Integrating Convolution Optimization into MLIR to Improve Performance Beyond Im2Col+GEMM

Victor Ferrari
v187890@dac.unicamp.br

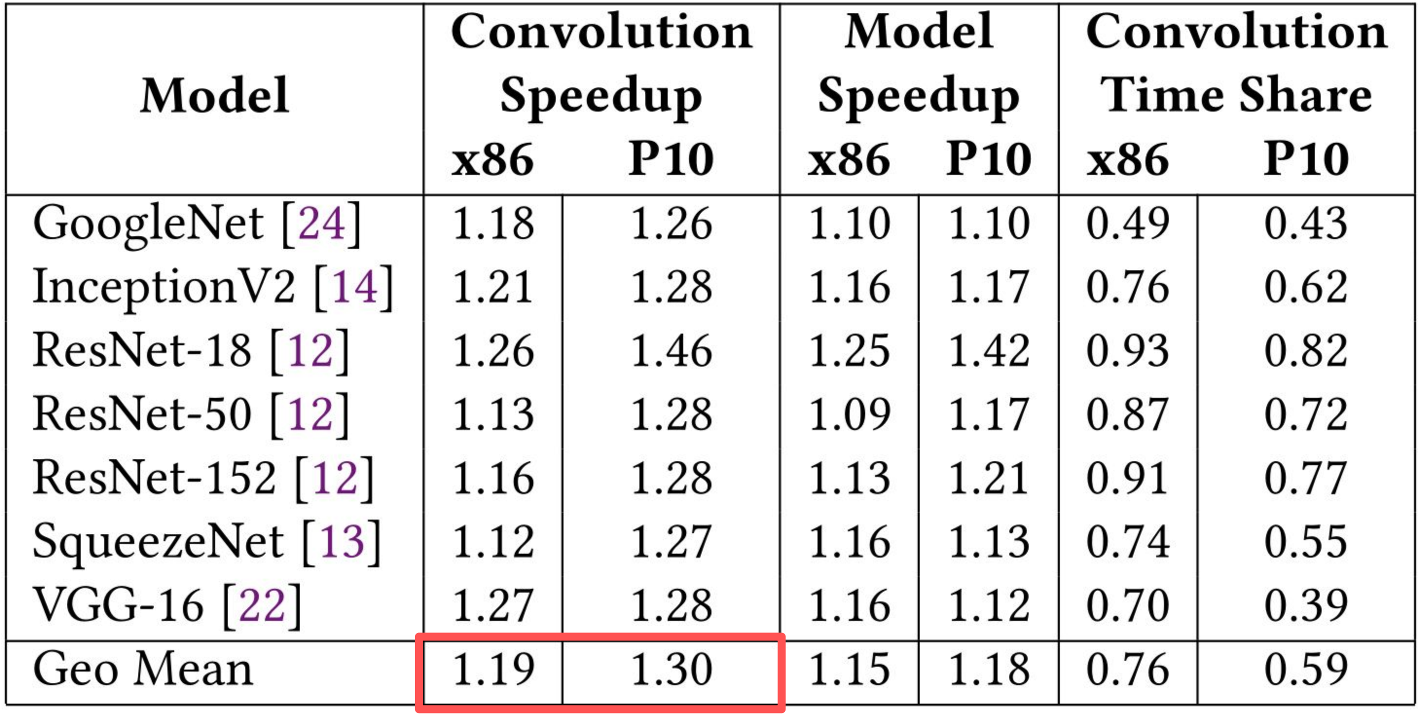
University of Campinas (Unicamp)
Institute of Computing (IC)
Computer Systems Laboratory (LSC)

MLIR Open Meeting, 2023-03-09

Agenda

- Motivation
- Optimizing Convolutions in MLIR
- Experimental Results
- Next Steps

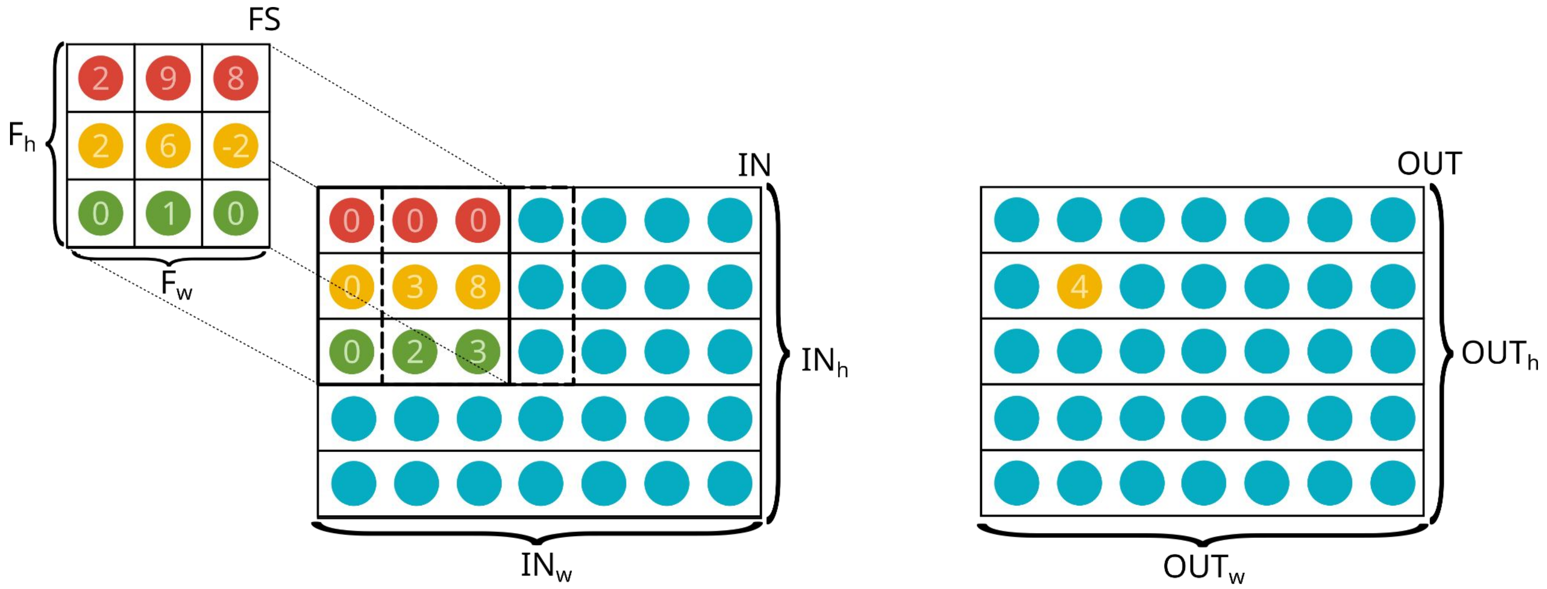
Model Inference Results



Model	Convolution Speedup		Model Speedup		Convolution Time Share	
	x86	P10	x86	P10	x86	P10
GoogleNet [24]	1.18	1.26	1.10	1.10	0.49	0.43
InceptionV2 [14]	1.21	1.28	1.16	1.17	0.76	0.62
ResNet-18 [12]	1.26	1.46	1.25	1.42	0.93	0.82
ResNet-50 [12]	1.13	1.28	1.09	1.17	0.87	0.72
ResNet-152 [12]	1.16	1.28	1.13	1.21	0.91	0.77
SqueezeNet [13]	1.12	1.27	1.16	1.13	0.74	0.55
VGG-16 [22]	1.27	1.28	1.16	1.12	0.70	0.39
Geo Mean	1.19	1.30	1.15	1.18	0.76	0.59

Motivation

Convolution



Optimizing Convolutions for CPUs

- Cache-Tiling Macro-Kernel
- Optimized Micro-Kernel
- Fast Packing
- Similar to how BLAS handles matrix-multiplications

Optimizing Convolutions in MLIR

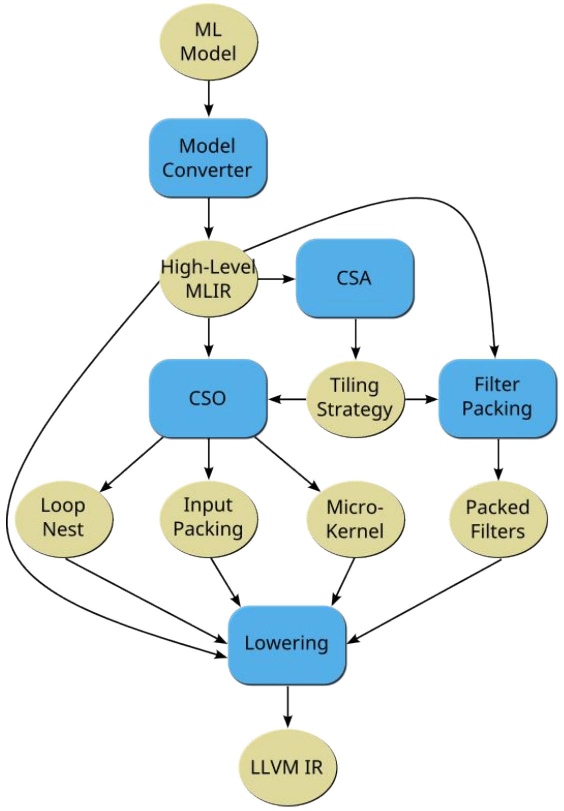
Optimizing Convolutions in MLIR

- 1 **analysis** pass
  - Convolution Slicing Analysis (CSA): responsible for calculating tile size and distribution (Tiling Strategy).

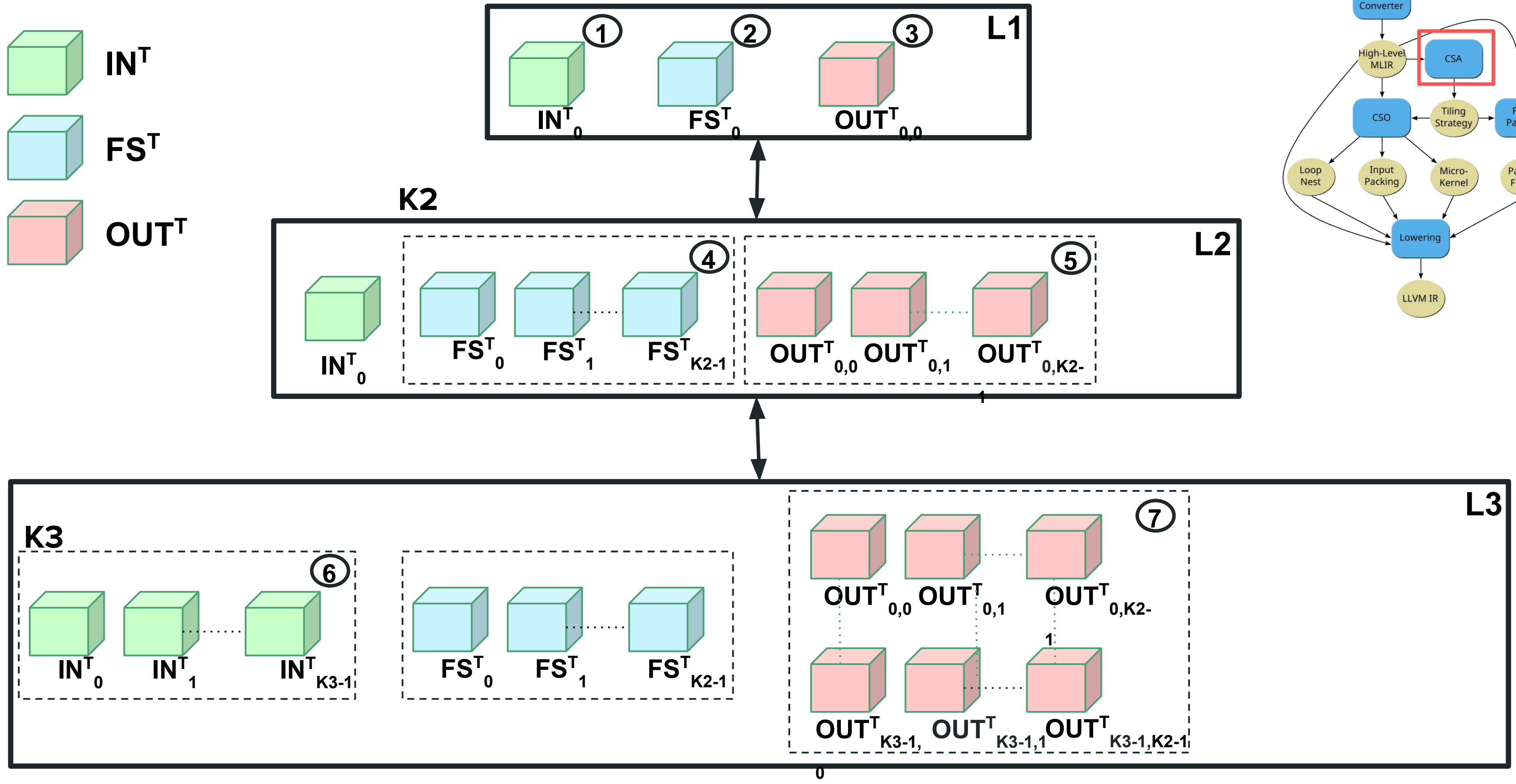
- 1 **data transformation** pass*
  - Filter Packing: **if the conv. filters are static**, they can be pre-packed for the computation at compile time.

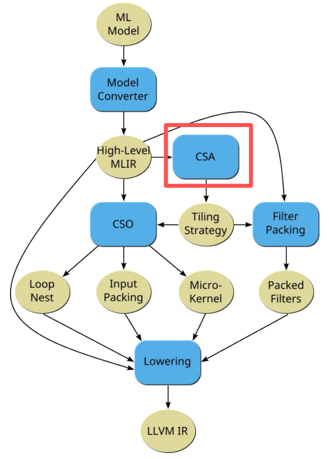
- 1 **lowering** pass
  - Convolution Slicing Optimization (CSO): responsible for lowering a conv. operation to an optimized implementation using CSA's tiling strategy.

Current Compilation Flow



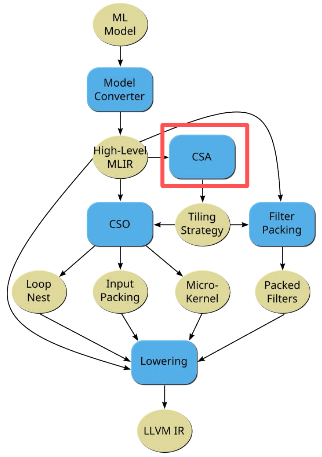
Convolution Slicing Analysis (CSA)



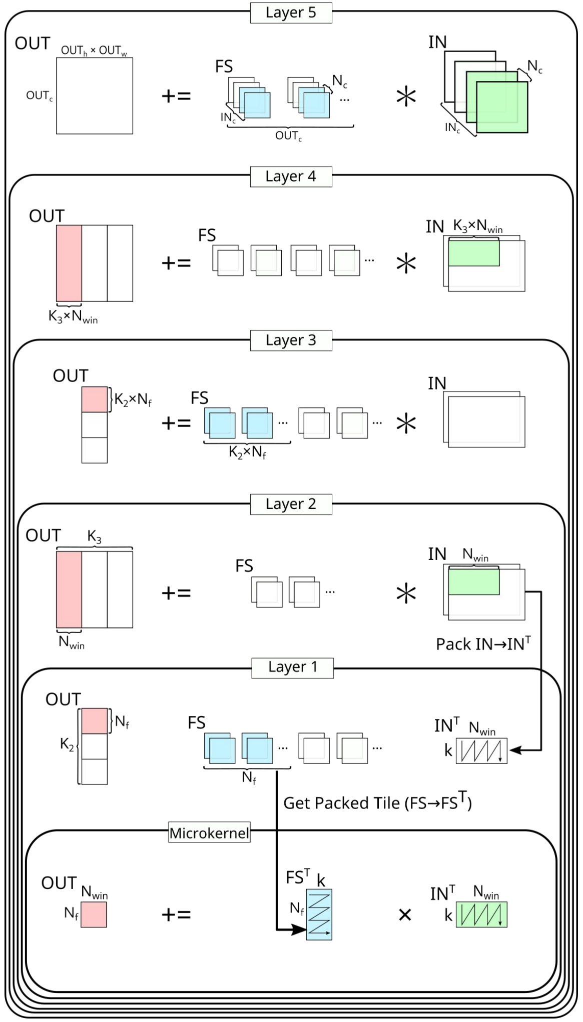


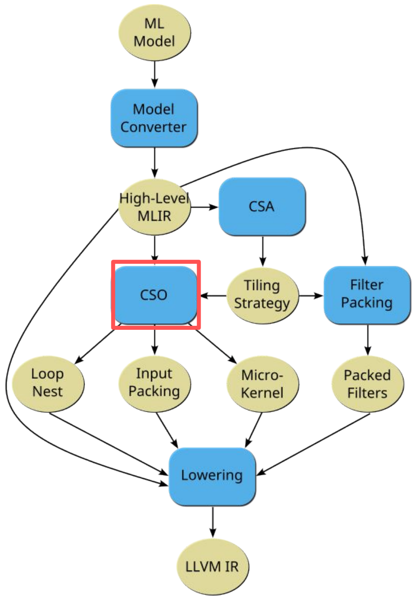
Convolution Slicing Analysis (CSA)

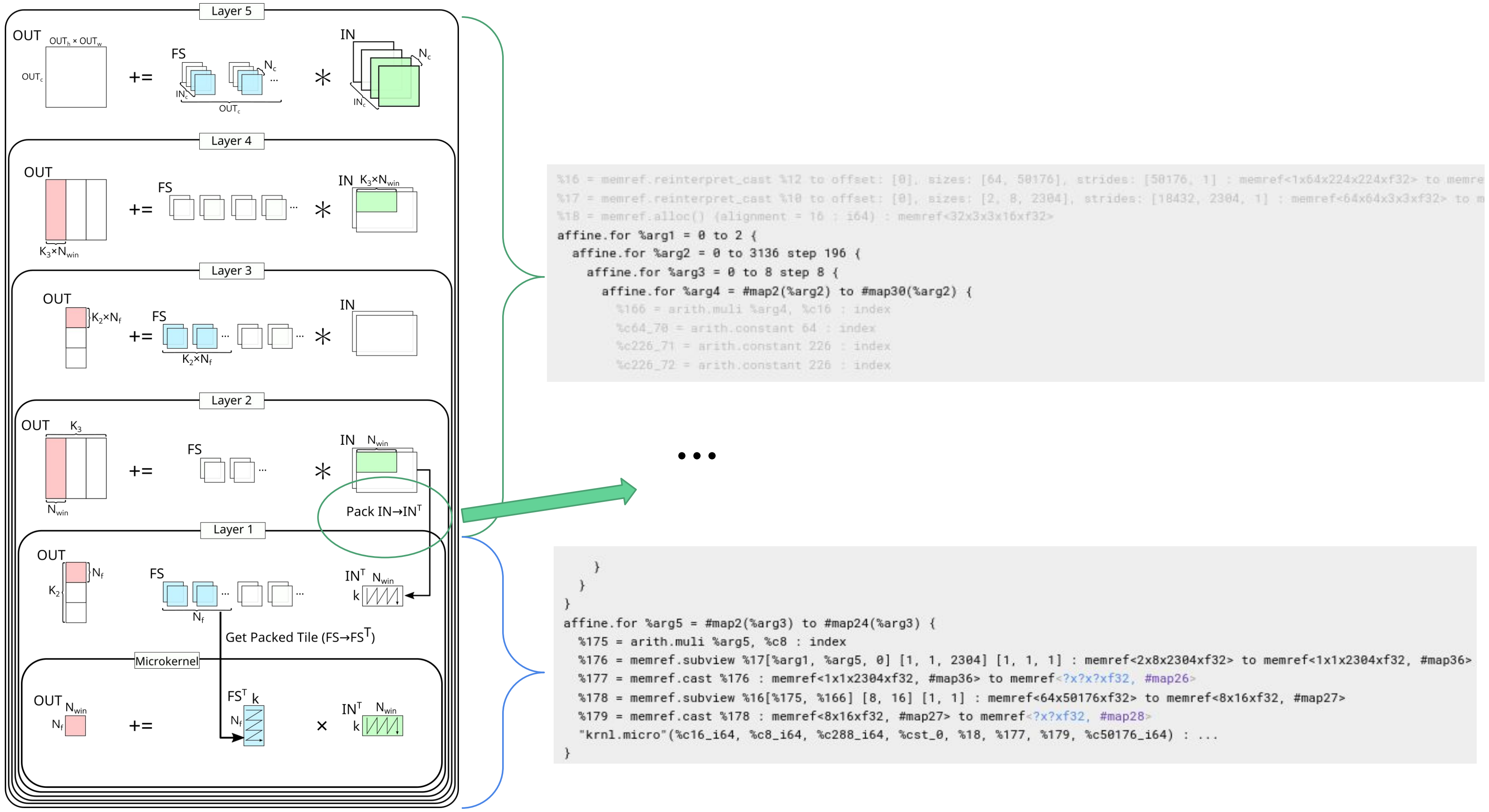
- Tiling on the **channel** dimension
- Tile size and distribution determined by **heuristic**
- Scheduling order determined by **cost model**
    - Takes DRAM bursts into consideration



Convolution Slicing Optimization (CSO)



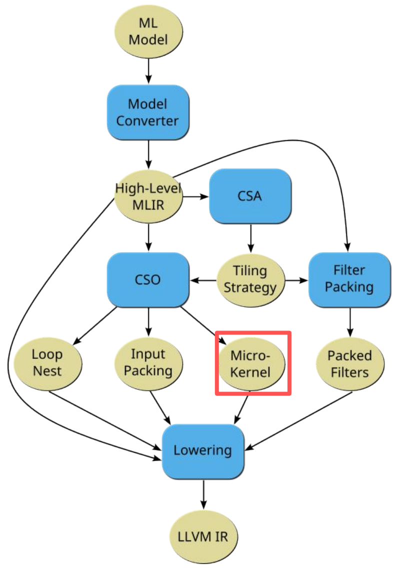




```
%16 = memref.reinterpret_cast %12 to offset: [0], sizes: [64, 50176], strides: [50176, 1] : memref<1x64x224x224xf32> to memre
%17 = memref.reinterpret_cast %10 to offset: [0], sizes: [2, 8, 2304], strides: [18432, 2304, 1] : memref<64x64x3x3xf32> to m
%18 = memref.alloc() {alignment = 16 : i64} : memref<32x3x3x16xf32>
affine.for %arg1 = 0 to 2 {
  affine.for %arg2 = 0 to 3136 step 196 {
    affine.for %arg3 = 0 to 8 step 8 {
      affine.for %arg4 = #map2(%arg2) to #map30(%arg2) {
        %166 = arith.muli %arg4, %c16 : index
        %c64_70 = arith.constant 64 : index
        %c226_71 = arith.constant 226 : index
        %c226_72 = arith.constant 226 : index
```

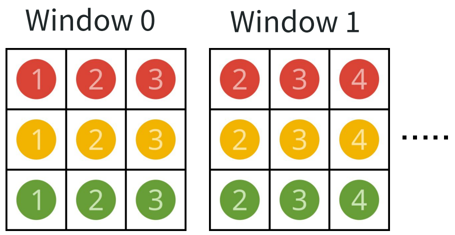
```
      }
    }
  }
affine.for %arg5 = #map2(%arg3) to #map24(%arg3) {
  %175 = arith.muli %arg5, %c8 : index
  %176 = memref.subview %17[%arg1, %arg5, 0] [1, 1, 2304] [1, 1, 1] : memref<2x8x2304xf32> to memref<1x1x2304xf32, #map36>
  %177 = memref.cast %176 : memref<1x1x2304xf32, #map36> to memref<?x?x?xf32, #map26>
  %178 = memref.subview %16[%175, %166] [8, 16] [1, 1] : memref<64x50176xf32> to memref<8x16xf32, #map27>
  %179 = memref.cast %178 : memref<8x16xf32, #map27> to memref<?x?xf32, #map28>
  "krnl.micro"(%c16_i64, %c8_i64, %c288_i64, %cst_0, %18, %177, %179, %c50176_i64) : ...
}
```

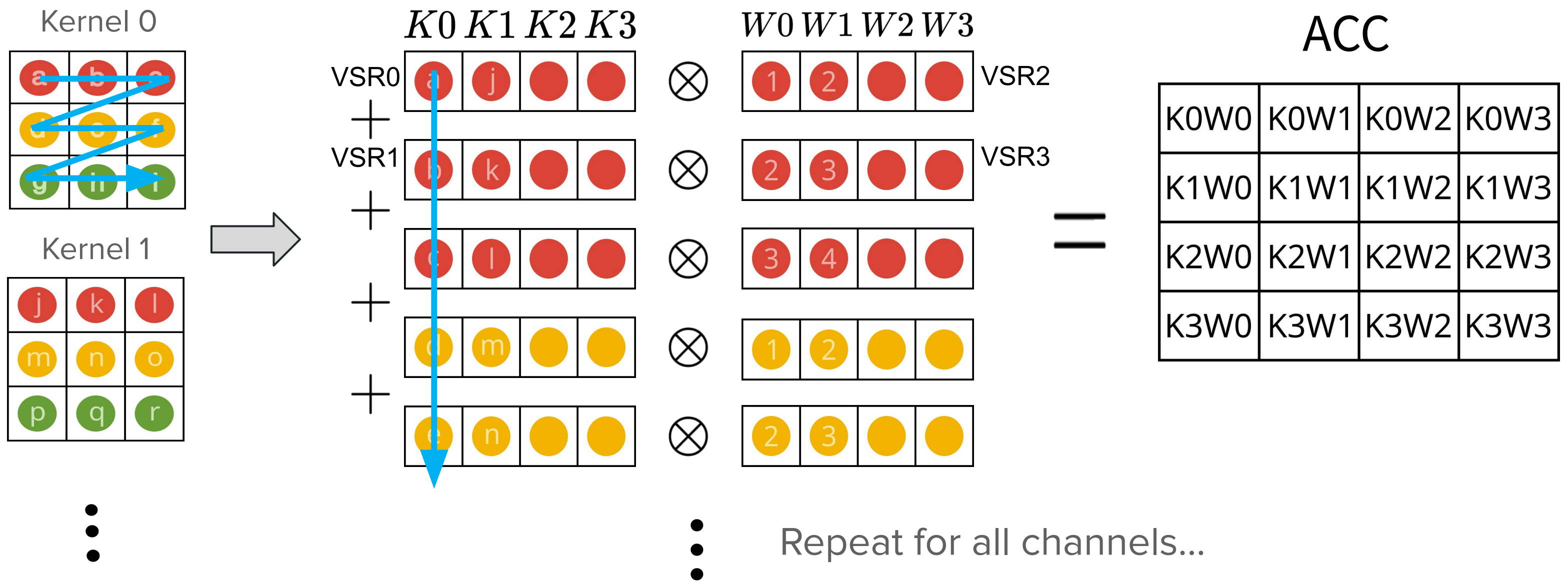
Optimized Micro-Kernel



- We use an **outer-product** micro-kernel
- Can be the same as matrix-multiplication
- Next step: move micro-kernel to one of these alternatives (feedback please!)
  - `llvm.matrix.multiply.*` intrinsic
  - MLIR-centric alternative

Micro-Kernel: access pattern

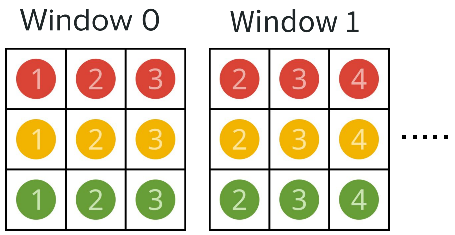


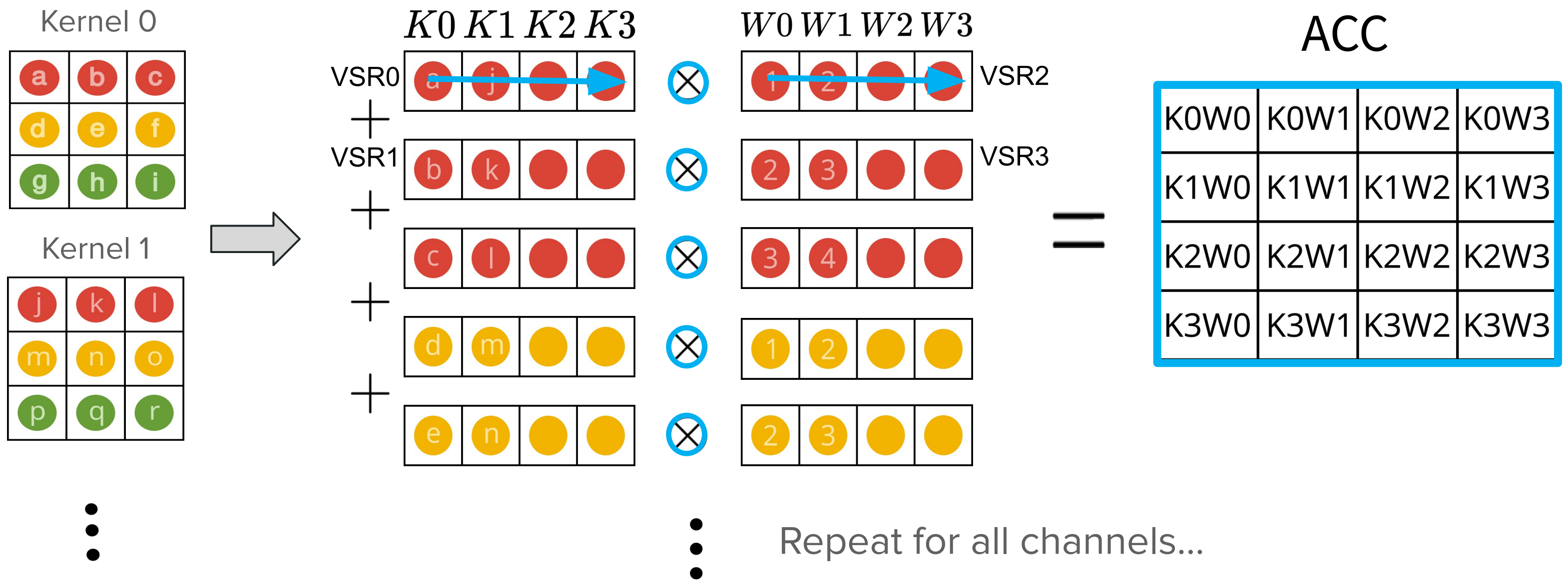


K0W0	K0W1	K0W2	K0W3
K1W0	K1W1	K1W2	K1W3
K2W0	K2W1	K2W2	K2W3
K3W0	K3W1	K3W2	K3W3

Repeat for all channels...

Micro-Kernel: outer product

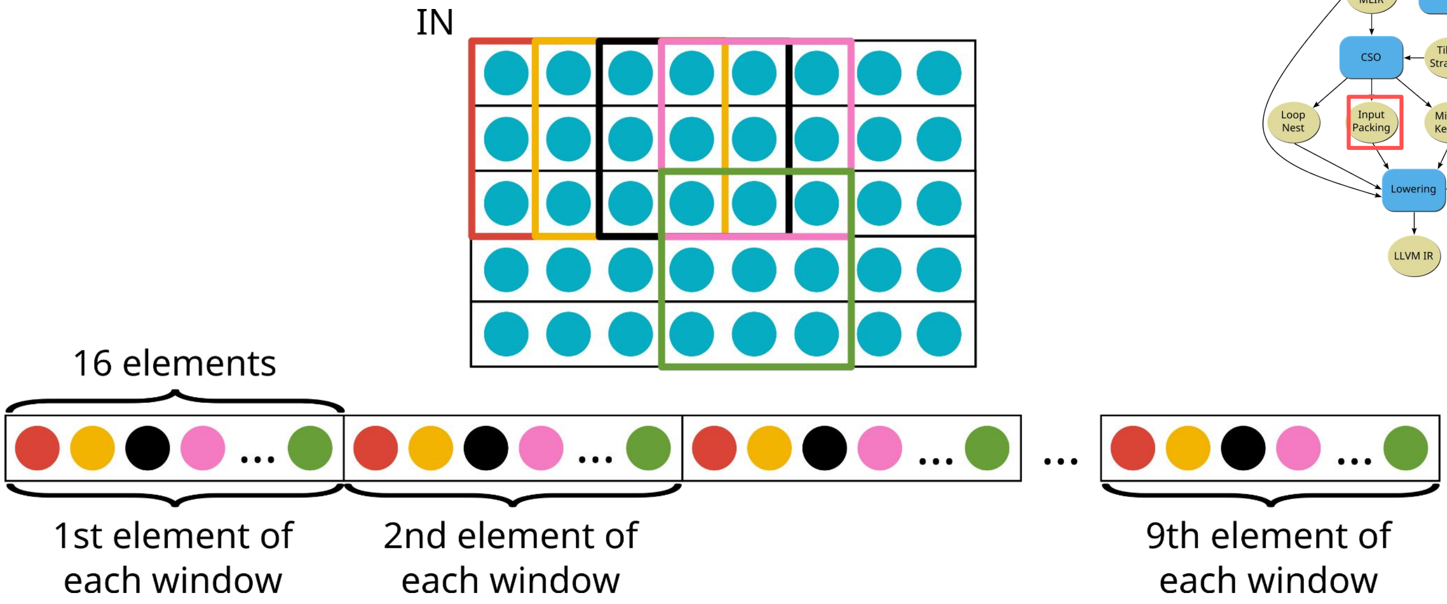


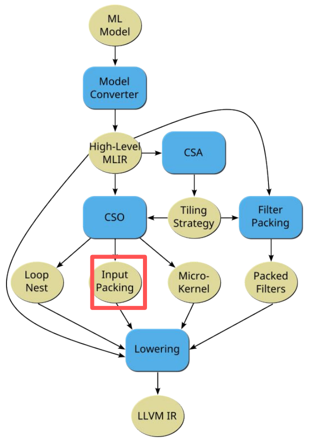


K0W0	K0W1	K0W2	K0W3
K1W0	K1W1	K1W2	K1W3
K2W0	K2W1	K2W2	K2W3
K3W0	K3W1	K3W2	K3W3

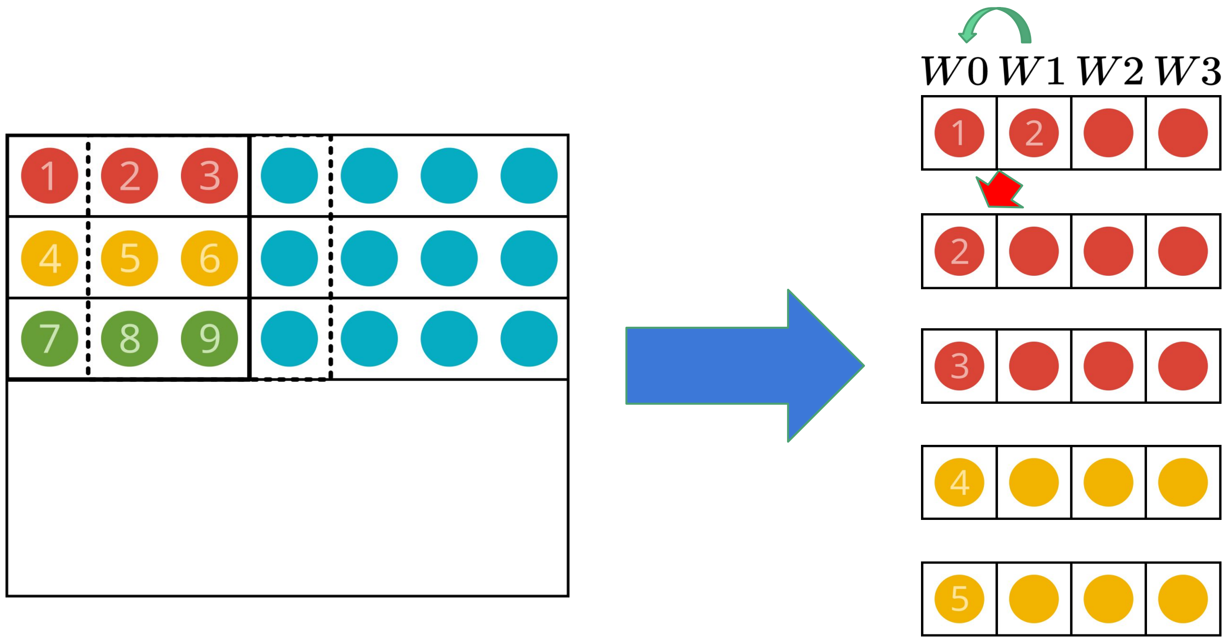
Repeat for all channels...

Packing

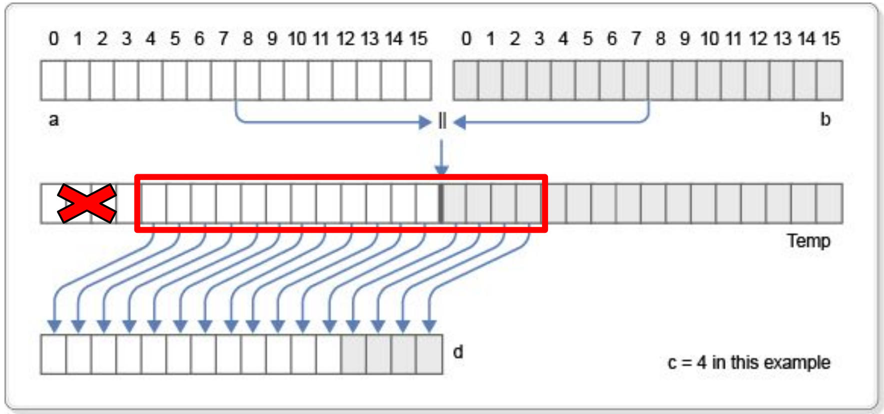

IN
16 elements
...
...
...
...
...
1st element of each window
2nd element of each window
9th element of each window


ML Model
Model Converter
High-Level MLIR
CSA
CSO
Tiling Strategy
Filter Packing
Loop Nest
Input Packing
Micro-Kernel
Packed Filters
Lowering
LLVM IR

Useful Observation



Vector-Based Packing (using the Vector dialect)



```
%c16_93 = arith.constant 16 : index
%190 = vector.load %184[%arg5, %arg6, %c16_93] : memref<32x226x226xf32>, vector<4xf32>
%c1_94 = arith.constant 1 : index
%c0_95 = arith.constant 0 : index
%191 = vector.shuffle %186, %187 [1, 2, 3, 4] : vector<4xf32>, vector<4xf32>
vector.store %191, %18[%arg5, %arg6, %c1_94, %c0_95] : memref<32x3x3x16xf32>, vector<4xf32>
%c4_96 = arith.constant 4 : index
%192 = vector.shuffle %187, %188 [1, 2, 3, 4] : vector<4xf32>, vector<4xf32>
vector.store %192, %18[%arg5, %arg6, %c1_94, %c4_96] : memref<32x3x3x16xf32>, vector<4xf32>
%c8_97 = arith.constant 8 : index
%193 = vector.shuffle %188, %189 [1, 2, 3, 4] : vector<4xf32>, vector<4xf32>
vector.store %193, %18[%arg5, %arg6, %c1_94, %c8_97] : memref<32x3x3x16xf32>, vector<4xf32>
```

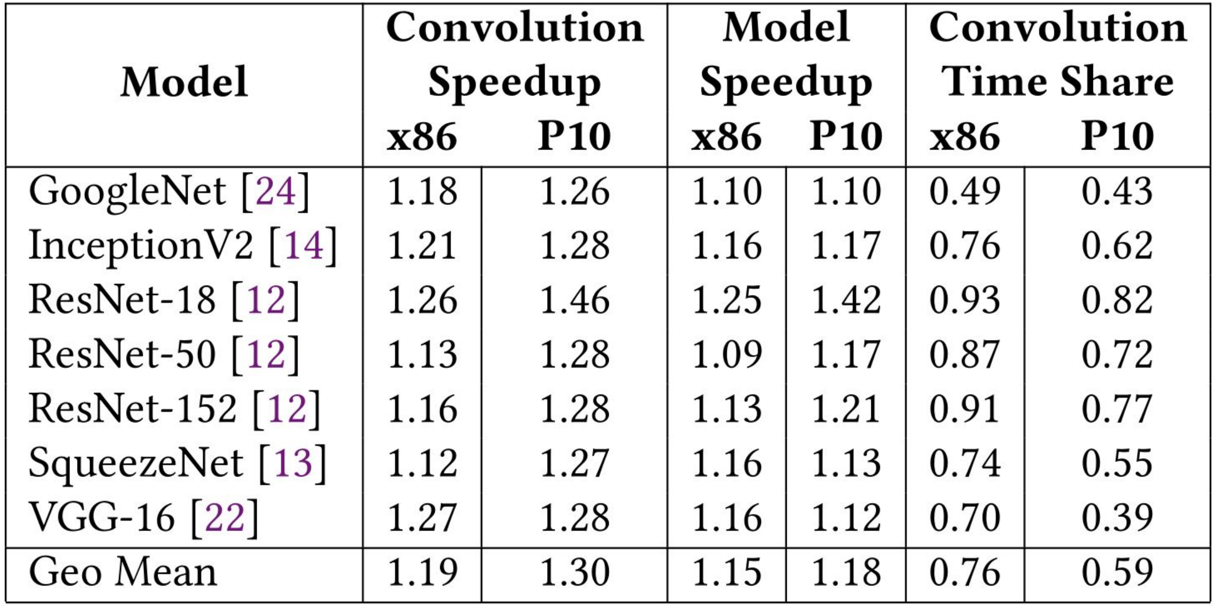
Experimental Results

Experimental Setup

- **IBM POWER10** E1050 CPU @ 4.00 GHz
  - Single-threaded **single-precision** computations peak at 256 GFLOPS/s
  - MMA Matrix Engine: 2 outer-products + accumulation per cycle
- **Intel Xeon** Silver 4208 CPU @ 2.10 GHz
  - Single-threaded **single-precision** computations peak at 67.2 GFLOPS/s
  - AVX-512 SIMD: 1 fma operation per cycle, 2 operations on 16 f32

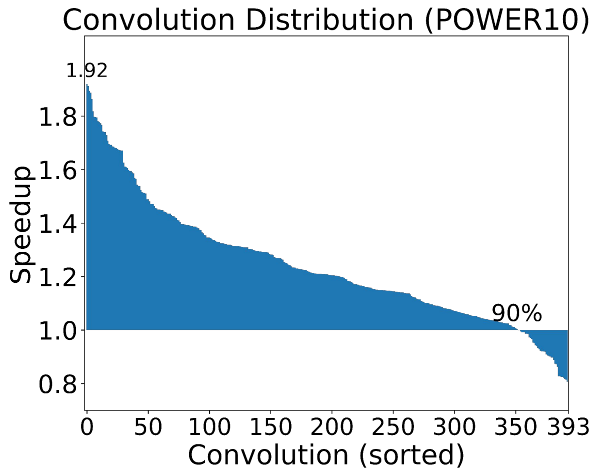
- **100 measurements**, with low variability
- ONNX-MLIR environment
  - Full machine-learning model inference
- Measured against Im2Col + OpenBLAS GEMM

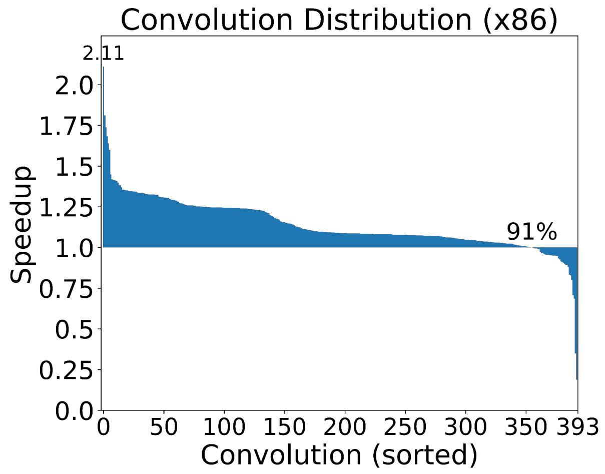
Model Inference Results



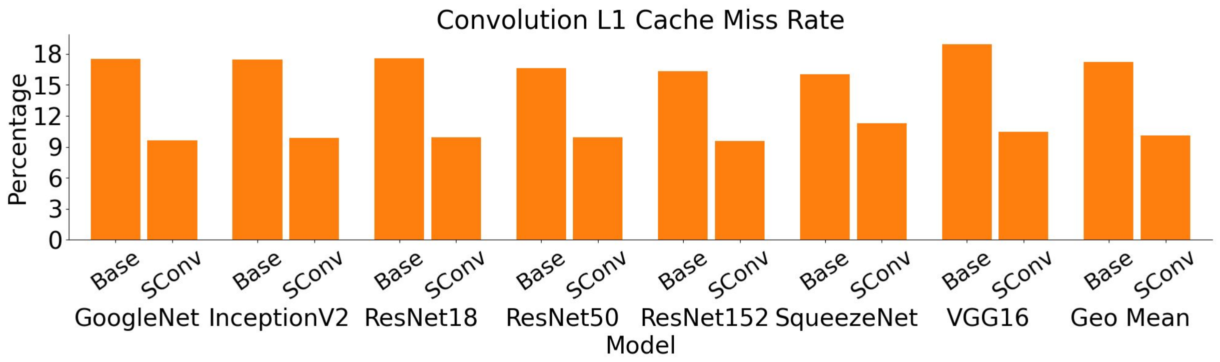
Model	Convolution Speedup		Model Speedup		Convolution Time Share	
	x86	P10	x86	P10	x86	P10
GoogleNet [24]	1.18	1.26	1.10	1.10	0.49	0.43
InceptionV2 [14]	1.21	1.28	1.16	1.17	0.76	0.62
ResNet-18 [12]	1.26	1.46	1.25	1.42	0.93	0.82
ResNet-50 [12]	1.13	1.28	1.09	1.17	0.87	0.72
ResNet-152 [12]	1.16	1.28	1.13	1.21	0.91	0.77
SqueezeNet [13]	1.12	1.27	1.16	1.13	0.74	0.55
VGG-16 [22]	1.27	1.28	1.16	1.12	0.70	0.39
Geo Mean	1.19	1.30	1.15	1.18	0.76	0.59

Individual Convolution Result

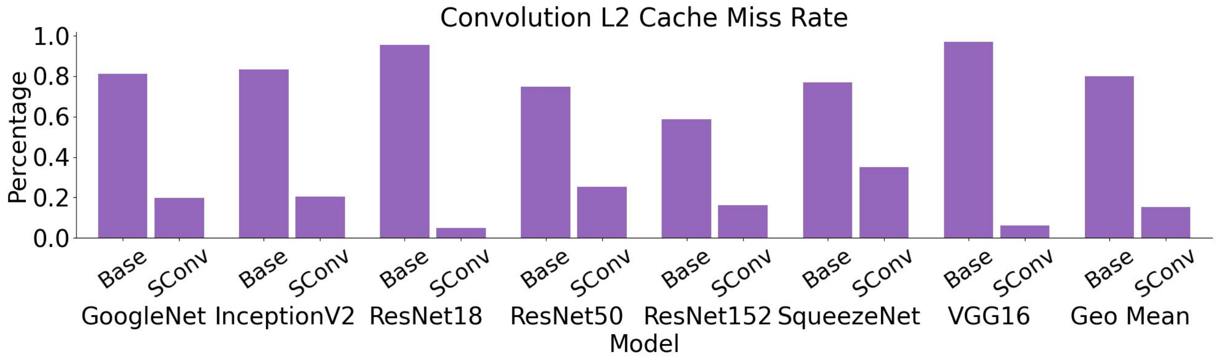




Why it Works

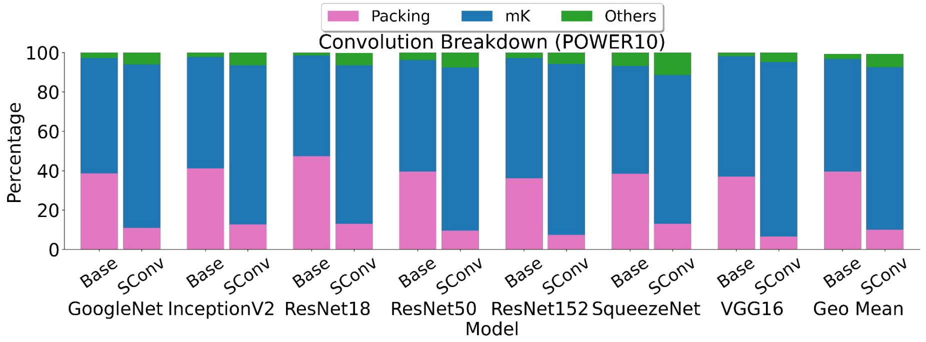


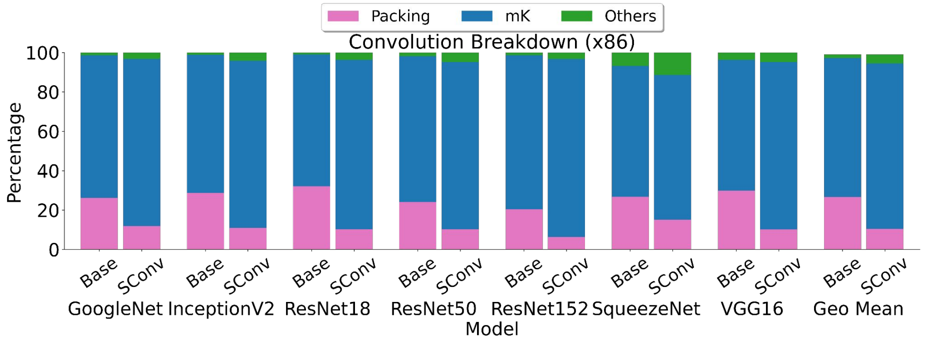
Convolution L1 Cache Miss Rate
Percentage
0
3
6
9
12
15
18
Base
SConv
GoogleNet
InceptionV2
ResNet18
ResNet50
ResNet152
SqueezeNet
VGG16
Geo Mean
Model



Convolution L2 Cache Miss Rate
Percentage
0.0
0.2
0.4
0.6
0.8
1.0
Base
SConv
GoogleNet
InceptionV2
ResNet18
ResNet50
ResNet152
SqueezeNet
VGG16
Geo Mean
Model

Packing Improvements





Next Steps

We want to contribute!

Next Steps (Micro-Kernel Lowering)

- Today: implemented as a separate library

- Next: suggestions?
  - `llvm.matrix.multiply.*` intrinsic
  - MLIR-centric alternative

Next Steps (Macro-Kernel Lowering)

- Today: lowered from ONNX Dialect to
  - Affine
  - Memref
  - Vector
  - Arith

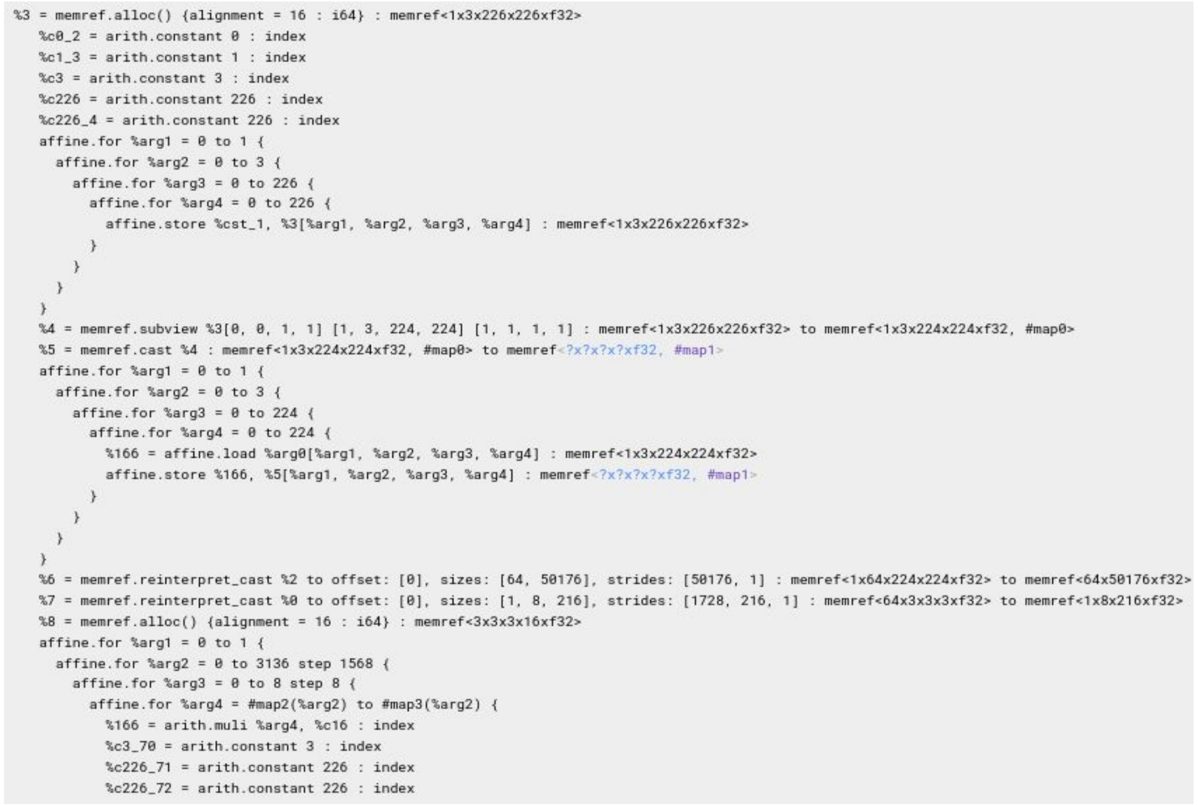
- Next: what are the suggestions?
  - Linalg?
  - Torch-MLIR?
  - TOSA (not NCHW) ✖
  - ...

More Information

- V. Ferrari, R. Sousa, M. Pereira, J. P. L. de Carvalho, J. N. Amaral, J. Moreira, G. Araujo. *Advancing Direct Convolution using Convolution Slicing Optimization and ISA Extensions*. 2023.
    - https://arxiv.org/abs/2303.04739

- R. Sousa, B. Jung, J. Kwak, M. Frank, G. Araujo. *Efficient Tensor Slicing for Multicore NPUs using Memory Burst Modeling*. 2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)
    - https://ieeexplore.ieee.org/document/9651605

A little bit of code navigation



```
%3 = memref.alloc() {alignment = 16 : i64} : memref<1x3x226x226xf32>
   %c0_2 = arith.constant 0 : index
   %c1_3 = arith.constant 1 : index
   %c3 = arith.constant 3 : index
   %c226 = arith.constant 226 : index
   %c226_4 = arith.constant 226 : index
   affine.for %arg1 = 0 to 1 {
     affine.for %arg2 = 0 to 3 {
       affine.for %arg3 = 0 to 226 {
         affine.for %arg4 = 0 to 226 {
           affine.store %cst_1, %3[%arg1, %arg2, %arg3, %arg4] : memref<1x3x226x226xf32>
         }
       }
     }
   }
   %4 = memref.subview %3[0, 0, 1, 1] [1, 3, 224, 224] [1, 1, 1, 1] : memref<1x3x226x226xf32> to memref<1x3x224x224xf32, #map0>
   %5 = memref.cast %4 : memref<1x3x224x224xf32, #map0> to memref<?x?x?x?xf32, #map1>
   affine.for %arg1 = 0 to 1 {
     affine.for %arg2 = 0 to 3 {
       affine.for %arg3 = 0 to 224 {
         affine.for %arg4 = 0 to 224 {
           %166 = affine.load %arg0[%arg1, %arg2, %arg3, %arg4] : memref<1x3x224x224xf32>
           affine.store %166, %5[%arg1, %arg2, %arg3, %arg4] : memref<?x?x?x?xf32, #map1>
         }
       }
     }
   }
   %6 = memref.reinterpret_cast %2 to offset: [0], sizes: [64, 50176], strides: [50176, 1] : memref<1x64x224x224xf32> to memref<64x50176xf32>
   %7 = memref.reinterpret_cast %0 to offset: [0], sizes: [1, 8, 216], strides: [1728, 216, 1] : memref<64x3x3x3xf32> to memref<1x8x216xf32>
   %8 = memref.alloc() {alignment = 16 : i64} : memref<3x3x3x16xf32>
   affine.for %arg1 = 0 to 1 {
     affine.for %arg2 = 0 to 3136 step 1568 {
       affine.for %arg3 = 0 to 8 step 8 {
         affine.for %arg4 = #map2(%arg2) to #map3(%arg2) {
           %166 = arith.muli %arg4, %c16 : index
           %c3_70 = arith.constant 3 : index
           %c226_71 = arith.constant 226 : index
           %c226_72 = arith.constant 226 : index
```

Thank You!

UNICAMP