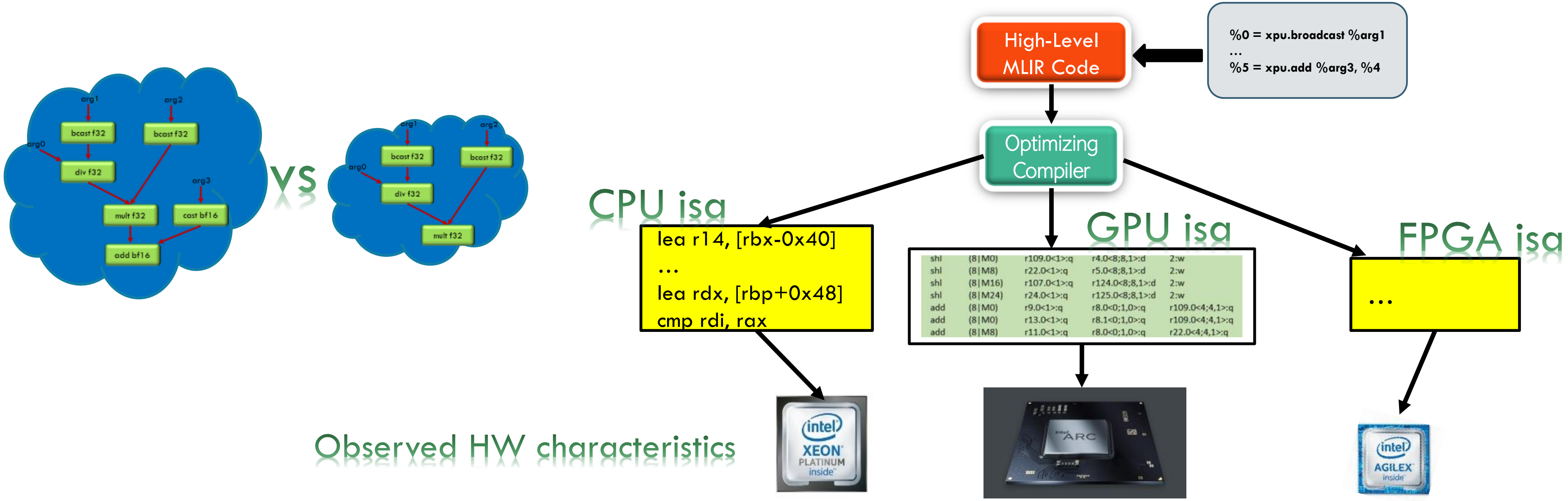
- The MLIR dialect used is a high-level proprietary dialect called xpu (which can be thought of at the same level as MHLO/TOSA)
- The output is either predicted XPU Utilization of the function OR register pressure/usage (this work)
- Can be extended to other HW characteristics – ex: total runtime, throughput …

```
func.func @main(%arg0: tensor<640x30522xf32>, %arg1: tensor<640x1xf32>,
                %arg2: tensor<640x1xf32>, %arg3: tensor<640x30522xbf16>)
                → tensor<640x30522xbf16> {
  %0 = xpu.broadcast %arg1 : tensor<640x1xf32> to tensor<640x30522xf32>
  %1 = xpu.div %arg0, %0 : tensor<640x30522xf32>
  %2 = xpu.broadcast %arg2 : tensor<640x1xf32> to tensor<640x30522xf32>
  %3 = xpu.mult %1, %2 : tensor<640x30522xf32>
  %4 = xpu.cast %3 : tensor<640x30522xf32> to tensor<640x30522xbf16>
  %5 = xpu.add %arg3, %4 : tensor<640x30522xbf16>
  return %5 : tensor<640x30522xbf16>
}
```

→ Reg Pressure/XPU utilization

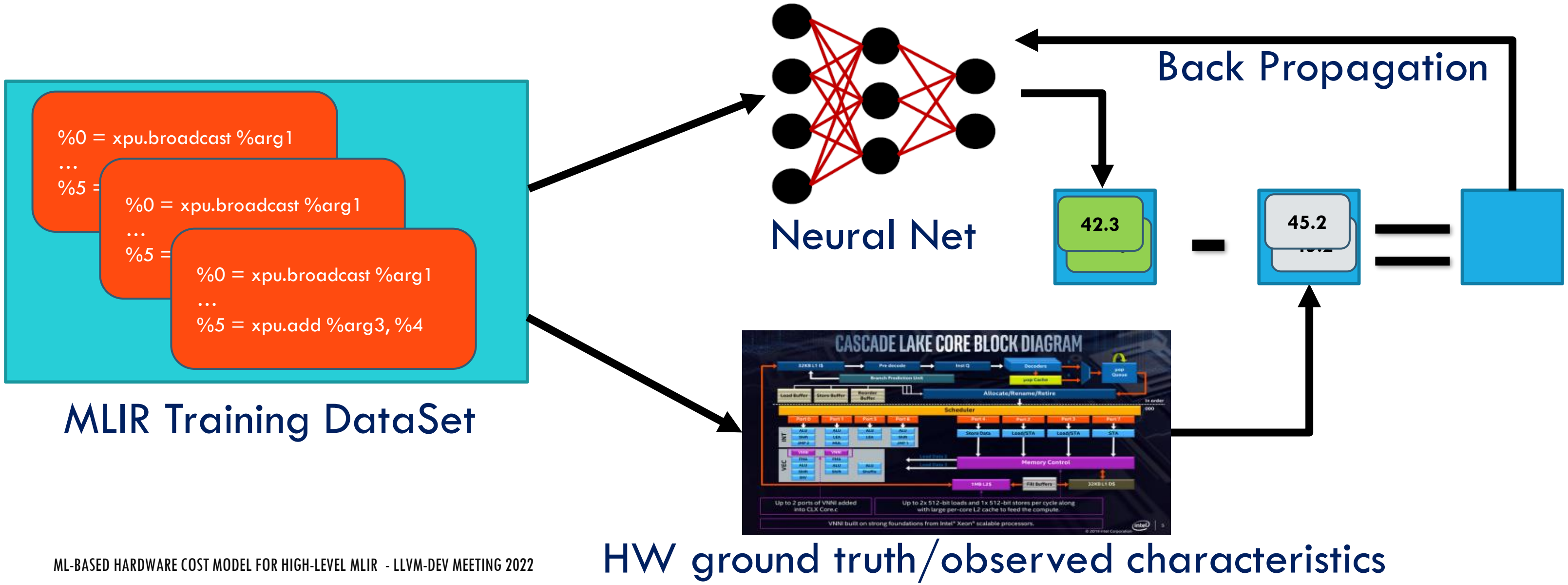
MOTIVATION

- Understand the pros- and cons- of picking one cluster over another during fusing
  - Can be used to drive efficiency in the clustering heuristic
- Estimate runtime/efficiency of a particular optimization vs another
- Predict these characteristics without actually running on hardware



OVERALL ML-DRIVEN ARCHITECTURE

- For training we ingest dataflow graphs in high-level MLIR which are tokenized
- Tokenized inputs fed to a NLP-like neural network
- Ground truth collected from assembly output, executable runs or simulators



TRAINING DATASET

- Given a set of graphs (MLIR level functions), we want to create a dataset which can be fed to an ML model for a target variable prediction
- Input is a sequence of xpu.ops and the input and output tensor shapes
- Create a CSV file, containing following columns
  - Filename
  - Full MLIR Text sequence
  - Input and output tensor shapes
  - XPU utilization or register pressure (target variable)
- Two kinds of CSVs created
  - Only the opcodes are used as a sequence (Example later)
  - The opcodes and the operands used as a sequence (Example later)
- Currently 20k+ MLIR files in the training set

INPUT REPRESENTATION FROM MLIR - TOKENIZATION

- The various pieces

```
func @main(  %arg0: tensor<64x128x1024xbf16>,
             %arg1: tensor<64x128x1024xbf16>,
             %arg2: tensor<64x128x1024xbf16>,
             %arg3: tensor<64x128x1024xbf16>) → tensor<64x128x1024xbf16>
{
  %0 = xpu.add %arg0, %arg1 : tensor<64x128x1024xbf16>
  %1 = xpu.add %0, %arg2 : tensor<64x128x1024xbf16>
  %2 = xpu.add %1, %arg3 : tensor<64x128x1024xbf16>
  return %2 : tensor<64x128x1024xbf16>
}
```

1

Input Tensor shapes –
['64x128x1024xbf16', '64x128x1024xbf16', '64x128x1024xbf16', '64x128x1024xbf16']

2

Output tensor shape -
['64x128x1024xbf16']

3

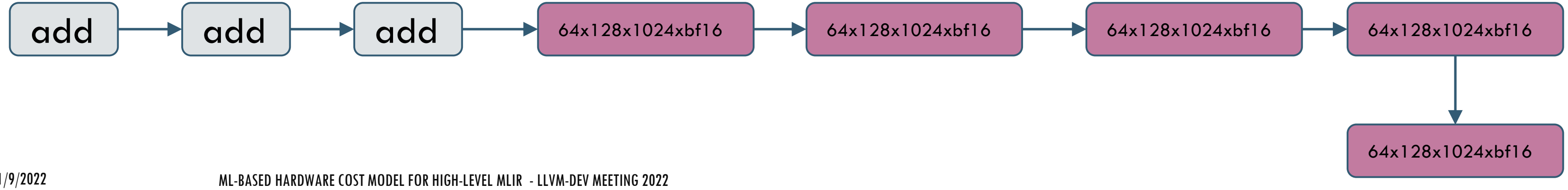
Op code sequence representation
%0 = add %arg0, %arg1 ## %1 = add %0, %arg2 ## %2 = add %1, %arg3 ## return %2 ##

Overall Input sequence to Model
Combination of tensor shapes and op code sequence

4

add add add **64x128x1024xbf16 64x128x1024xbf16 64x128x1024xbf16 64x128x1024xbf16 64x128x1024xbf16**

Sequence



INPUT REPRESENTATION FROM MLIR – ENCODING+EMBEDDING

- Encoding of the tokens
  - This creates a sparse vector representation
- Encoding should be followed by embedding
  - This creates a dense vector representation
  - Good for learning

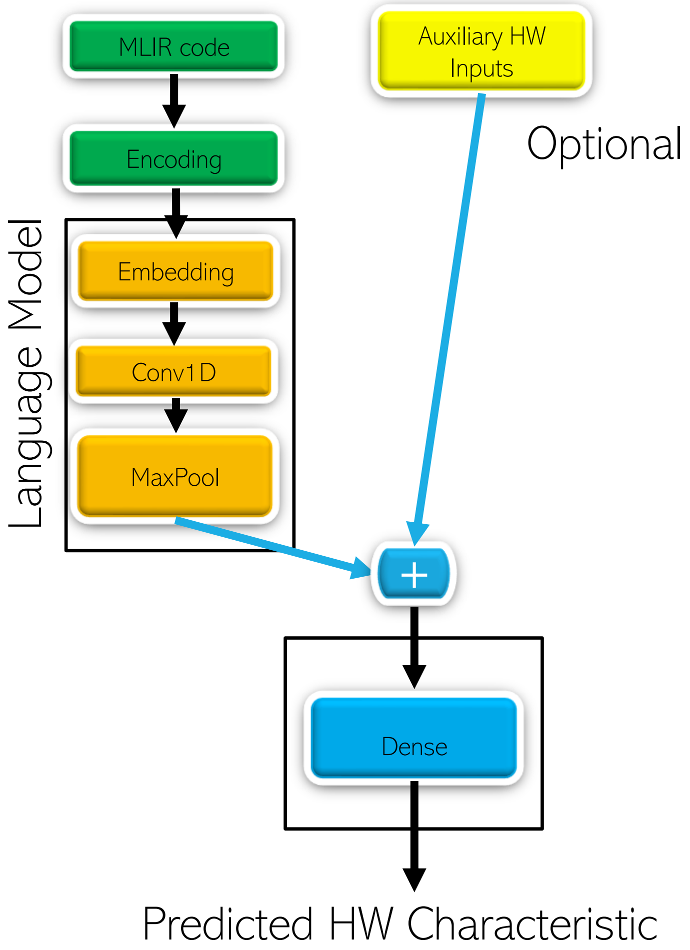
sub sub broadcast mult add sub sqrt mult sub div broadcast mult sqrt broadcast add div sub is ['3x3x64x64xf32', '3x3x64x64xf32', '1xf32', '1xf32', '1xf32', '1xf32', '1xf32', '1xf32', '3x3x64x64xf32', '1xf32', '3x3x64x64xf32', '3x3x64x64xf32']

Tokenized Sequence

[13 13 4 3 15 13 31 3 13 40 4 3 31 4 15 40 13 18 112 112 5 5 5 5 5 5 112 5 112 182 …]

MODELS

- Three models were tried:
  - Simple sequence of FC (Fully Connected) layers
    - Converged with higher RMSE (root mean square error)
  - LSTM
    - Better perf than FC
  - Conv1D+MaxPool followed by FC layers
    - Best performance so far with lowest RMSE

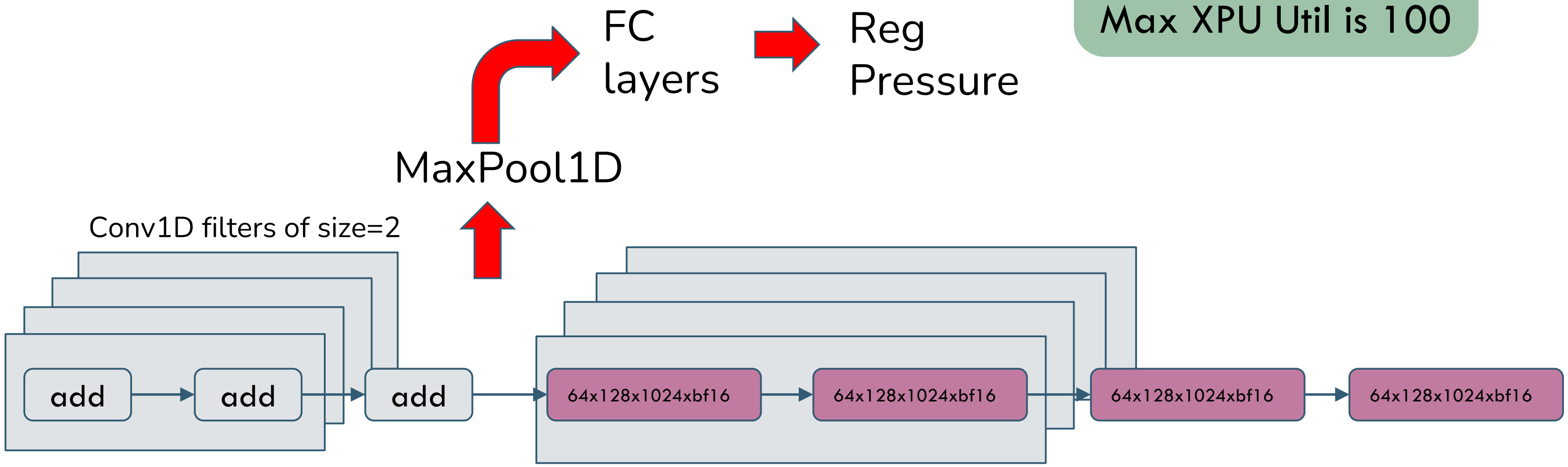


CONV1D + MAXPOOL + FC

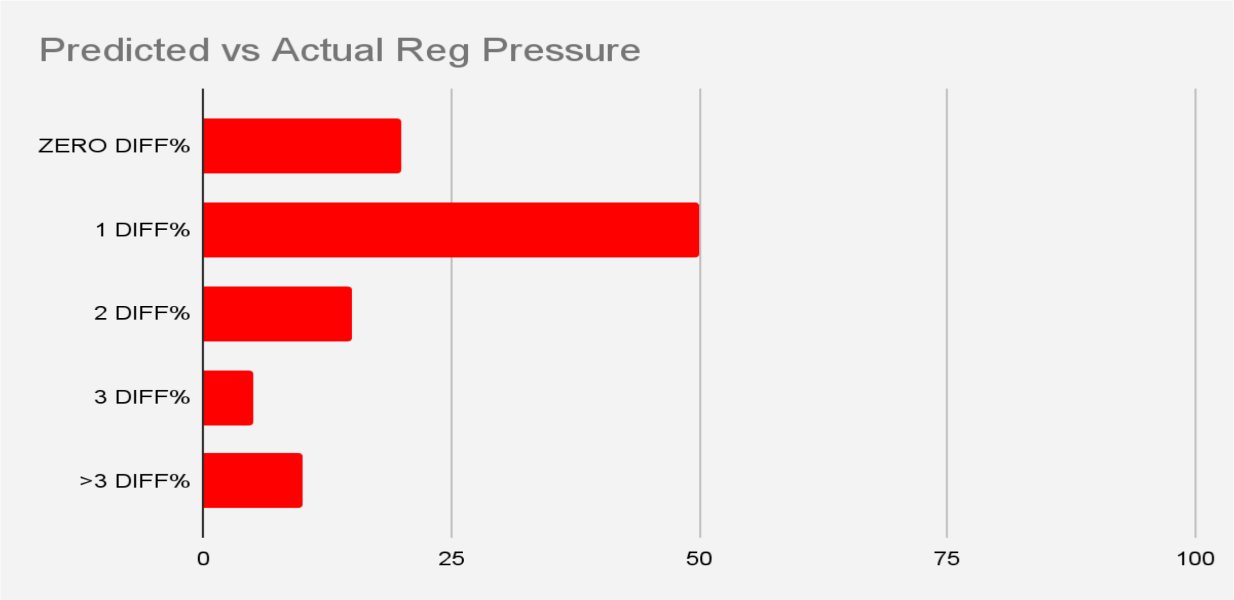
- 6 Conv1D layers of filter size=2
- A single MaxPool1D
- 3 FC layers
- Predict: Reg Pressure/XPU Util

RMSE: +/- 5%
Wrt Register Usage Prediction

RMSE: +/- 4%
Wrt XPU Utilization
Max XPU Util is 100



EXAMPLE OF SOME PREDICTIONS



MAPPING OPERATORS AND OPERANDS

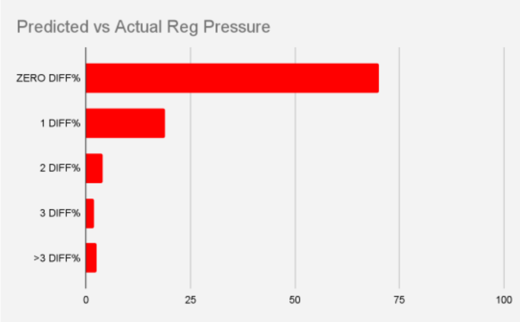
- If you add the operands to the operator sequence, we do get better prediction accuracy vs only-ops sequence
  - The sequences are on average 4x longer; training slower
  - Uses 6-deep Conv1D with fs=16,16,8,8,2,1
  - Unseen %argk or %k cause bad vector mapping (OOV)

%0 = add %arg0, %arg1 ## %1 = add %0, %arg2 ## %2 = add %1, %arg3 ## return %2 ##

Overall Input sequence to Model

Combination of tensor shapes and operator + operands sequence

%0 = add %arg0, %arg1 ## %1 = add %0, %arg2 ## %2 = add %1, %arg3 ## return %2 ## 64x128x1024xbf16 64x128x1024xbf16 64x128x1024xbf16 64x128x1024xbf16 64x128x1024xbf16



NEXT STEPS/DISCUSSIONS

- Run with larger training set
- Use other models like Transformer
- Integration in a production compiler
- Should we do pre-training for the MLIR sequence ?
- Extend similar prediction models for other MLIR dialects like affine, linalg, scf ?

REFERENCES

1. *A Learned Performance Model for Tensor Processing Units*, S. J. Kaufman et al. MLSys 2021.
2. *Towards Optimal VPU Compiler Cost Modeling by using Neural Networks to Infer Hardware Performances*, I. Hunter et al. arXiv, May 2022.
3. *End-to-end Deep Learning of Optimization Heuristics*, Chris Cummins et al. PACT 2017
4. *Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks*, Charith Mendis et al. ICML 2019.
5. *Neural Instruction Combiner*, Mannarswamy et al. ICLR DL4C Workshop, 2022 (Also as https://arxiv.org/pdf/2202.12379.pdf)