

# Gemmini @ MLSys 2022

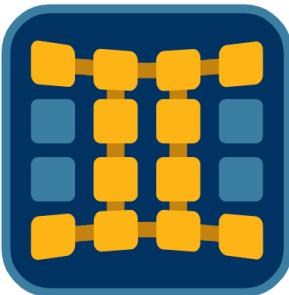
Hasan Genc, Simon Guo, Seah Kim, Vadim (Dima) Nikiforov



Berkeley  
Architecture  
Research

# Acknowledgements

This research was, in part, funded by the U.S. Government under the DARPA RTML program (contract FA8650-20-2-7006). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.



# GEMINI

# Outline

- **Session 0:** Setup
- **Session 1:** Motivation and Architectural Template
- **Session 2:** Gemmini's Programming Model
- **Session 3:** Performance Profiling on Gemmini
- **Session 4:** Parting Thoughts

# Session 0: Setup

# AWS Instances

- Go to the link below for information about connecting to AWS instances
  - Cntl-F for “Part 0 Setup”

[tinyurl.com/gemmini-mlsys-instructions](https://tinyurl.com/gemmini-mlsys-instructions)



# Docker

```
# How to log in to Docker. Recommend doing this in  
# tmux or screen.
```

```
sudo docker run -it --privileged  
simonguozi@berkeley/gemmini_mlsys_22:v1.10 bash
```

# Warmup Your Instance

```
# Run these commands to warmup your instance, so that future  
# commands run faster.  
  
cd /root/chipyard/generators/gemmini  
  
bash tutorial/warmup.sh  
  
# View all instructions here:  
# tinyurl.com/gemmini-mlsys-instructions
```

# Session 1: Motivation and Gemmini Architectural Template

# Outline

- Introduction
- Gemmini Architectural Template
  - Gemmini and Architectural Template
  - SoC and System-Level Parameters
- *Interactive Activities:*
  - Build your own Gemmini
  - Add a New Datatype to Gemmini

# Introduction: Why did we build Gemmini?

# DNNs are exploding in popularity...



*Matt Christenson/BLM/2017*



*By Dllu - Own work, CC BY-SA 4.0,  
<https://commons.wikimedia.org/w/index.php?curid=64517567>*

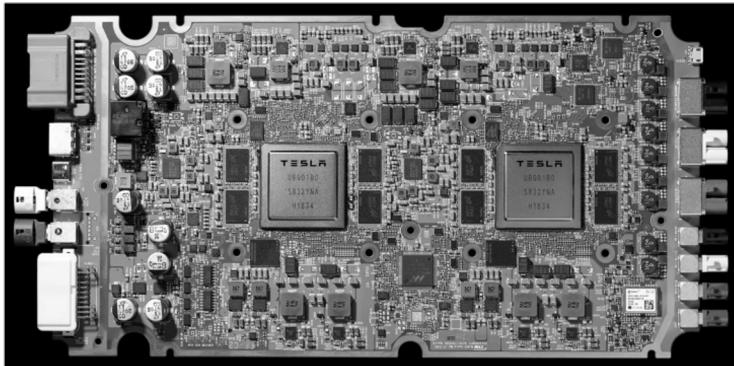


*Apple Support*

# Which means DNN accelerators are exploding in popularity...



*Edge TPU*



*Tesla FSD*

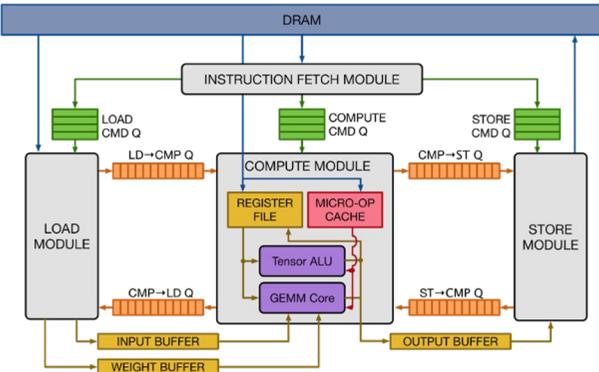


*Cloud TPU*

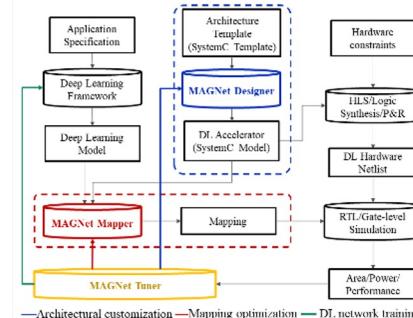
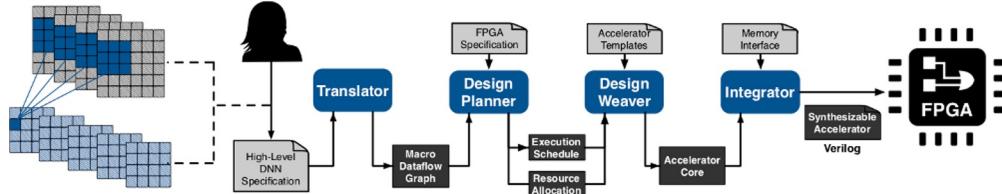
# Which means DNN accelerator generators are exploding in popularity...



VTA



DNNWeaver

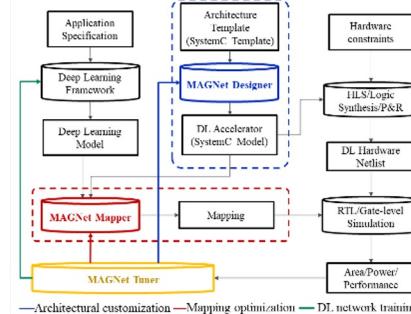
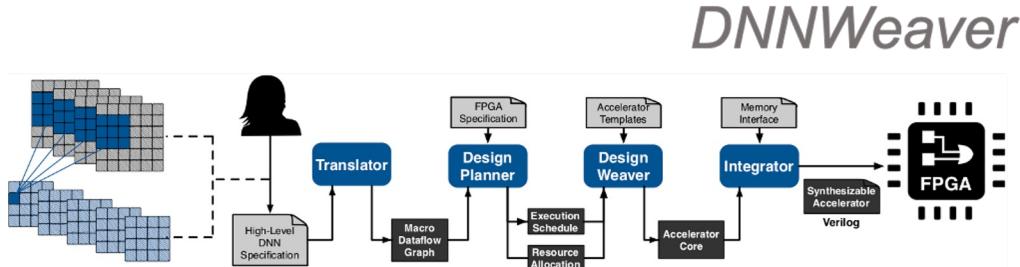
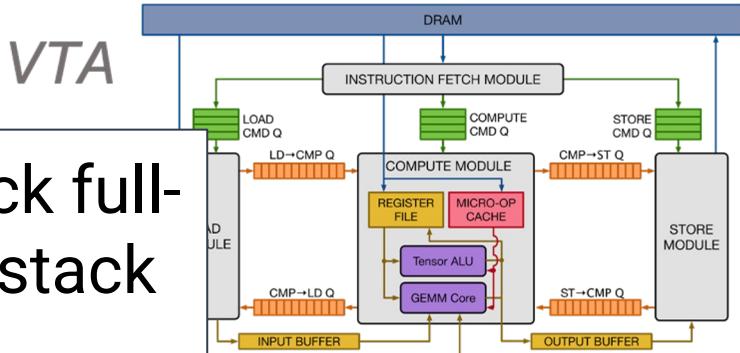


MAGNet

# Which means DNN accelerator generators are exploding in popularity...



However, they lack full-system and full-stack visibility

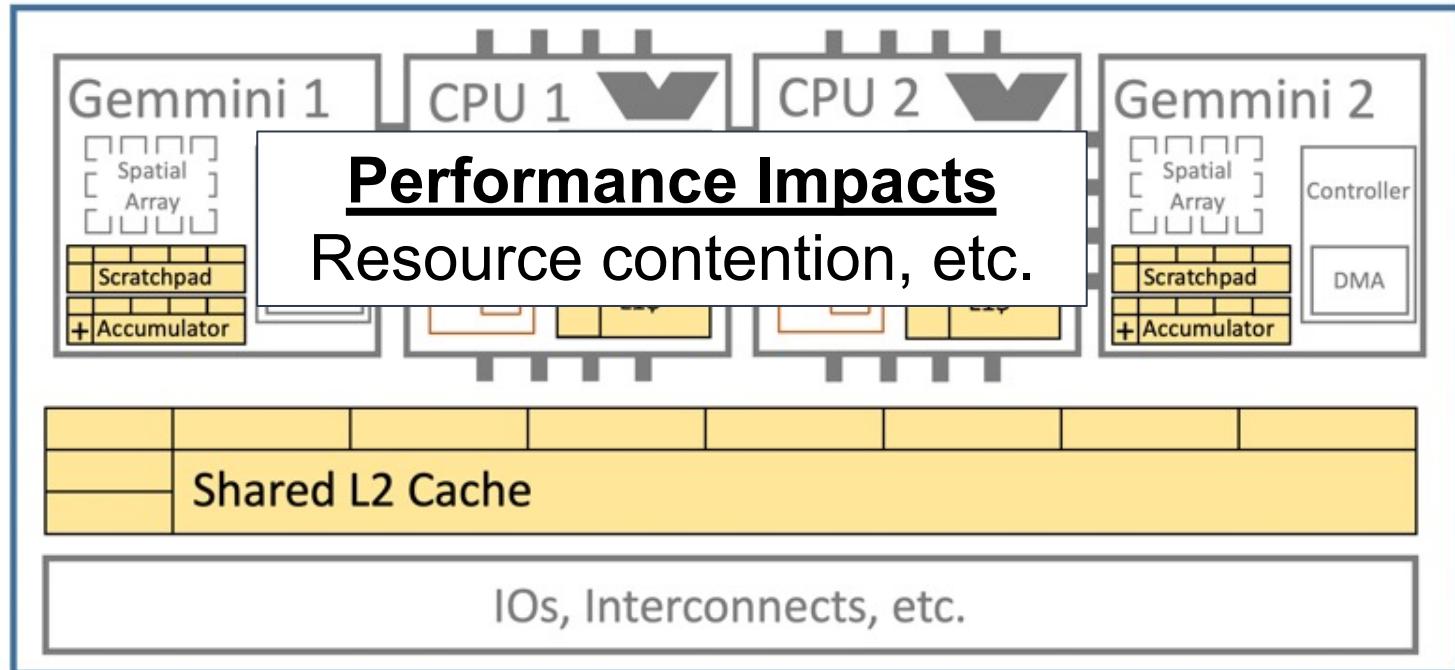


MAGNet

# Full-System Visibility

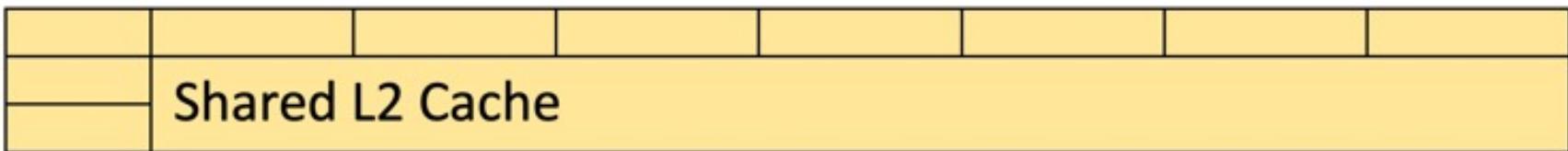
# Full-System Visibility: SoC

SoC



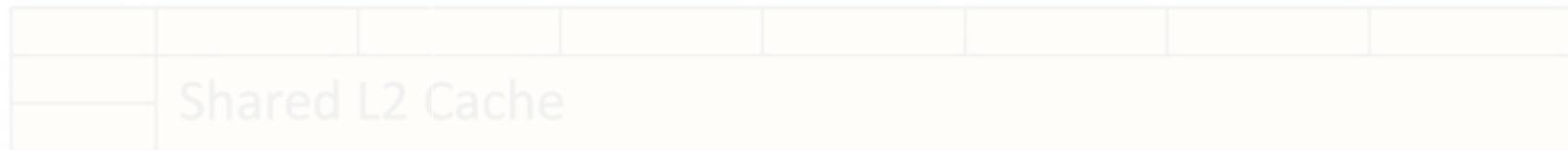
# Full-System Visibility: Memory Hierarchy

**Performance Impacts**  
Cache coherence, miss  
rates/latencies, etc.

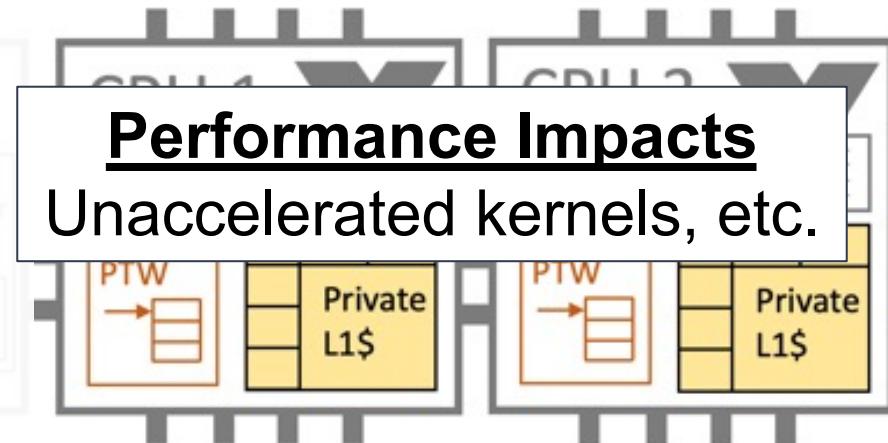


# Full-System Visibility: Virtual Addresses

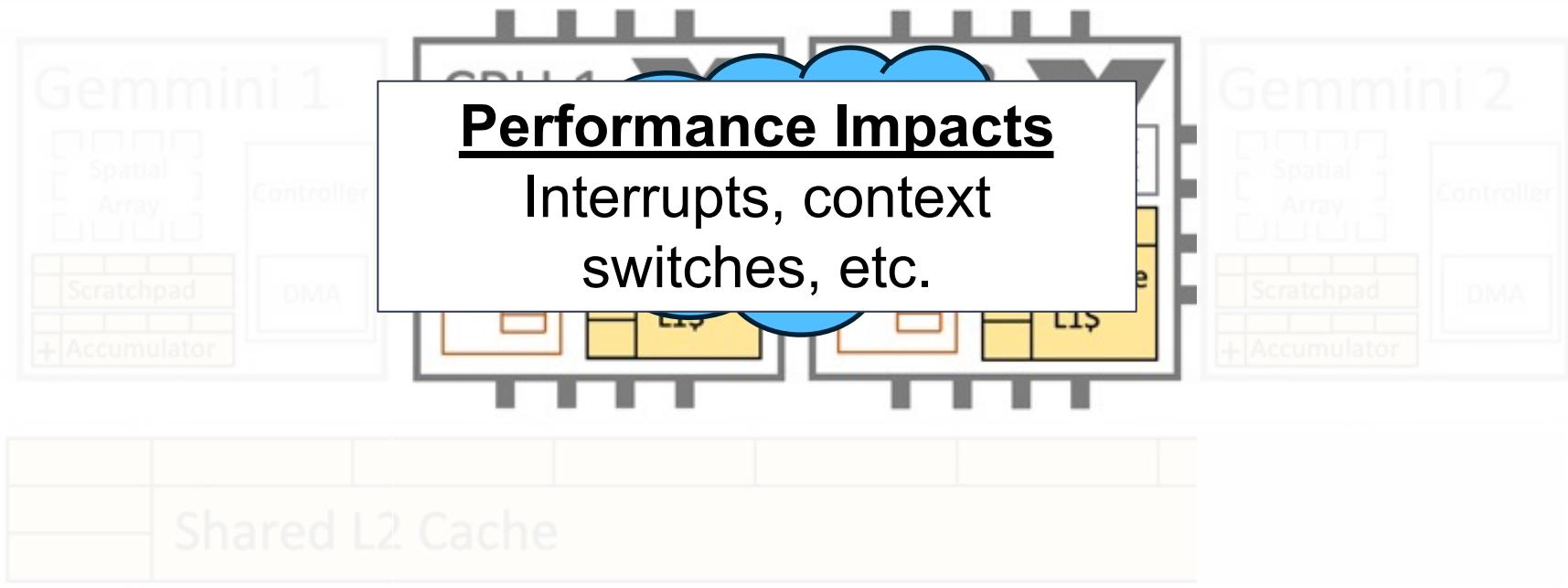
**Performance Impacts**  
Page faults, TLB hits, etc.



# Full-System Visibility: Host CPUs



# Full-System Visibility: Operating System



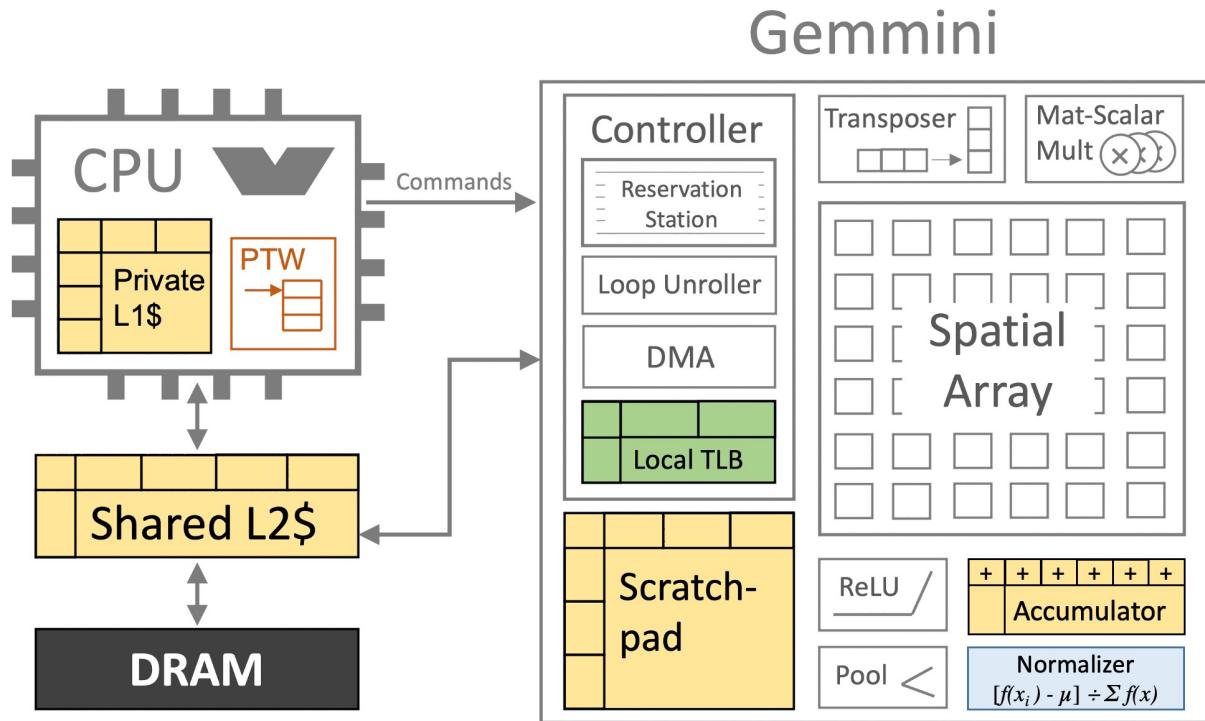
# Full-Stack Visibility



# Gemmini Architectural Template

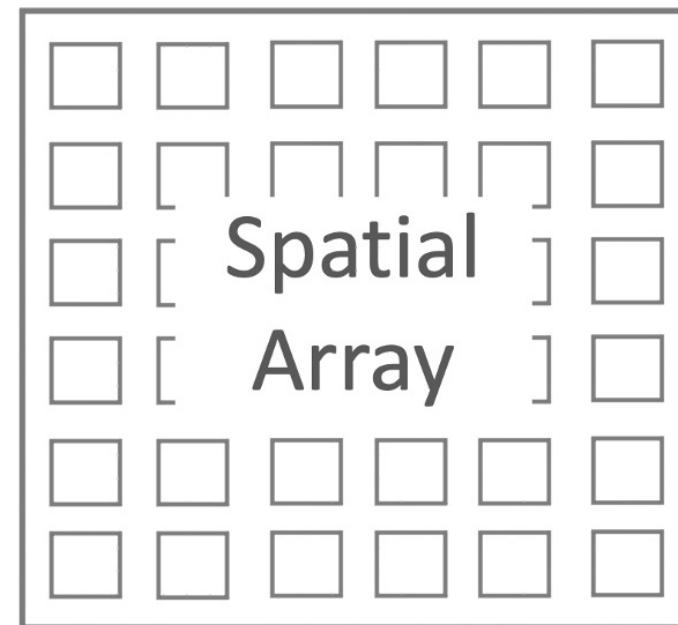
# Gemmini

- DNN accelerator generator
- Flexible hardware template
- Full-stack
- Full-system



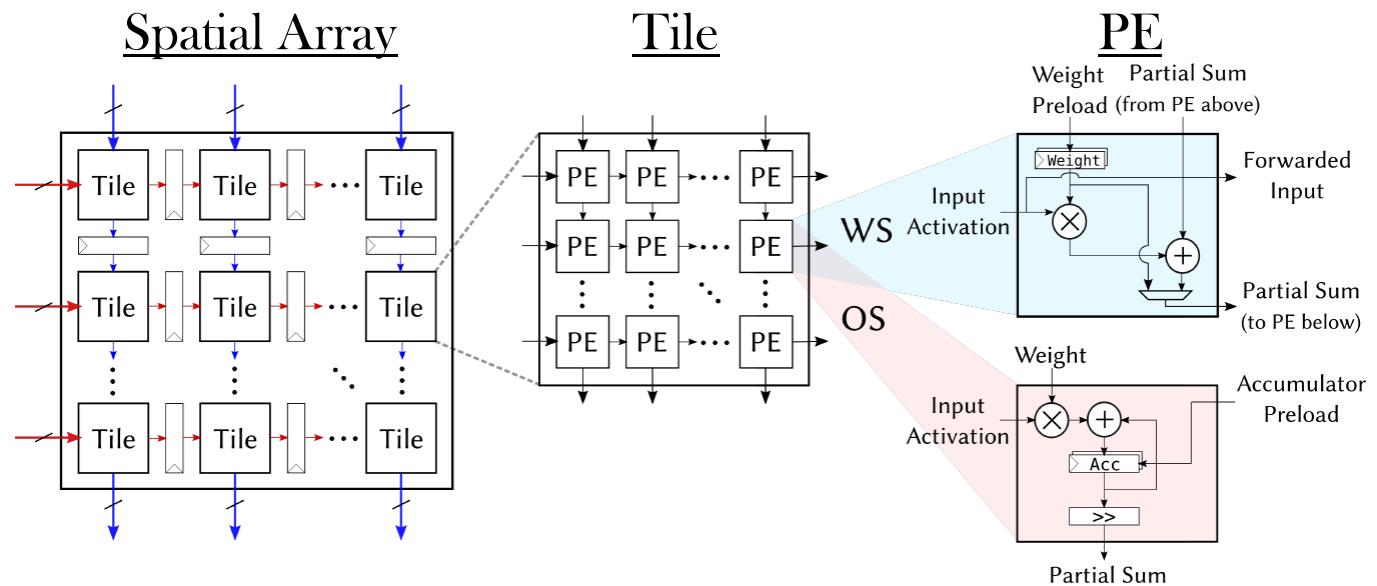
# Gemmini: Spatial Array

- Parameters:
  - Dataflow
  - Dimensions
  - Pipelining



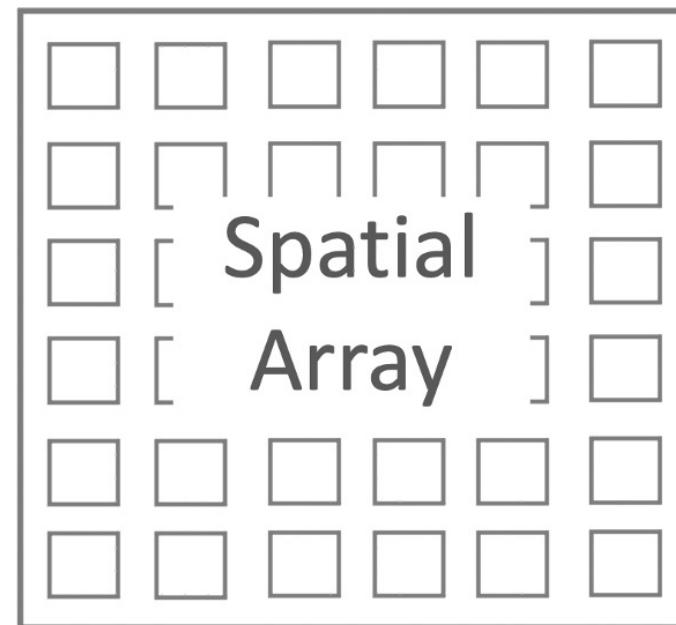
# Gemmini: Spatial Array

- Parameters:
  - Dataflow
  - Dimensions
  - Pipelining



# Gemmini: Spatial Array

- Parameters:
  - Dataflow
  - Dimensions
  - Pipelining



# Gemmini: Non-GEMM Functionality

- Can be optimized out at elaboration-time



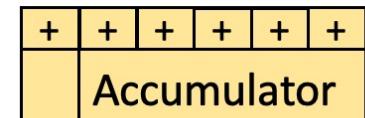
# Gemmini: Normalization Unit

- **New feature**
- Added to support I-BERT
  - But customizable to other transformer models as well
- Performs normalization operations, e.g.:
  - I-BERT's softmax
  - Layernorm
- **Caveats:** Only performs normalizations on **innermost dimension**
  - So BatchNorm is not yet supported



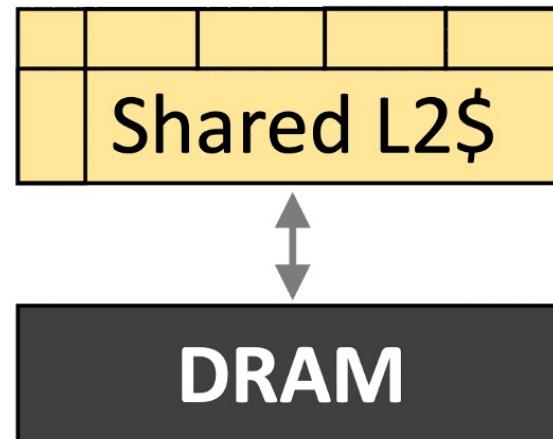
# Gemmini: Local Scratchpad and Accumulator

- Parameters:
  - Capacity
  - Banks
  - Single- or dual-port



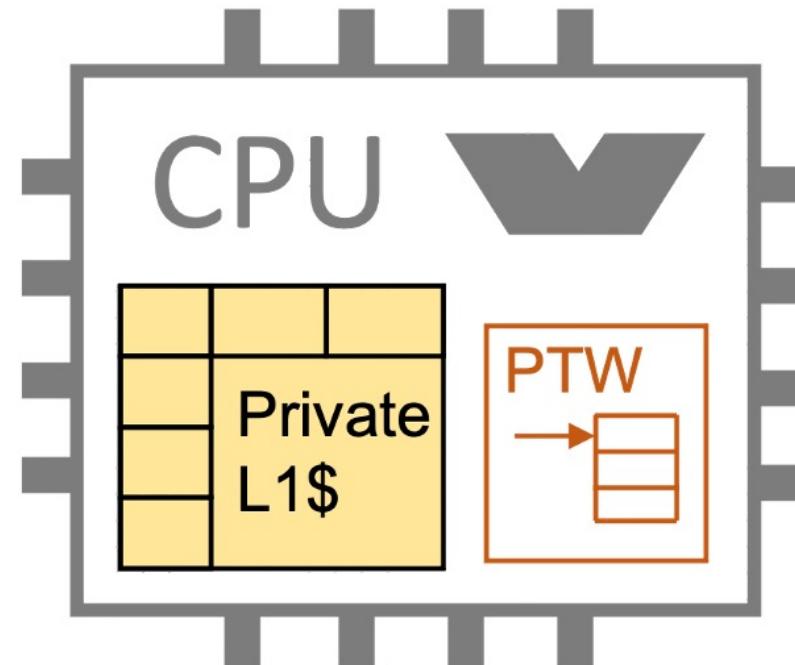
# Gemmini: Global Memory

- Parameters:
  - Capacity
  - Banks
  - Optional L3
  - DRAM controller



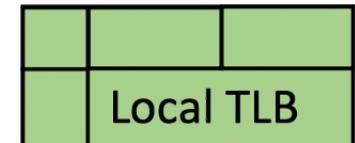
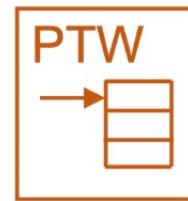
# Gemmini: Host CPU

- Parameters:
  - In-order/out-of-order
  - ROB capacity
  - L1 capacity
  - Branch predictor



# Gemmini: Virtual Address Translation

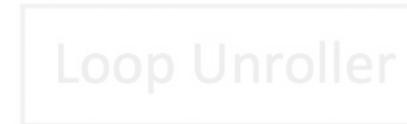
- Parameters:
  - TLB capacity
  - TLB hierarchy
    - e.g. L2 TLB



# Gemmini: Reservation Station

- Parameters:
  - Size
    - Separate entries for Id-, st-, and execute-type instructions

Controller



# Gemmini: Loop Unroller

- Dynamically schedules operations
- Parameters:
  - Types of loops to unroll in hardware
    - Only inference kernels?
    - Training *and* inference kernels?

Controller

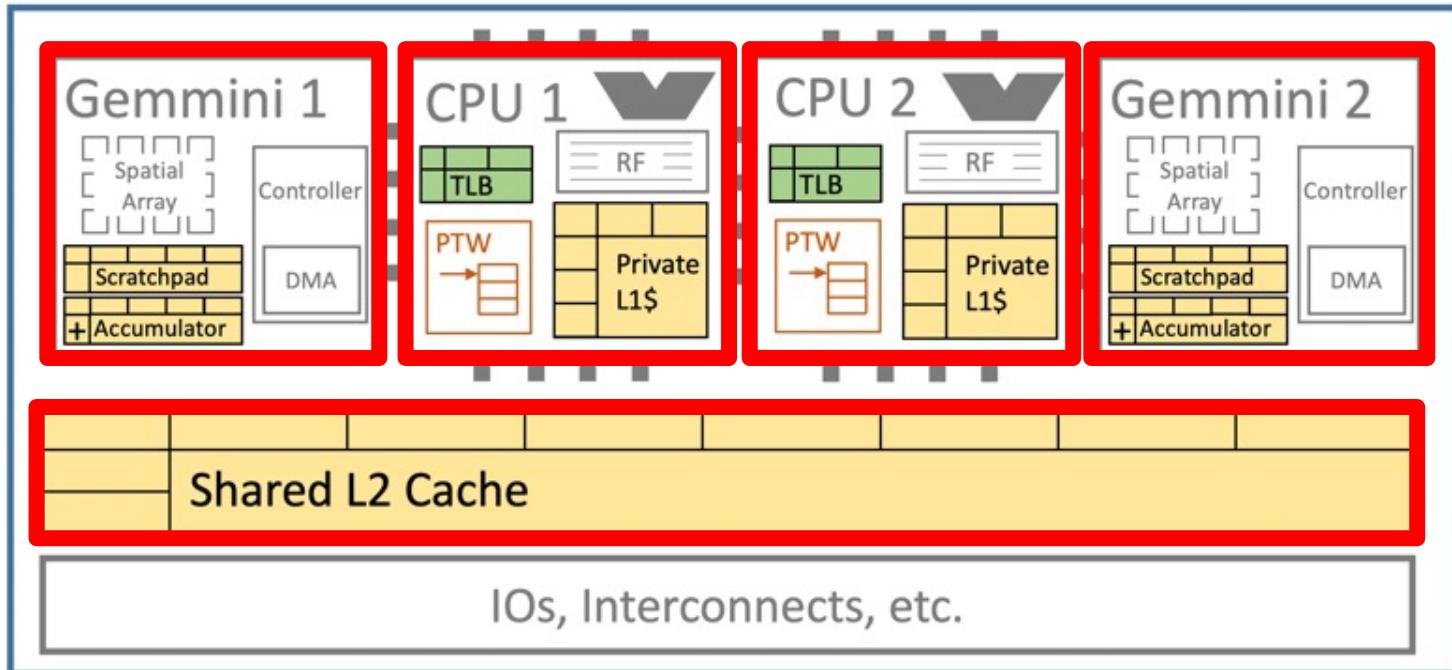
Reservation  
Station

Loop Unroller

DMA

# Gemmini: Full SoC

SoC



# *Interactive Activity: Build Your Own Gemmini*

# Setup Gemmini Config Files

```
cd /root/chipyard/generators/gemmini  
./scripts/setup-paths.sh  
ls configs/  
  
# Expected output:  
#   CPUConfigs.scala  GemminiCustomConfigs.scala  
#   GemminiDefaultConfigs.scala  SoCConfigs.scala
```

# View Gemmini Configs

```
cd /root/chipyard/generators/gemmini  
vim configs/GemminiDefaultConfigs.scala  
vim configs/GemminiCustomConfigs.scala  
# Update "val customConfig" to decide which Gemmini config you want  
# to build
```

# View CPU Configs

```
cd /root/chipyard/generators/gemmini
```

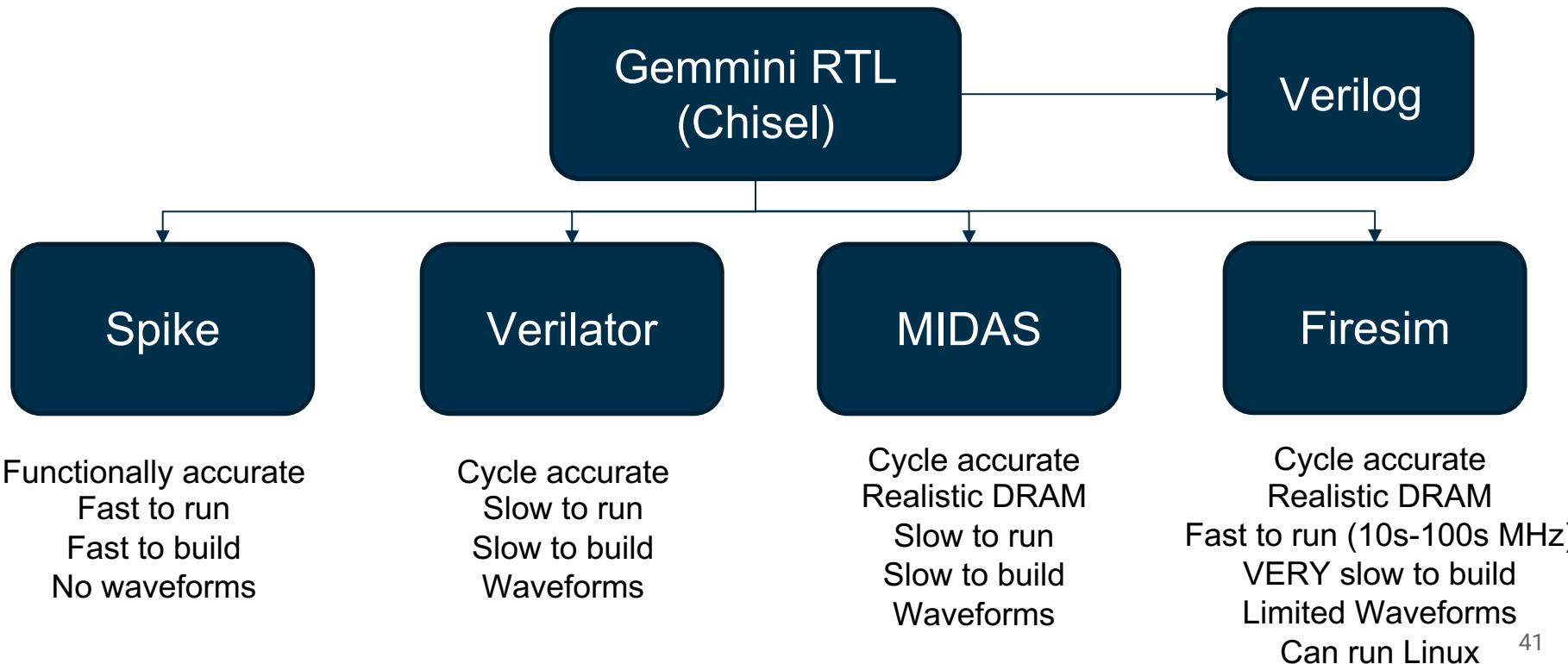
```
vim configs/CPUConfigs.scala
```

```
# Update "type CustomCPU" to decide which CPU config you want to build
```

# View SoC Configs

```
cd /root/chipyard/generators/gemmini  
vim configs/SoCConfigs.scala  
# Update "class CustomGemminiSoCConfig" to configure your SoC
```

# Gemmini Build Outputs



# Build Functional Simulator (Spike)

```
cd /root/chipyard/generators/gemmini  
./scripts/build-spike.sh
```

# Build Cycle-Accurate Simulator (Verilator)

```
cd /root/chipyard/generators/gemmini  
./scripts/build-verilator.sh  
  
# Or, if you have VCS:  
# ./scripts/build-vcs.sh  
  
# To build a simulator which can generate waveforms:  
# ./scripts/build-verilator.sh --debug  
  
# To speed up compilation, you can try:  
# ./scripts/build-verilator.sh -j 2  
  
# Ctrl-C because this will take too long, and run the script below  
# to load a prebuilt Verilator simulator  
  
./tutorial/checkpoint.sh build-verilator
```

# Build Cycle-Accurate Simulator With Realistic DRAM and LLC (MIDAS)

```
cd /root/chipyard/generators/gemmini  
  
./scripts/build-midas.sh --help  
  
./scripts/build-midas.sh DDR3FCFS  
  
# Ctrl-C because this will take too long, and run the script below  
# to load a prebuilt MIDAS simulator  
  
.tutorial/checkpoint.sh build-midas
```

# Build Tests

```
cd /root/chipyard/generators/gemmini      # Your binaries will be in build/  
  
cd software/gemmini-rocc-tests  
  
vim bareMetalC/template.c  
  
.build.sh  
  
# You can pass in Make arguments:  
# ./build.sh -j1  
  
# Baremetal binaries  
ls build/*/*-baremetal  
  
# Linux binaries  
ls build/*/*-linux
```

# Run Tests on Functional Simulator

```
cd /root/chipyard/generators/gemmini  
  
./scripts/run-spike.sh template  
  
./scripts/run-spike.sh tiled_matmul_ws  
  
./scripts/run-spike.sh mobilenet  
  
# “run-spike.sh” is a simple convenience wrapper around:  
#   “spike --extension=gemmini BINARY”
```

# Run Tests on Cycle-Accurate Simulators

```
cd /root/chipyard/generators/gemmini  
  
./scripts/run-verilator.sh template  
  
./scripts/run-midas.sh DDR3FCFS template  
  
# You may need to run “reset” after “run-midas.sh” to see what  
# you’re typing  
  
# If “run-midas.sh” doesn’t work for you, then run this instead:  
# vim tutorial/midas.log
```

# View Generated Verilog

```
cd /root/chipyard/generators/gemmini
```

```
# The path below may look very long, but tab-complete will get you  
# almost all the way there. Alternatively, copy the path from the  
# Google Doc.
```

```
vim generated-  
src/verilator/chipyard.TestHarness.CustomGemminiSoCConfig/chipyard.T  
estHarness.CustomGemminiSoCConfig.top.v
```

# *Interactive Activity: Add a New Datatype to Gemmini*

# Create a New Datatype

```
cd /root/chipyard/generators/gemmini
```

```
vim src/main/scala/gemmini/Arithmetic.scala
```

```
# Fill in '+' and 'mac' operators for Complex
```

```
# Hint: You can define 'mac' in terms of the existing '+' and '*'  
# operators
```

```
# Note: By default, Gemmini supports power-of-2 length datatypes, up  
# to 32-bits wide.
```

```
# Note: Signed integers, unsigned integers, and floats with  
# arbitrary exp and sig widths (e.g. float32, float16,  
# bfloat16, float8, etc.) are all supported.
```

# Update Custom Config

```
cd /root/chipyard/generators/gemmini  
vim configs/GemminiCustomConfigs.scala  
# Update "val customConfig" to "complexConfig"
```

# Build Simulator

```
cd /chipyard/generators/gemmini  
  
./scripts/build-verilator.sh  
  
# Ctrl-C because this will take too long, and run the script below  
# to load a prebuilt ComplexNumbers Verilator simulator  
  
.tutorial/checkpoint.sh build-complex
```

# Build Tests

```
cd /chipyard/generators/gemmini
```

```
cd software/gemmini-rocc-tests
```

```
./build.sh
```

```
# You can pass in Make arguments:
```

```
# ./build.sh -j1
```

# Run Cycle-Accurate Simulation

```
cd /chipyard/generators/gemmini  
  
./scripts/run-verilator.sh complex_numbers  
  
# Test squares a diagonal matrix of (-i), producing a negative  
# identity matrix
```

# Session 2: Gemmini's Programming Stack

# Outline

- Programming Stack
  - High-level: Compiling ONNX models
  - Mid-level: Hand-tuned Kernel Library
  - Low-level: Direct Machine Configuration and Assembly
- *Interactive Activities:*
  - Visualize and Run ONNX Models
  - ResNet50 First-Layer Optimizations With Low-Level ISA

# Programming Model



ONNX

---

matmul(...); conv(...); residual\_add(...);  
max\_pool(...); global\_averaging(...)

---

Direct hardware configuration, low-level  
preload\_spatial\_array(...); feed\_spatial\_array(...)



High

Medium

Low

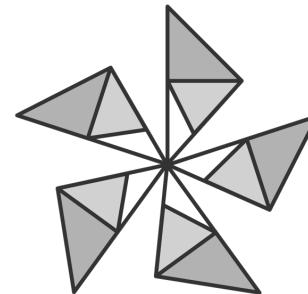
# High-Level: ONNX Models

- Run ONNX models on Gemmini!
- ONNX files can be supported by major frameworks
  - PyTorch
  - TensorFlow
  - MXNet
- Tons of models available on ONNX Model Zoo
  - Image classification
  - Object detection
  - NLP

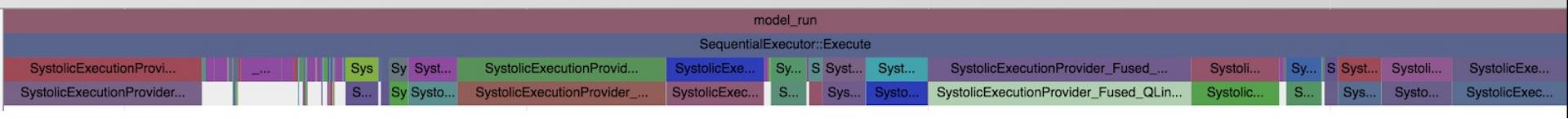


# High-Level: ONNX Runtime

- Maps ONNX kernels to accelerators
  - Dynamic dispatch
  - Falls back to CPU if no accelerators available
- Supports inference and training
- We **forked** ONNX Runtime
- Our ONNX Runtime fork:
  - Quantizes models to int8
  - Fuses ops
    - Matmul + ReLU, matmul + pooling, etc.
  - NCHW -> NHWC
  - **New:** Model-parallelism on some operations
    - “Tensor-parallelism” in particular
- Provides timing profile



ONNX  
RUNTIME



# Mid-Level: Hand-Tuned Library

- C library of kernels

- `#include "gemmini.h"`

```
void tiled_matmul(...)
```

```
void tiled_resadd(...)
```

```
// Convs and max-pools are fused  
// together  
void tiled_conv(...)
```

# Mid-Level: Hand-Tuned Library

- C library of kernels
  - `#include "gemmini.h"`
- Requires a C description of the accelerator configuration
  - `#define DIM ...`
  - `typedef elem_t ...`

# Mid-Level: Hand-Tuned Library

- C library of kernels
  - `#include "gemmini.h"`
- Requires a C description of the accelerator configuration
  - `#include "gemmini_params.h"`
- Write your own Gemmini applications

```
// My CV application  
  
#include "gemmini.h"  
  
int main() {  
    elem_t image[...][...][...];  
  
    tiled_conv(image, ...);  
  
    // ...  
}
```

# Mid-Level: Hand-Tuned Library

- C library of kernels
  - `#include "gemmini.h"`
- Requires a C description of the accelerator configuration
  - `#include "gemmini_params.h"`
- Write your own Gemmini applications
- Performs loop tiling
  - Uses heuristics to maximize buffer usage

```
for (int i = 0; i < N; i++) {  
    // ...  
}
```



```
// Global memory level  
for (int i0 = 0; i0 < N; i0 += tile_I)  
{  
    // Scratchpad level  
    for (int i = 0; i < tile_I; i++) {  
        // ...  
    }  
}
```

# Mid-Level: Hand-Tuned Library

- C library of kernels
  - `#include "gemmini.h"`
- Requires a C description of the accelerator configuration
  - `#include "gemmini_params.h"`
- Write your own Gemmini applications
- Performs loop tiling
  - Uses heuristics to maximize buffer usage
- Assumes NHWC format

# Mid-Level: Hand-Tuned Library

- C library of kernels
    - `#include "gemmini.h"`
  - Requires a C description of the accelerator configuration
    - `#include "gemmini_params.h"`
  - Write your own Gemmini applications
  - Performs loop tiling
    - Uses heuristics to maximize buffer usage
  - Assumes NHWC format
- 
- Matmuls
    - Transposed
    - Strided
  - Convolutions
    - Forward and backward passes
    - Kernel or input dilated
  - Matrix additions
    - Scaled
  - Max-pooling
  - Global-averaging

# Low-Level: Gemmini ISA

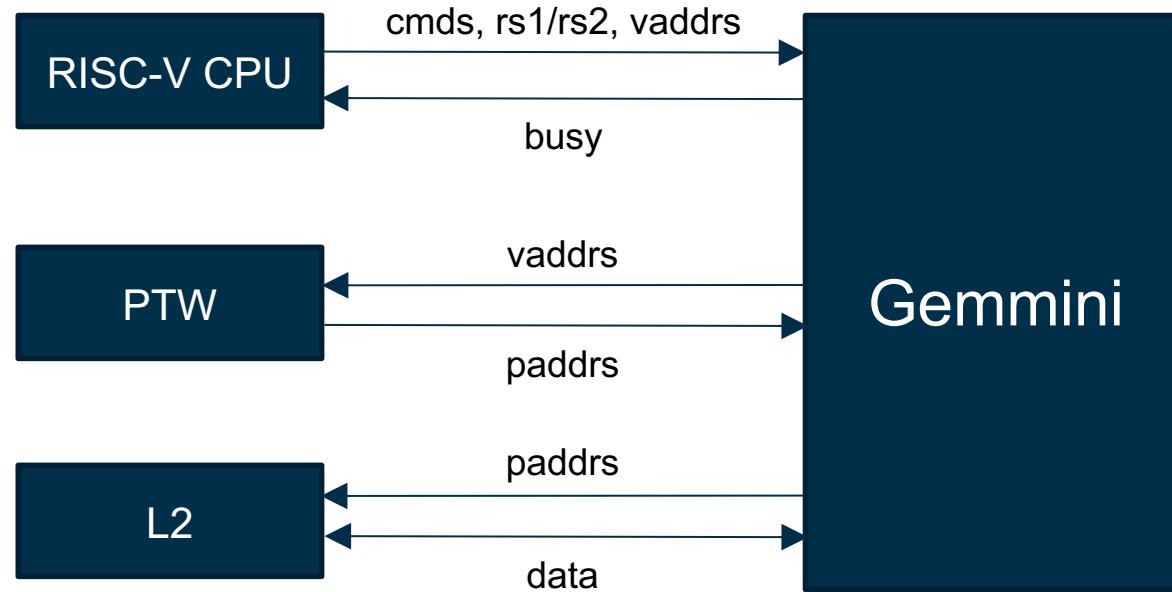
- Only necessary if you:
  - Want to write your own mid-level kernels
  - Performance tuning
- Low-level assembly instructions
  - Wrapped in C "#define" macros
  - `#include "gemmini.h"`

# Low-Level: Programming Model

- Gemmini is programmed using custom RISC-V instructions
  - "RoCC" interface
    - "Rocket-Chip Coprocessor"
- Gemmini instructions are interspersed between ordinary CPU instructions
- Gemmini does *not* fetch its own instructions
- Whenever the CPU encounters a RoCC instruction, it:
  - Checks the opcode to find target
  - Dispatches to target co-processor
  - *Immediately* continues CPU execution
    - Synchronize with fences

31	25	24	20	19	15	14	13	12	11	7	6	0
funct	rs2	rs1	xd	xs1	xs2	rd						opcode
7	5	5	1	1	1	1	1	1	1	5	7	0

# Low-Level: Gemmini Interface With CPU

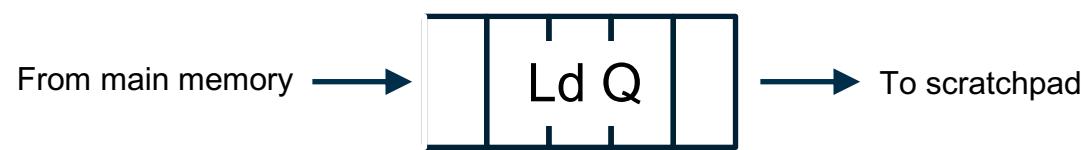


# Low-Level: Decoupled Access-Execute Pipelines

- Three main “types” of instructions:

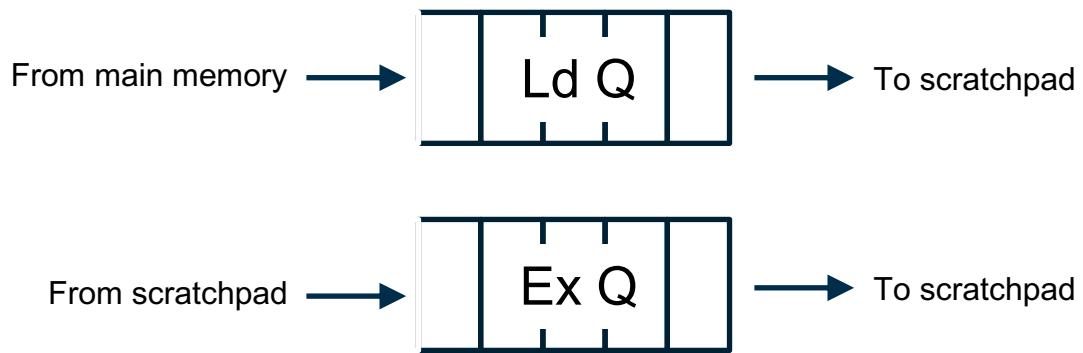
# Low-Level: Decoupled Access-Execute Pipelines

- Three main “types” of instructions:
  - Load instructions
    - Main memory -> Scratchpad



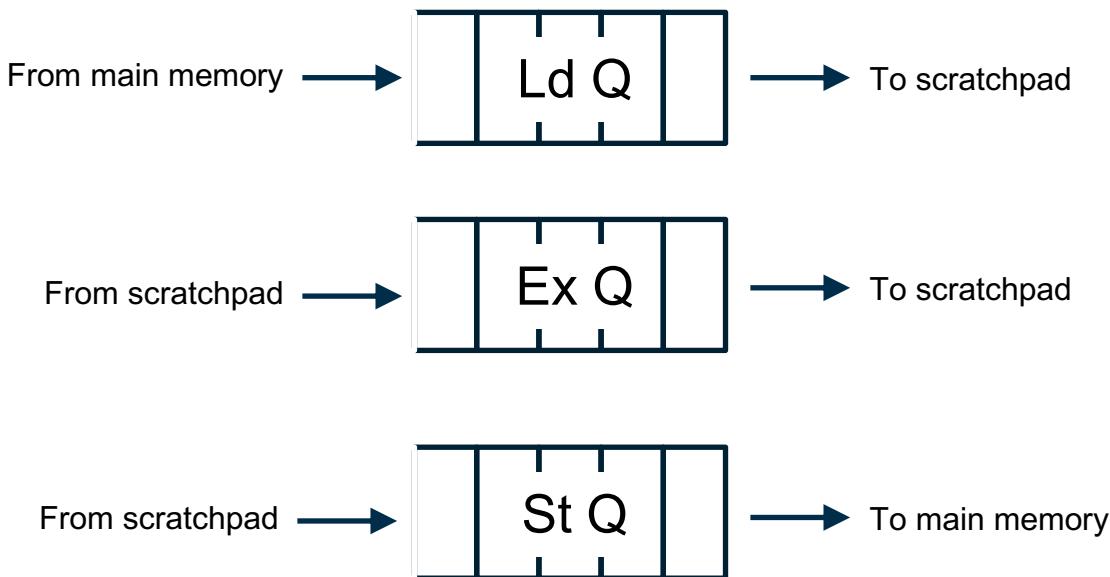
# Low-Level: Decoupled Access-Execute Pipelines

- Three main “types” of instructions:
  - Load instructions
    - Main memory -> Scratchpad
  - Execute instructions
    - Scratchpad -> Scratchpad



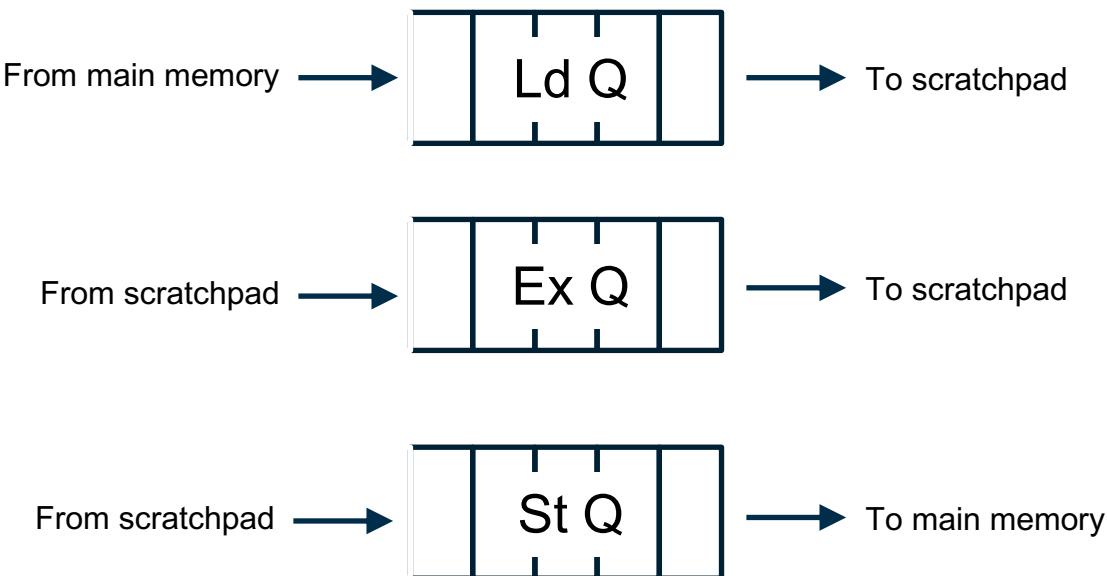
# Low-Level: Decoupled Access-Execute Pipelines

- Three main “types” of instructions:
  - Load instructions
    - Main memory -> Scratchpad
  - Execute instructions
    - Scratchpad -> Scratchpad
  - Store instructions
    - Scratchpad -> Main memory



# Low-Level: Decoupled Access-Execute Pipelines

- Three main “types” of instructions:
  - Load instructions
    - Main memory -> Scratchpad
  - Execute instructions
    - Scratchpad -> Scratchpad
  - Store instructions
    - Scratchpad -> Main memory
- Matches Gemmini’s decoupled-access execute pipelines



# Low-Level: Gemmini ISA Example Program

```
cd /root/chipyard/generators/gemmini
```

```
vim software/gemmini-rocc-tests/bareMetalC/template.c
```

```
# A basic program which loads a matrix into the scratchpad, performs  
# a matmul with it, and stores the result back into main memory  
# where the CPU can read it.
```

# Low-Level: Local Memory

Config  
DIM: 2  
inputType: int8  
accType: int32

## Scratchpad

2 int8 values

	Addr 0x0
	Addr 0x1
	Addr 0x2
	Addr 0x3
	Addr 0x4
	Addr 0x5
...	

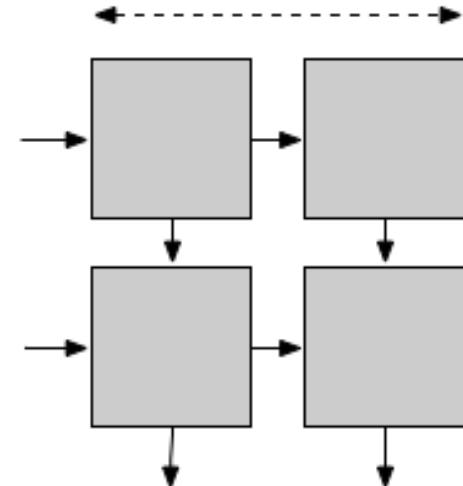
## Accumulator

2 int32 values

	Addr 0x80000000
	Addr 0x80000001
	Addr 0x80000002
	Addr 0x80000003
	Addr 0x80000004
	Addr 0x80000005
...	

## Systolic Array

2 PEs



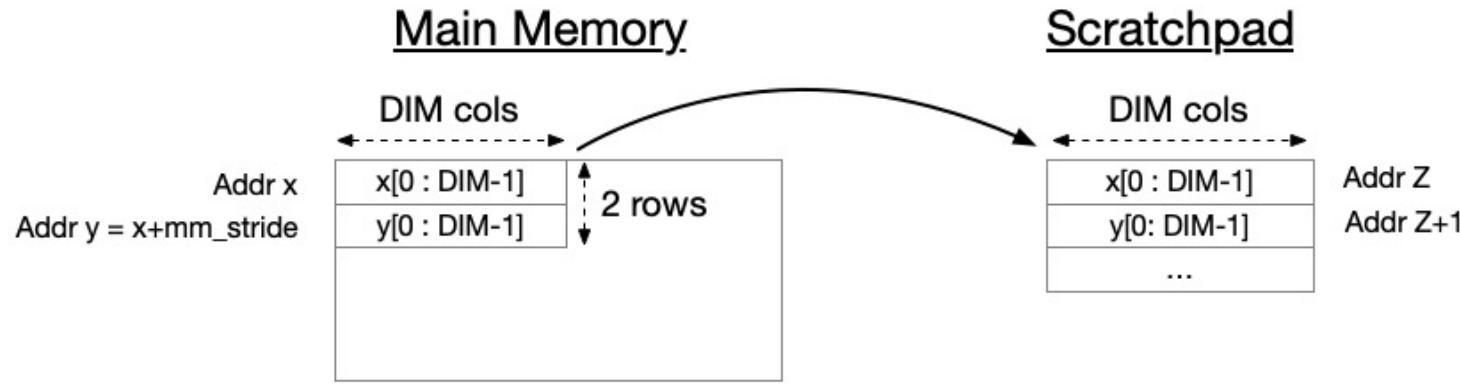
# Low-Level: Load Pipeline

- Three “identical” load instructions:
  - gemmini\_mvin1
  - gemmini\_mvin2
  - gemmini\_mvin3
- Each mvin has it’s own config registers
- Intended use:
  - Different mvins for Inputs, Weights, and Bias
- Mvins are meant to move DIMxDIM **submatrices** from main memory into the scratchpad
  - DIM is the dimension of the matmul unit

# Low-Level: Load Pipeline

- Three “identical” load instructions:
  - gemmini\_mvin1
  - gemmini\_mvin2
  - gemmini\_mvin3
- Each mvin has its own config registers
- Intended use:
  - Different mvins for Inputs, Weights, and Bias
- Mvins are meant to move DIMxDIM **submatrices** from main memory into the scratchpad
  - DIM is the dimension of the matmul unit
- Config instruction
  - config\_mvin
- Config options
  - mm\_stride
    - Stride in bytes between rows of submatrix
  - scale
    - Enables scalar-matrix multiplications during mvins
  - id
    - Which mvin instruction is being configured?

# Low-Level: Mvin



## Mvin Instruction Parameters

*From:* x (main memory address)

*To:* Z (private memory address)

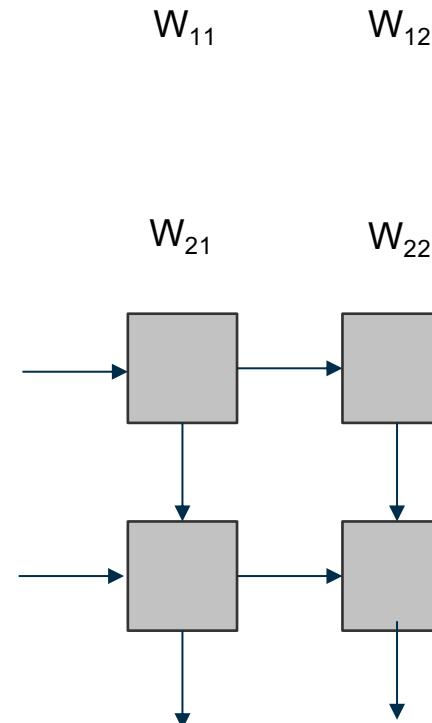
*Rows:* 2   *Columns:* DIM

*Main memory stride:* mm\_stride

# Low-Level: Execute Pipeline

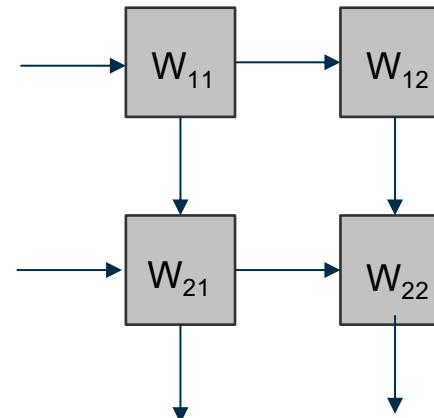
# Low-Level: Execute Pipeline

- “Preload” instruction loads stationary data into the matmul unit
  - preload
  - Weights for weight-stationary, bias/output for output-stationary



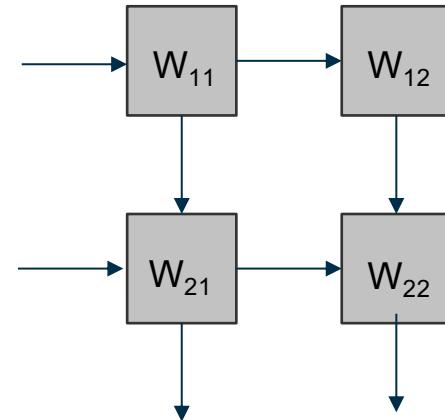
# Low-Level: Execute Pipeline

- “Preload” instruction loads stationary data into the matmul unit
  - preload
  - Weights for weight-stationary, bias/output for output-stationary



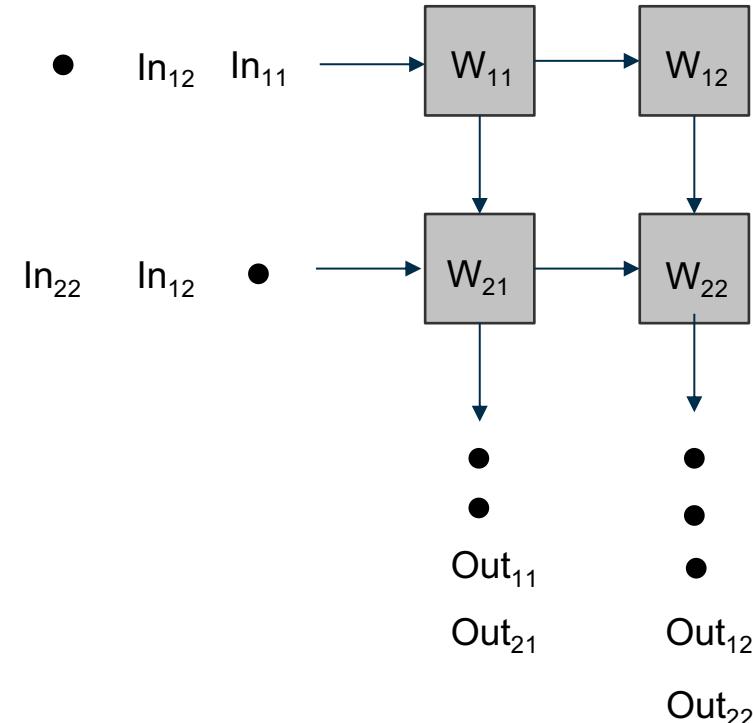
# Low-Level: Execute Pipeline

- “Preload” instruction loads stationary data into the matmul unit
  - preload
  - Weights for weight-stationary, bias/output for output-stationary



# Low-Level: Execute Pipeline

- “Preload” instruction loads stationary data into the matmul unit
  - preload
  - Weights for weight-stationary, bias/output for output-stationary
- “Compute” instruction performs matmul after stationary data has been loaded
  - compute\_preloaded
  - compute\_accumulated



# Low-Level: Execute Pipeline

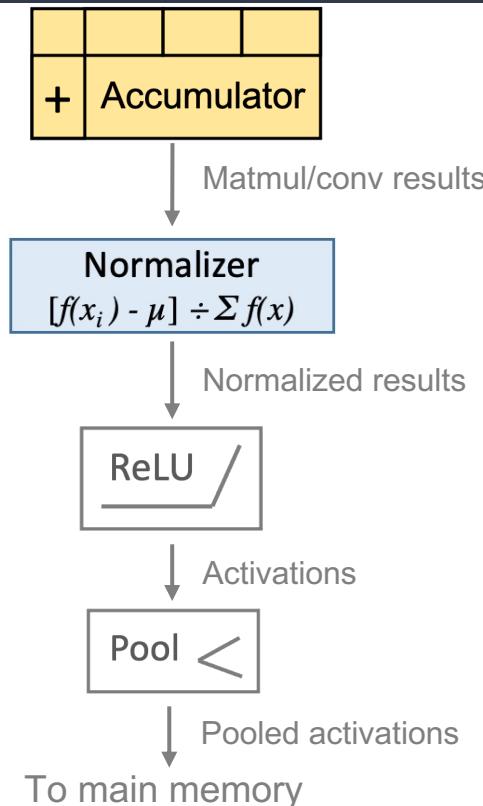
- “Preload” instruction loads stationary data into the matmul unit
  - `preload`
  - Weights for weight-stationary, bias/output for output-stationary
- “Compute” instruction performs matmul after stationary data has been loaded
  - `compute_preloaded`
  - `compute_accumulated`
- Config instruction
  - `config_ex`
- Config options:
  - `dataflow`
    - OS/WS
    - Can also be hardened at compile-time
  - `activation`
    - Only used for OS dataflow, as results are read out of the matmul unit
  - `transpose_a`
  - `transpose_b`
  - `a_stride`
    - Useful for convolutions with stride > 1
  - `c_stride`
    - Useful for backward-pass convs
    - Enables non-contiguous outputs

# Low-Level: Store Pipeline

- Very similar to load pipeline
  - gemmini\_mvout
- Stores **DIMxDIM submatrices** into main memory

# Low-Level: Store Pipeline

- Very similar to load pipeline
  - gemmini\_mvout
- Stores **DIMxDIM submatrices** into main memory
- For the WS dataflow, **normalization** and **activation functions** are applied as results are read out of the accumulator, during mvouts
- **Max-pooling** happens during mvouts



# Low-Level: Store Pipeline

- Very similar to load pipeline
  - gemmini\_mvout
- Stores DIMxDIM **submatrices** into main memory
- For the WS dataflow, **normalization** and **activation** functions are applied as results are read out of the accumulator, during mvouts
- **Max-pooling** happens during mvouts
- **Normalization** happens during mvouts
- Config instructions
  - config\_st
  - config\_norm
- Config options
  - stride
  - activation
    - Only used for WS dataflow
  - scale
    - Scalar–matrix multiplication during mvouts from accumulator
  - Pooling options
    - pool\_size
    - pool\_stride
  - ...

# Low-Level: “Complex” vs “Primitive” Instructions

- “Primitive” instructions
  - Mvin
  - Preload
  - Compute
  - Mvout
- Usually operate on DIMxDIM submatrices
- Explicitly manage scratchpad
- “Mid-level” programming interface composes these functions into loops
  - Convolutions
    - With fused pooling
  - Matmuls
  - Mat-adds
- “Complex” instructions hardcode these loops
  - loop\_matmul
  - loop\_conv
- Unrolled by hardware FSMs
- Main memory to main memory
  - Scratchpad is not explicitly managed by programmer
- **Optional** feature
- Enables performance optimizations
  - Dynamic scheduling
    - Good overlap between ld/ex/st instructions
  - Double-buffering
  - Lower instruction bandwidth requirements

# Low-Level: “Primitive”

```
// Main memory tiles
for (int i0 = 0; i0 < N; i0 += TILE_I) {
    for (int j0 = 0; j0 < N; j0 += TILE_J) {
        for (int k0 = 0; k0 < N; k0 += TILE_K) {

            // Scratchpad tiles
            for (int i1 = 0; i1 < N; i1 += DIM) {
                for (int j1 = 0; j1 < N; j1 += DIM) {
                    for (int k1 = 0; k1 < N; k1 += DIM) {
                        int i = i0 * tile_I + i1;
                        int j = j0 * tile_J + j1;
                        int k = k0 * tile_K + k1;

                        A_addr = ...; B_addr = ...; C_addr = ...;

                        if (j1 == 0) gemmini_mvin(&A[i0*tile_I + i1][k*TILE_K + k1], A_addr);
                        if (i1 == 0) gemmini_mvin(&B[k0*tile_K + k1][j*TILE_J + j1], B_addr);

                        gemmini_preload(B_addr, C_addr);
                        gemmini_compute(A_addr);
                    }
                }
            }
        }
    }
}
```

# Low-Level: “Primitive”

```
// Main memory tiles
for (int i0 = 0; i0 < N; i0 += TILE_I) {
    for (int j0 = 0; j0 < N; j0 += TILE_J) {
        for (int k0 = 0; k0 < N; k0 += TILE_K) {

            // Scratchpad tiles
            for (int i1 = 0; i1 < N; i1 += DIM) {
                for (int j1 = 0; j1 < N; j1 += DIM) {
                    for (int k1 = 0; k1 < N; k1 += DIM) {
                        int i = i0 * tile_I + i1;
                        int j = j0 * tile_J + j1;
                        int k = k0 * tile_K + k1;

                        A_addr = ...; B_addr = ...; C_addr = ...;

                        if (j1 == 0) gemmini_mvin(&A[i0*tile_I + i1][k*TILE_K + k1], A_addr);
                        if (i1 == 0) gemmini_mvin(&B[k0*tile_K + k1][j*TILE_J + j1], B_addr);

                        gemmini_preload(B_addr, C_addr);
                        gemmini_compute(A_addr);

                    }
                }
            }
        }
    }
}
```

# Low-Level: “Complex”

```
// Main memory tiles
for (int i0 = 0; i0 < N; i0 += TILE_I) {
    for (int j0 = 0; j0 < N; j0 += TILE_J) {
        for (int k0 = 0; k0 < N; k0 += TILE_K) {
            // Scratchpad tiles
            gemmini_loop_matmul(...);
        }
    }
}
```

# *Interactive Activity: Visualize and Run ONNX Models*

# Get ONNX Model

# On your personal laptop, download this file:

<https://tinyurl.com/gemm-mlsys>

# Visualize ONNX Model

- Go to [netron.app](https://netron.app)
- Load model into website



# Run ONNX Model On Spike: Inference

```
cd /root/chipyard/generators/gemmini

# Load pre-compiled Spike and Onnx-Runtime builds
./tutorial/checkpoint.sh build-onnx-inference

# Run inference
cd /root/chipyard/generators/gemmini/software/onnxruntime-riscv/
cd systolic_runner/imagenet_runner/

spike --extension=gemmini pk ort_test -h

spike --extension=gemmini pk ort_test -m resnet50_opt_quant.onnx -i
images/dog.jpg -p caffe2 -x 2 -0 99

# If you don't want to type all that, try:
# cd /root/chipyard/generators/gemmini
# ./tutorial/run-onnx-inference.sh
```

# Run ONNX Model On Spike: ResNet50 Training

```
cd /root/chipyard/generators/gemmini

# Load pre-compiled Spike and Onnx-Runtime builds
./tutorial/checkpoint.sh build-onnx-training

# Run ResNet50 training
cd /root/chipyard/generators/gemmini/software/onnxruntime-riscv/
cd systolic_runner/imagenet_trainer/

spike --extension=gemmini pk resnet_train --model_name resnet50.onnx --
train_data_txt batch_out.txt --num_train_steps 1 --train_batch_size 10 -x 2 -d 0 -
0 99

# If you don't want to type all that, try:
# cd /root/chipyard/generators/gemmini
# ./tutorial/run-onnx-training.sh
```

# Run ONNX Model On Spike: ResNet50 Training

```
# Simulating training is very slow. Look in this file instead to see  
# the output:  
  
vim tutorial/onnx-train.log
```

# Visualize ONNX-Runtime Trace

# On your personal laptop, download this file:

<https://tinyurl.com/gemmini-resnet50-trace>

# Load file into “chrome://tracing” on Google Chrome

# Inspect Trace From Command-Line

```
cd /root/chipyard/generators/gemmini/software/onnx-
runtime/systolic_runner/imagenet_runner/tools

# Inspect overall op statistics
python3 perfstats.py -t ../example_trace/resnet_inference_gemmini.json

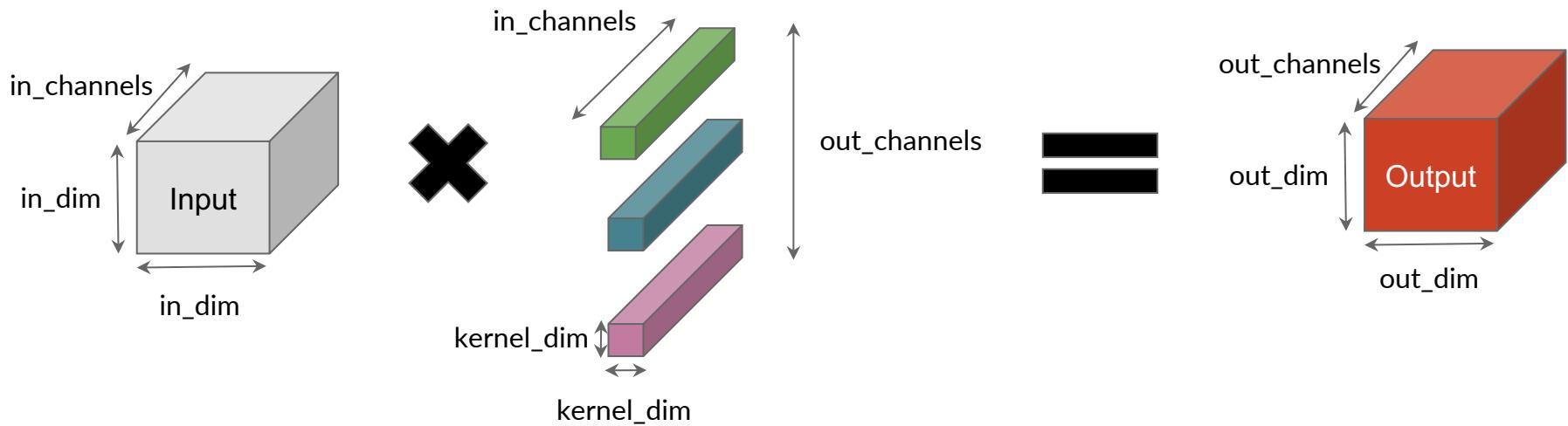
# Inspect layer-wise statistics
python3 perfstats.py -n ../example_trace/resnet_inference_gemmini.json

# Inspect accelerator vs CPU breakdowns
python3 perfstats.py -p ../example_trace/resnet_inference_gemmini.json

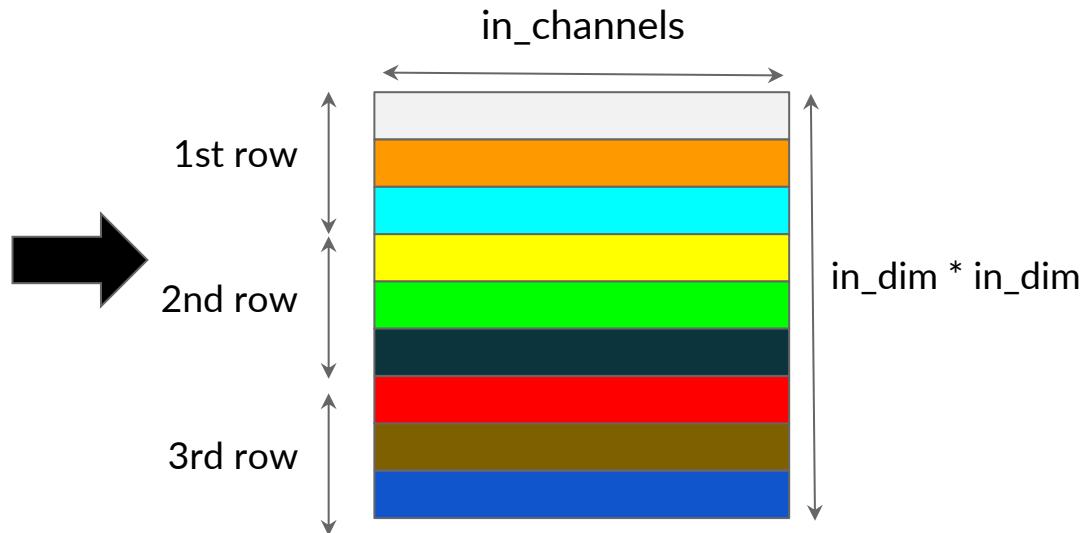
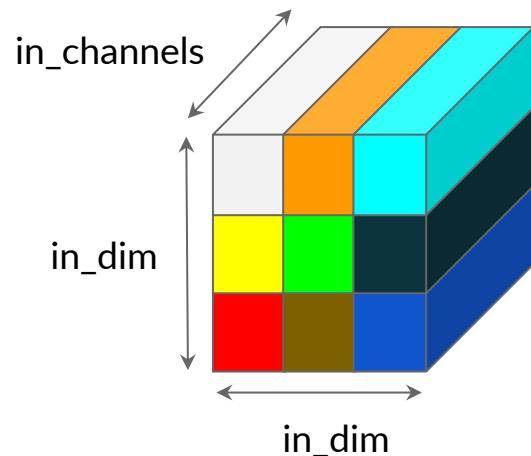
# Group by accelerator/CPU and op
python3 perfstats.py -po ../example_trace/resnet_inference_gemmini.json
```

# *Interactive Activity: ResNet50 First-Layer Optimizations With Low-Level ISA*

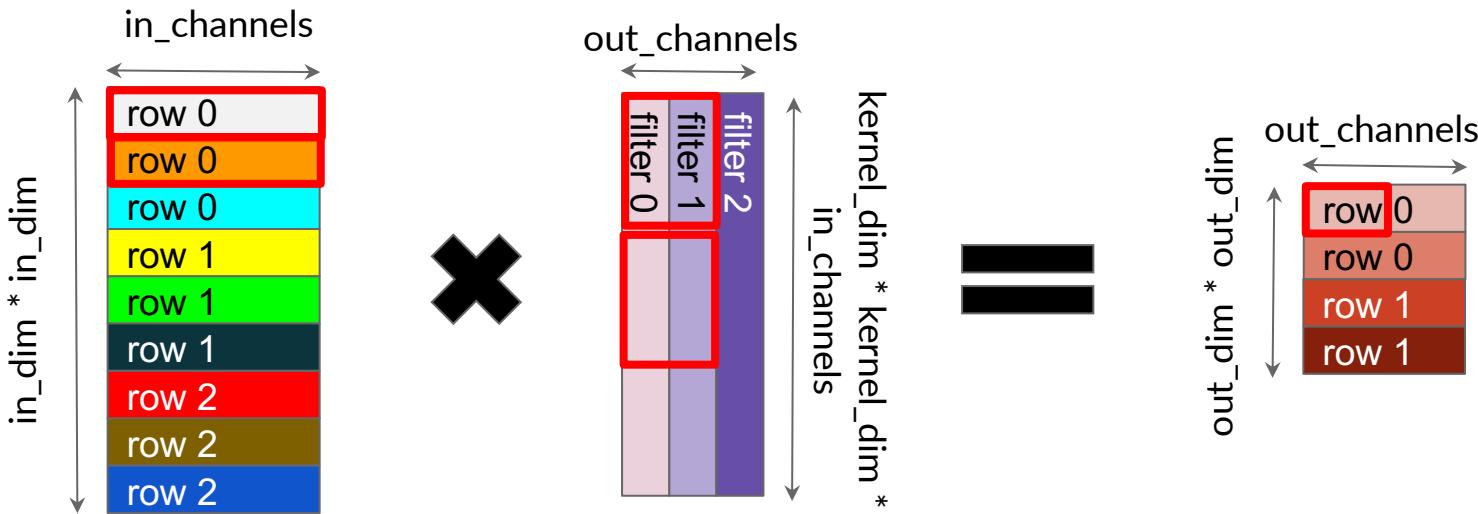
# Convolutions on Gemmini



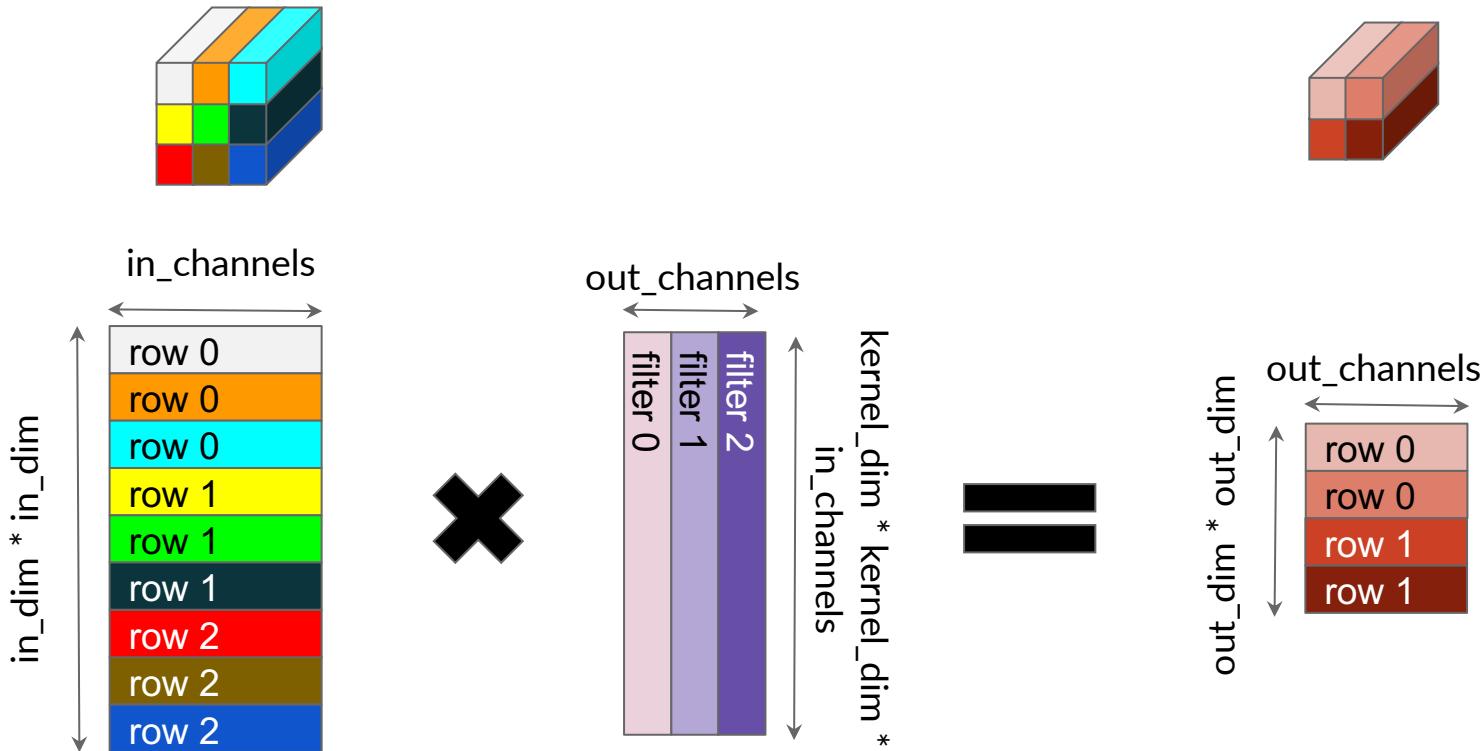
# How does Gemmini store inputs? (NHWC)



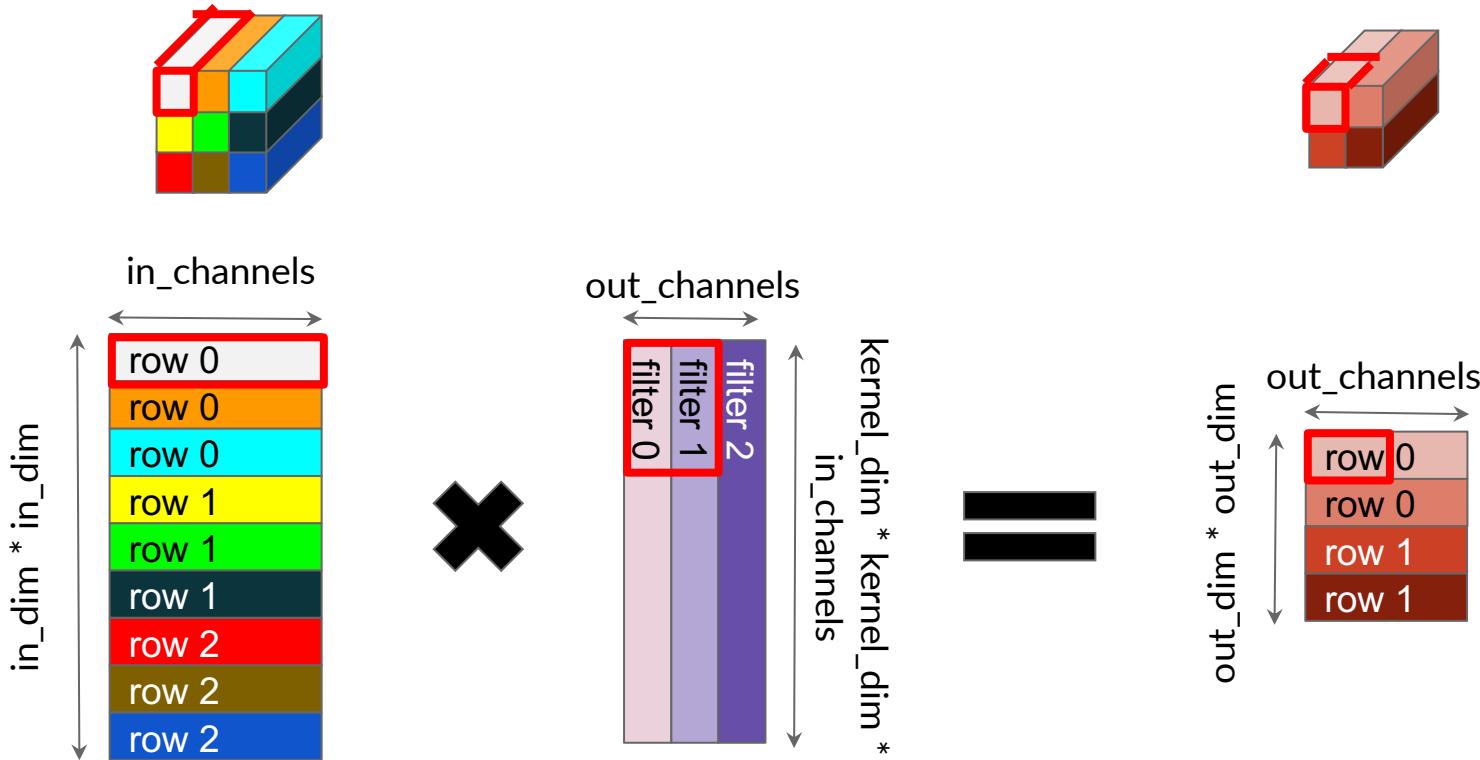
# Computing Convs with Matvecs (2x2 Conv)



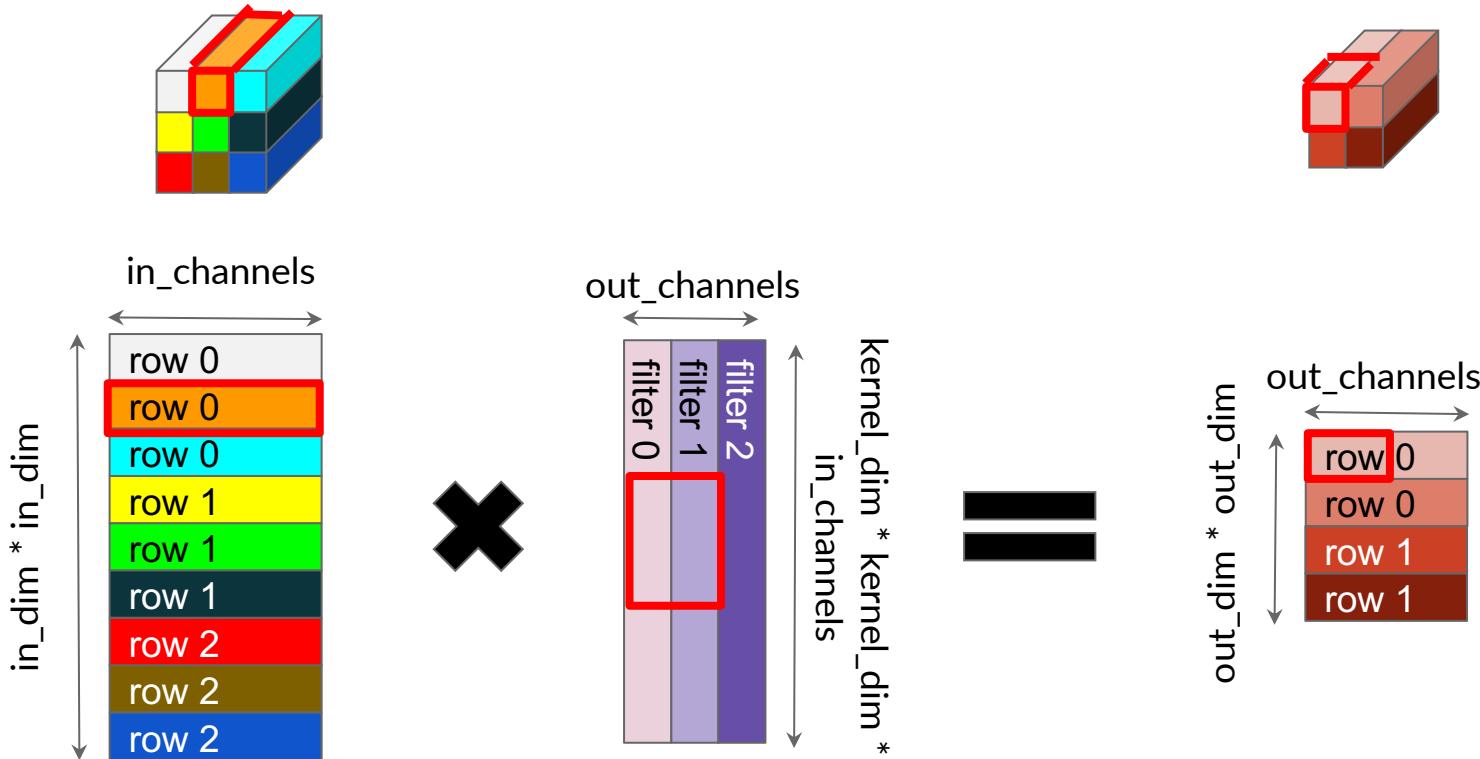
# Computing Convs with Matvecs (2x2 Conv)



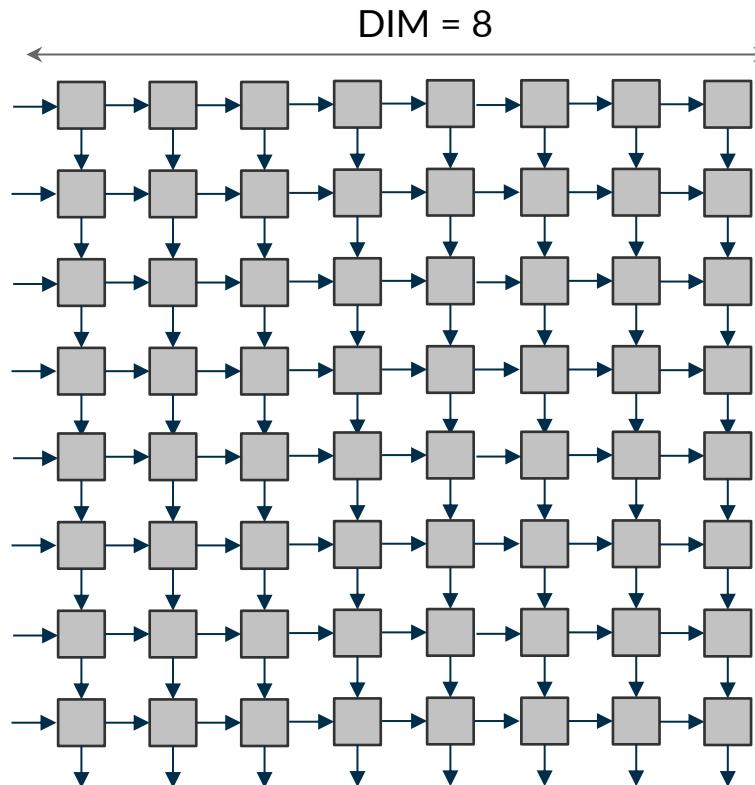
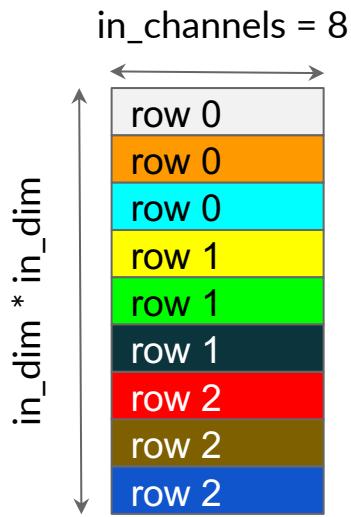
# Computing Convs with Matvecs (2x2 Conv)



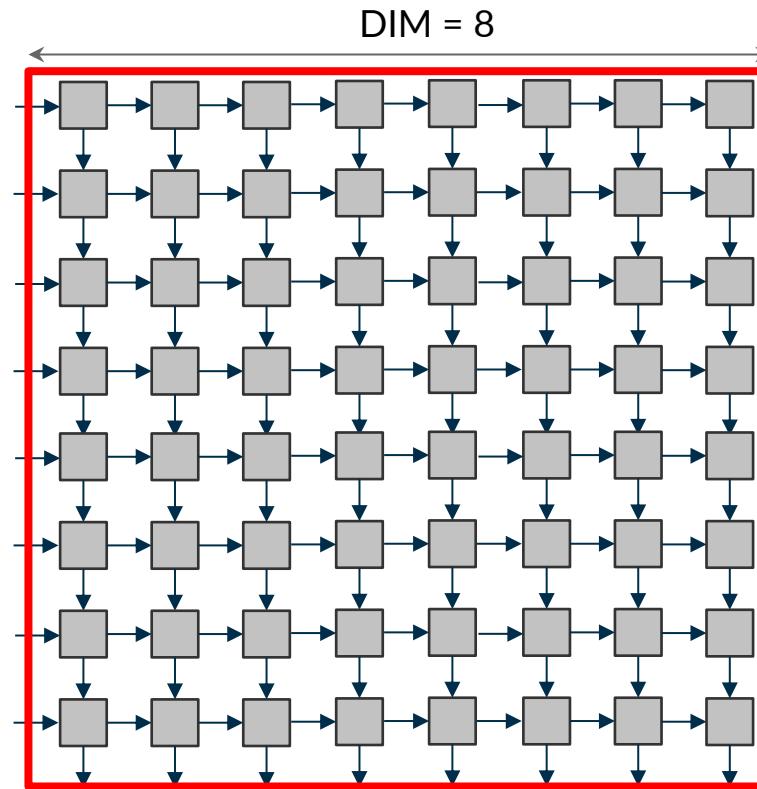
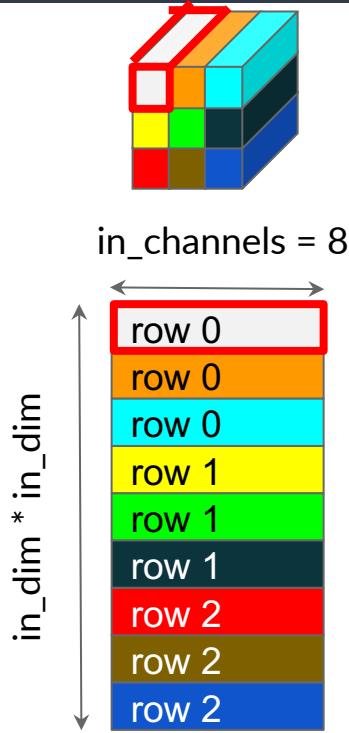
# Computing Convs with Matvecs (2x2 Conv)



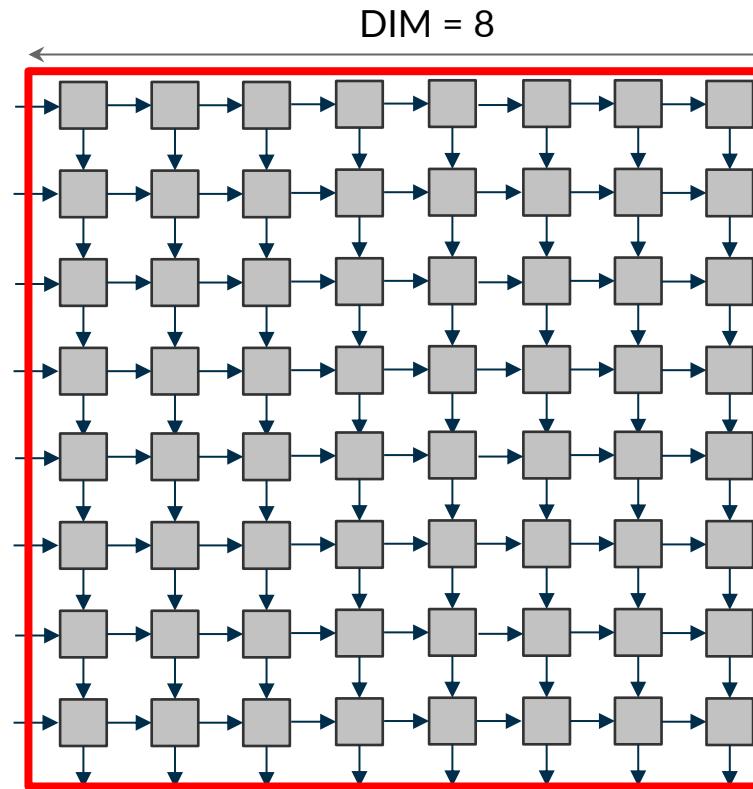
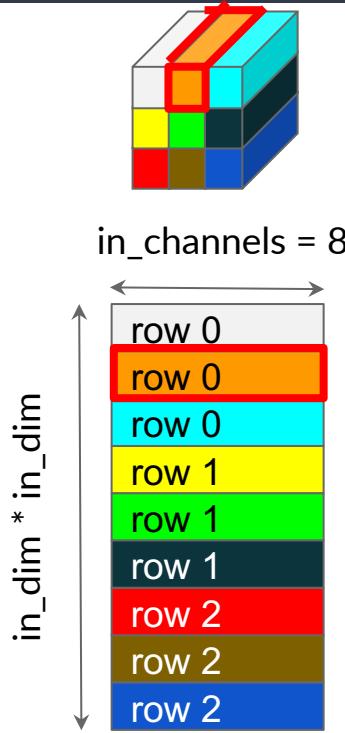
# What happens when in-channels is large?



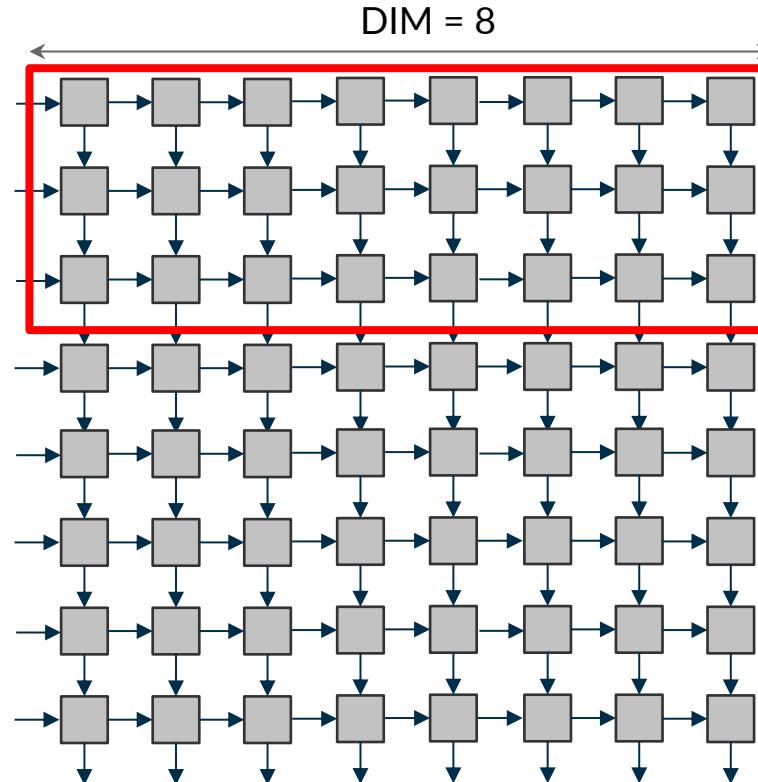
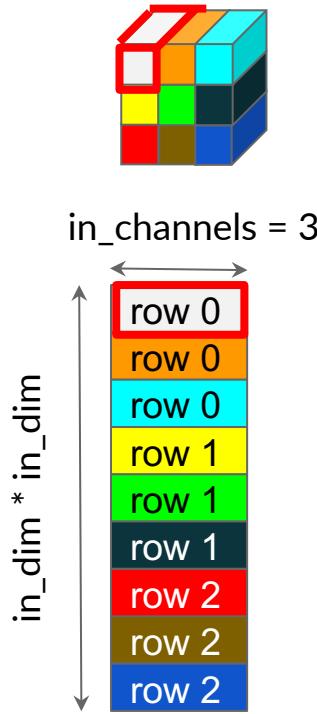
# What happens when in-channels is large?



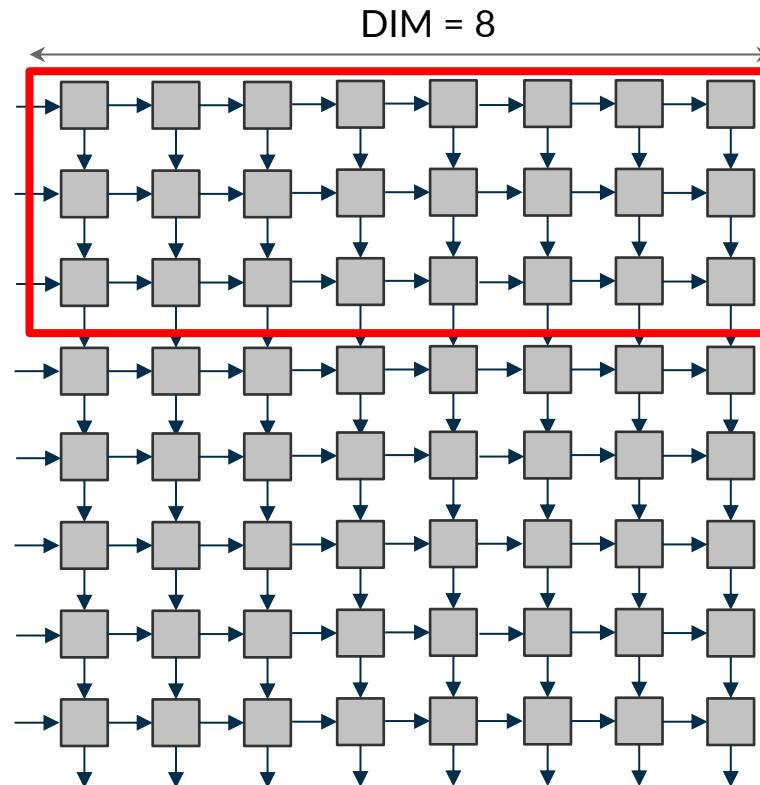
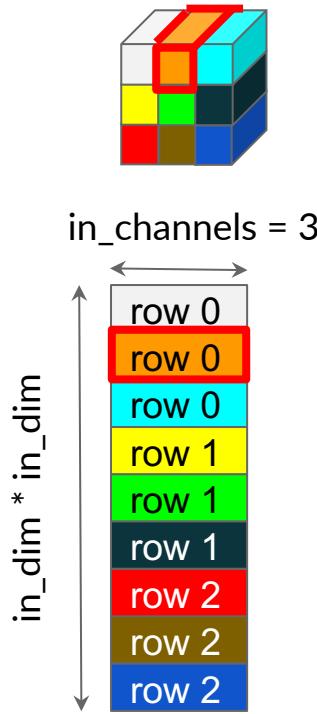
# What happens when in-channels is large?



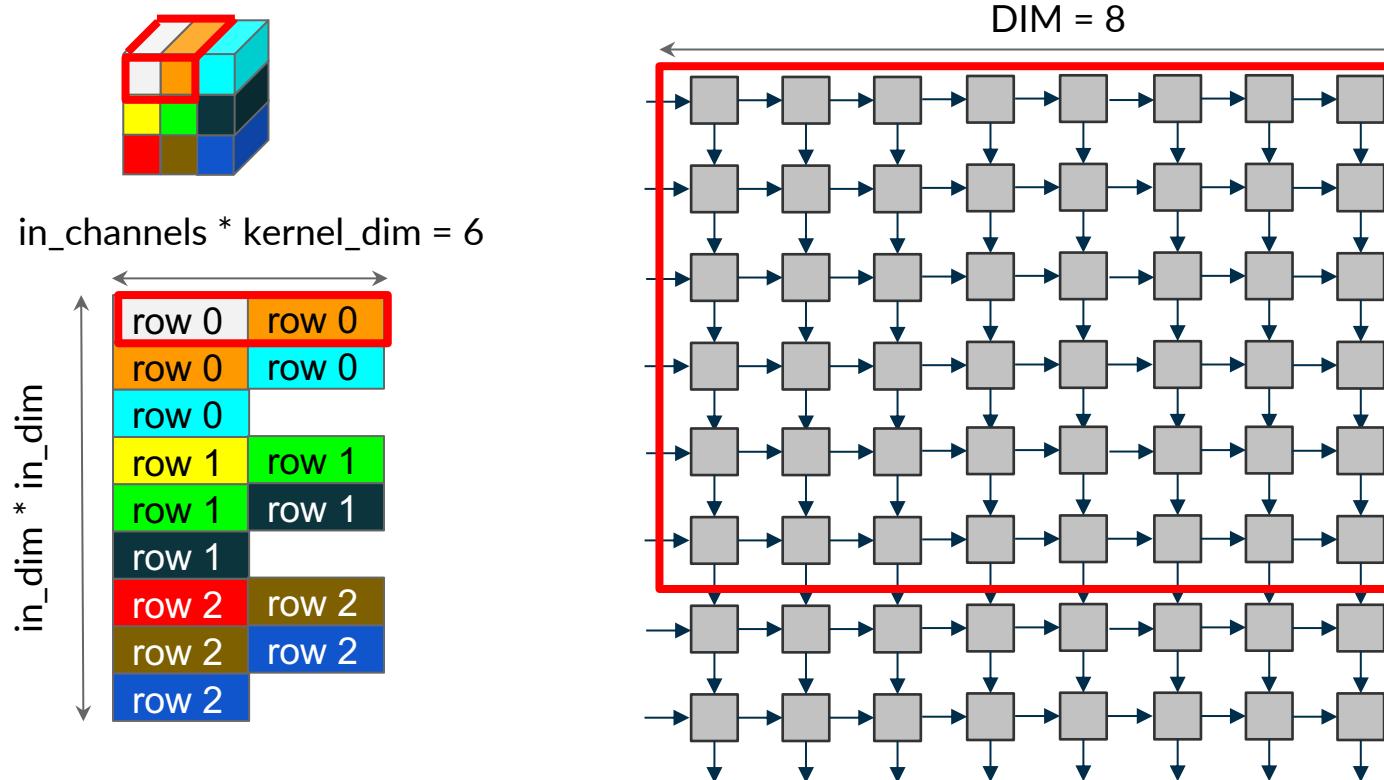
# What happens when in-channels is small?



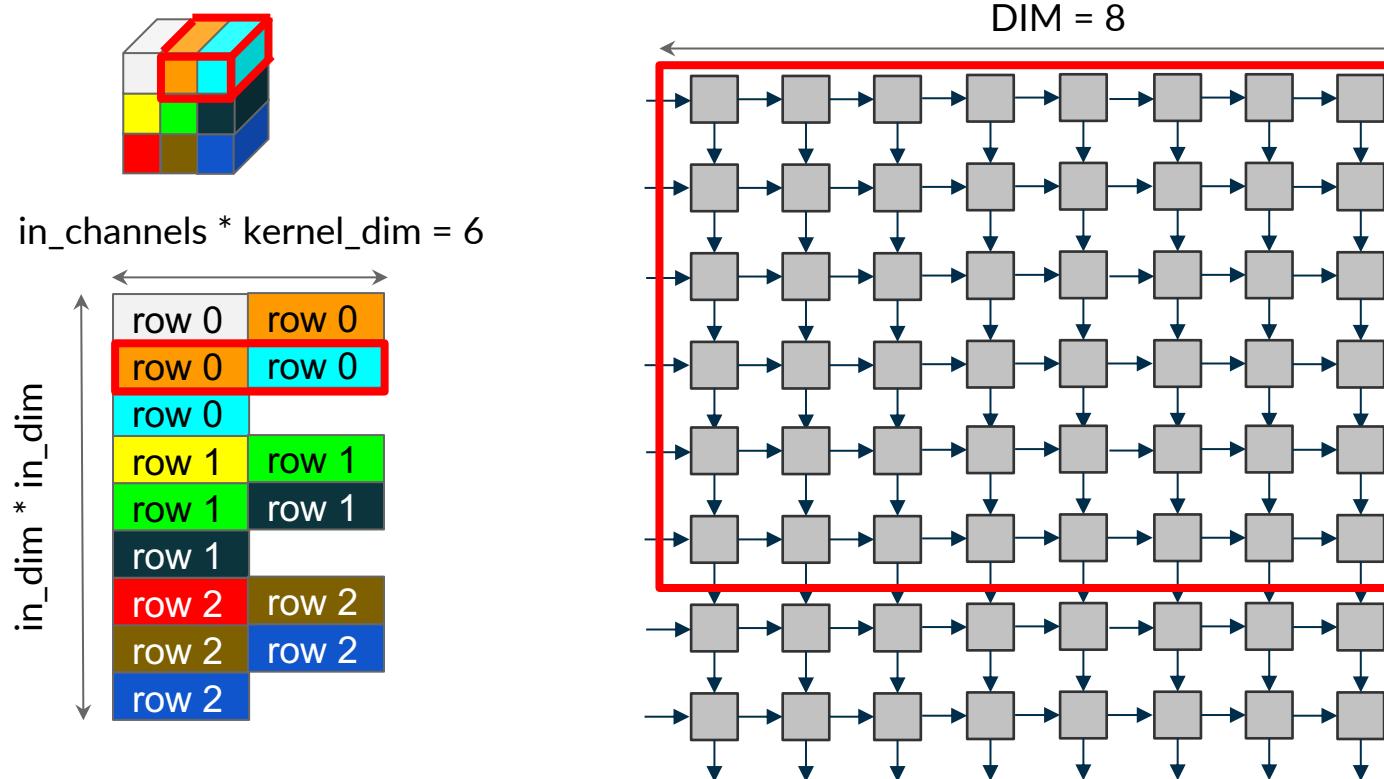
# What happens when in-channels is small?



# Workaround: Reduce over both in-channels and kernel-columns



# Workaround: Reduce over both in-channels and kernel-columns



# Optimize First Layer

```
cd /root/chipyard/generators/gemmini/software/gemmini-rocc-tests  
vim include/gemmini.h  
# Update "sp_tiled_conv_tutorial". Search for "TUTORIAL"
```

# Build Optimized First Layer Code

```
cd /root/chipyard/generators/gemmini/  
  
vim configs/GemminiCustomConfigs.scala # Use baseline config  
  
.scripts/build-spike.sh # Generate "gemmini_params.h"  
  
cd software/gemmini-rocc-tests  
  
.build.sh # Build binaries
```

# Run Optimized First Layer

```
cd /root/chipyard/generators/gemmini/  
./scripts/run-spike.sh conv_first_layer  
  
# Results from Verilator:  
# Before first-layer-opt: 23,198 cycles  
# After first-layer-opt: 15,215 cycles (52% speedup)
```

# Session 3: Performance Profiling

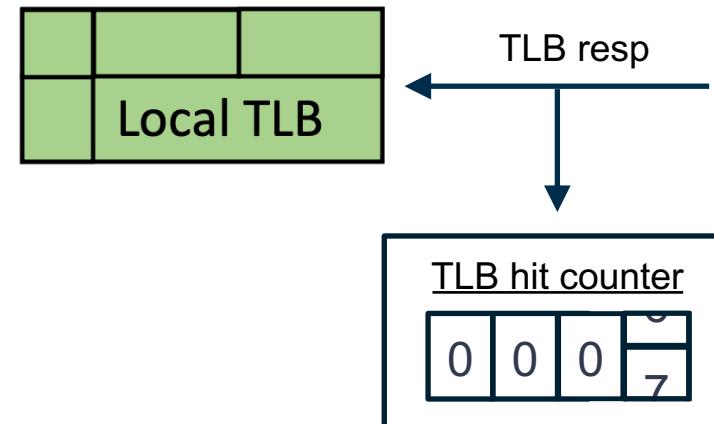
# Outline

- Types of Gemmini Counters
  - SoC Counters
  - Simulation Counters
- *Interactive Activities:*
  - Look at Simulation Logs
  - Evaluate Effect of TLB on DMA Performance
  - Use SoC Counters to Investigate Layer Bottlenecks
  - Parallelizing Data Collection

# Types of Performance Counters

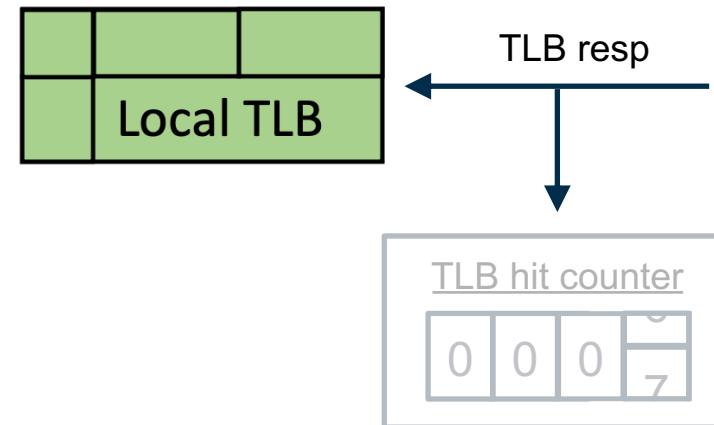
# Types of Performance Counters

- Counters that **exist on SoC**
  - Accessing these counters may affect performance/area/power
  - Useful for profiling actual chips



# Types of Performance Counters

- Counters that **exist on SoC**
  - Accessing these counters may affect performance/area/power
  - Useful for profiling actual chips
- Counters that **only exist in simulation**
  - Only exist during **Firesim** simulations
  - Don't affect performance/area/power

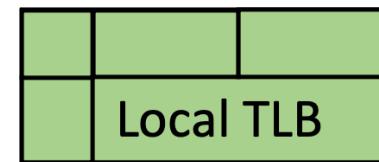


# SoC Counters

- Architect decides how many counters to add to RTL at elaboration-time
- Gemmini ISA:
  - read
  - reset
  - snapshot
    - Saves all counter values simultaneously in snapshot registers that can be read out afterwards
- Some are monotonic
  - E.g. TLB hits/misses
- Others are non-monotonic
  - E.g. Reservation station utilization

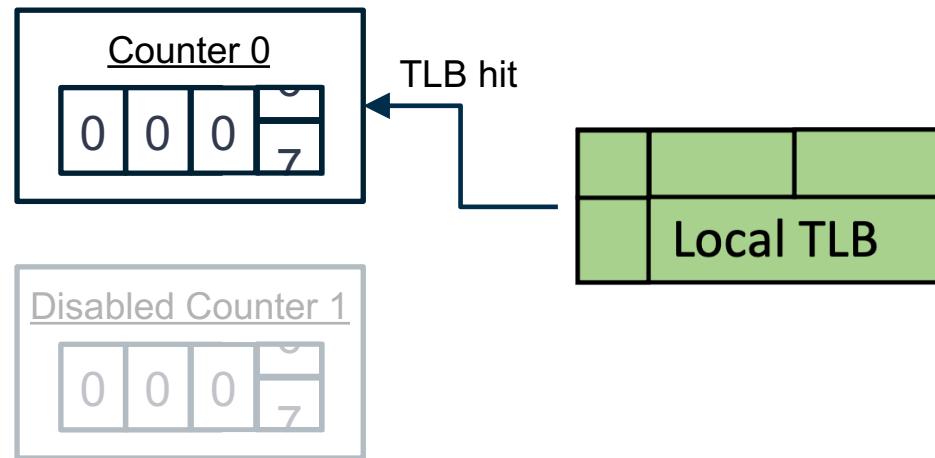
# SoC Counters: Initialization

- All counter begin “**disabled**”
  - Disabled counters count nothing
- Programmer decides which “**counter-code**” to enable each counter with
  - There are **counter-codes** for:
    - TLB hits
    - DMA latency
    - Etc.
- Once enabled, each counter begins incrementing whenever the signal associated with its **counter-code** is high
  - E.g., on every TLB hit



# SoC Counters: Initialization

- All counter begin “disabled”
  - Disabled counters count nothing
- Programmer decides which “counter-code” to enable each counter with
  - There are **counter-codes** for:
    - TLB hits
    - DMA latency
    - Etc.
- Once enabled, each counter begins incrementing whenever the signal associated with its **counter-code** is high
  - E.g., on every TLB hit



# Simulation Counters

- Only exist in **Firesim** simulations
- Accessible through logs
  - Monotonic counter logs
  - Non-monotonic counter logs
  - LLC memory stats logs
- Enable by setting `use_firesim_simulation_counters = true` in your config
- Firesim is an FPGA-accelerated simulation platform
  - Think Verilator but much faster



# Existing Gemmini Counters

- SoC counters
  - TLB miss rates / hit rates
  - Latency of Gemmini DMA requests
  - Bandwidth of Gemmini DMA requests
  - Reservation station utilization
    - Ld/st/ex
    - Issued vs unissued instructions
  - Estimated ex/ld/st overlap
  - Reasons why DMA is blocked
    - Blocked on TLB response
    - Blocked on TileLink port
- Simulation counters
  - All the SoC counters
  - LLC memory stats

# *Interactive Activity: Look at Simulation Logs*

# View Simulation Logs: Monotonic Counters

```
cd /root/chipyard/generators/gemmini/  
  
vim tutorial/resnet50/counter.log  
  
# Results from batch-4 ResNet50 inference  
  
# In Firesim, this will be called “AUTOCOUNTERFILE*”
```

# View Simulation Logs: Non-Monotonic Counters

```
cd /root/chipyard/generators/gemmini/  
  
vim tutorial/resnet50/printfs.log  
  
# Results from batch-4 ResNet50 inference  
  
# In Firesim, this will be called "synthesized-prints.txt"
```

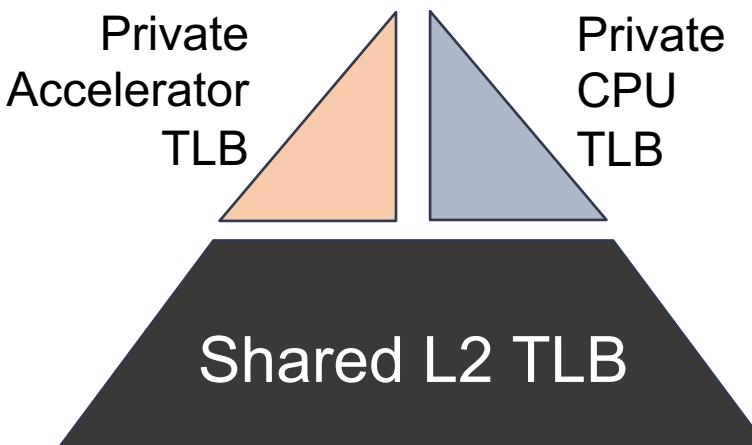
# View Simulation Logs: Memory Stats

```
cd /root/chipyard/generators/gemmini/  
  
vim tutorial/resnet50/memory_stats.csv  
  
# Results from batch-4 ResNet50 inference  
  
# In Firesim, this will be called "memory_stats.csv"
```

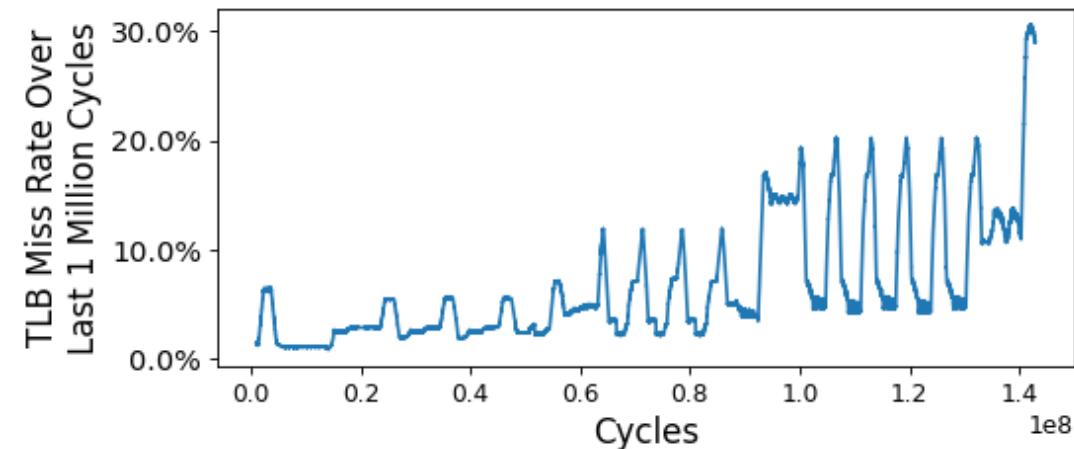
# *Interactive Activity: Evaluate Effect of TLB on DMA Performance*

# Virtual Memory for DNNs

Two-level TLB hierarchy

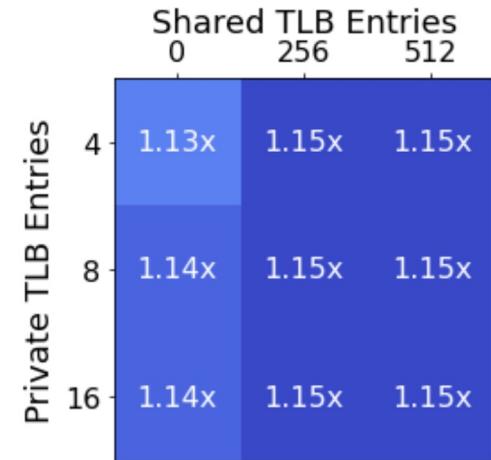
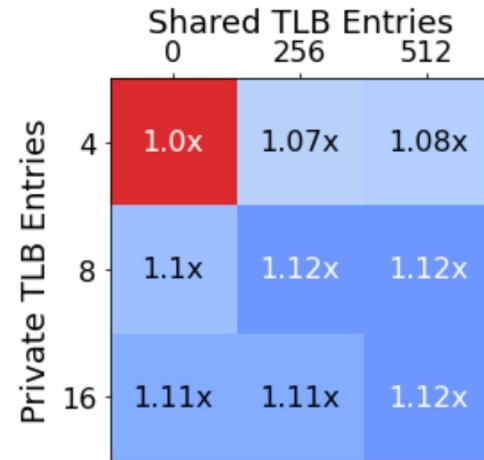


TLB Misses for ResNet50



# Virtual Memory for DNNs

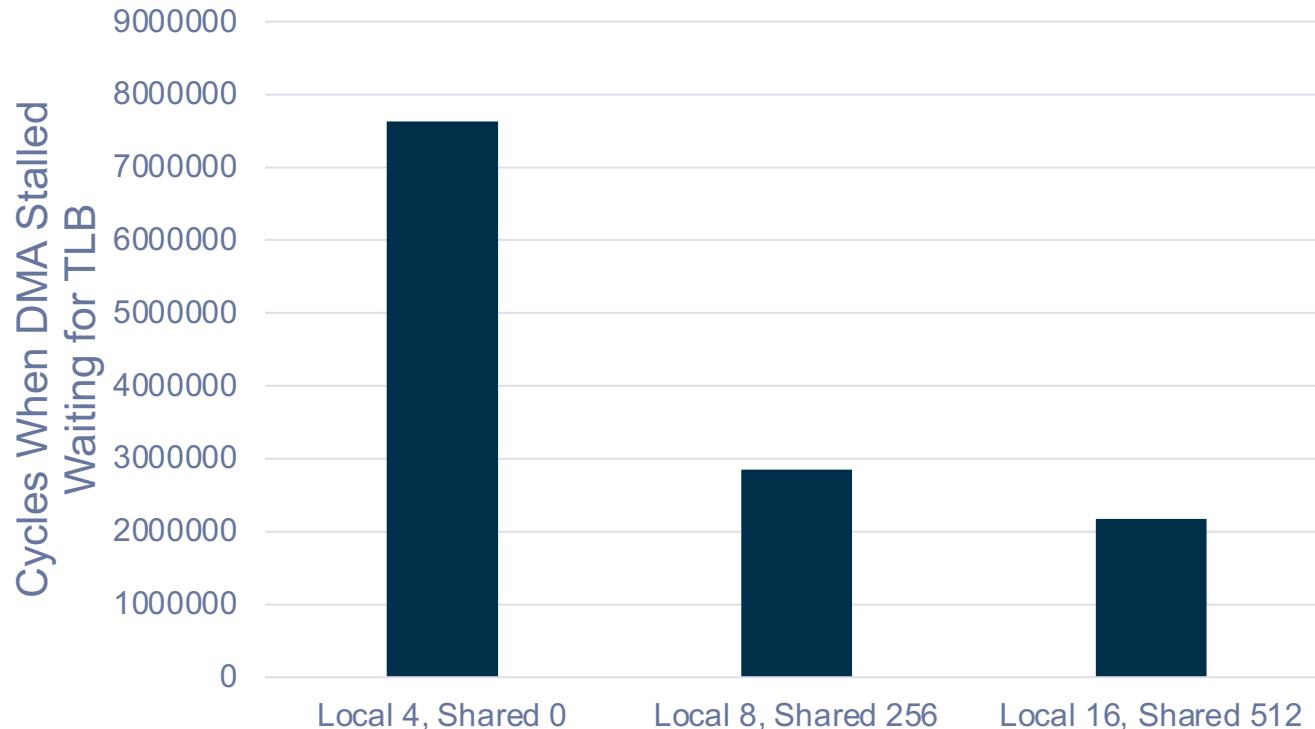
- Small private TLB much more impactful
- Low-cost optimizations:
  - Single-entry L0 TLB filters out consecutive TLB requests to same page



# View Simulation Logs: DMA Stalls Caused by TLB

```
cd /root/chipyard/generators/gemmini/  
  
# Look for "rdma_tlb_wait_cycles"  
  
vim tutorial/tlb/local_4_shared_0.log  
vim tutorial/tlb/local_8_shared_256.log  
vim tutorial/tlb/local_16_shared_512.log
```

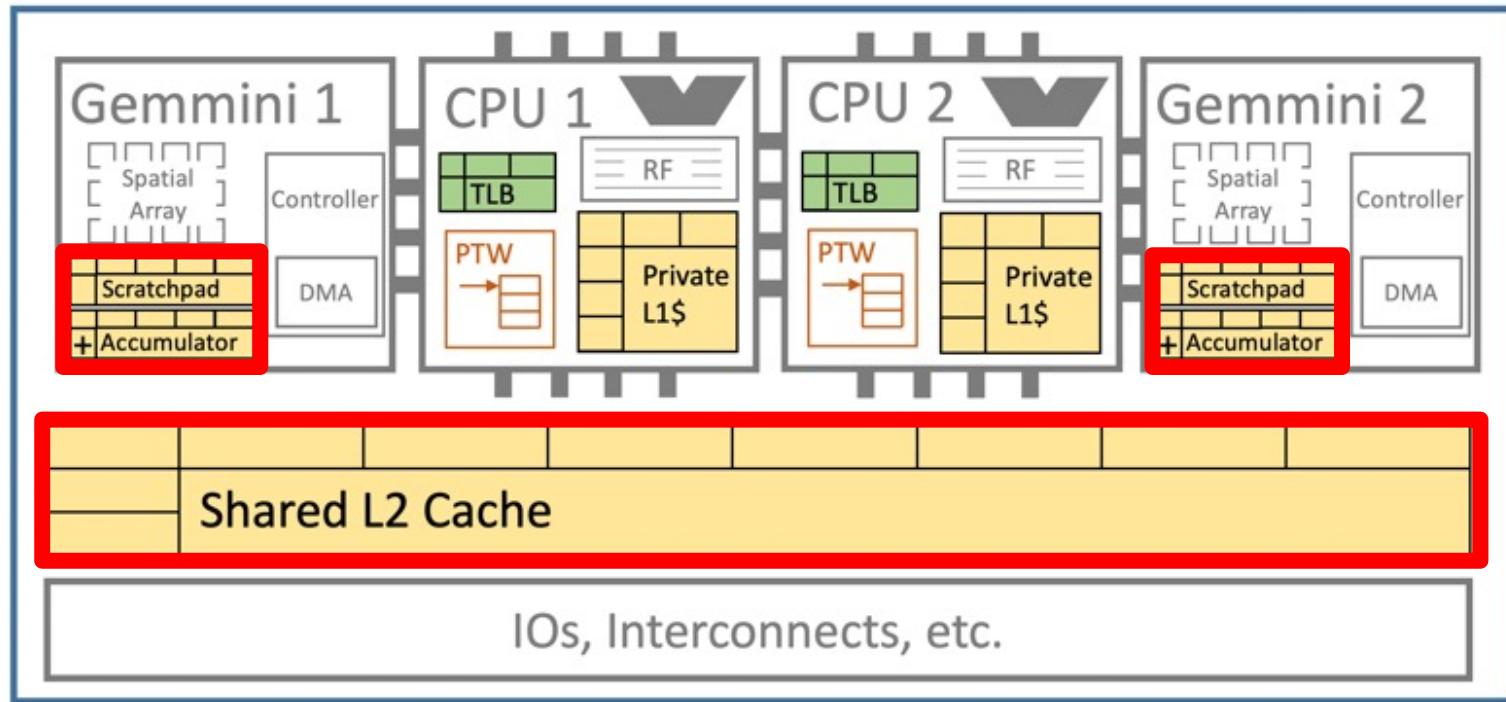
# View Simulation Logs: DMA Stalls Caused by TLB



# *Interactive Activity:* Use SoC Counters To Investigate Layer Bottlenecks

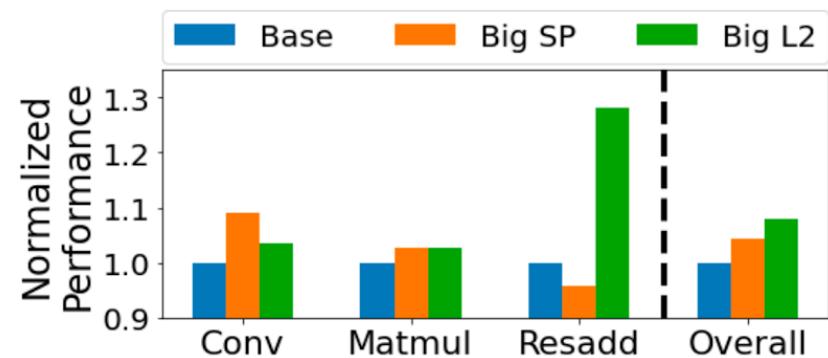
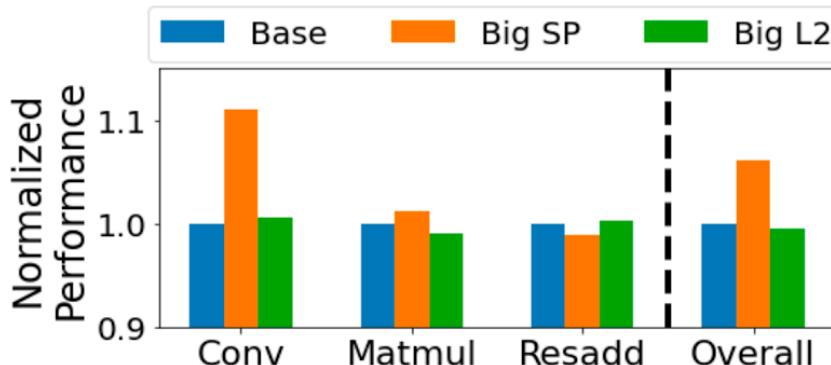
# DNN Memory Partitioning in Single- and Dual-Core Data-Parallel Inferences

SoC



# DNN Memory Partitioning in Single- and Dual-Core Data-Parallel Inferences

- Single core
  - Private scratchpad more helpful
  - Much better for convs
- Dual core
  - Shared L2 more helpful
  - Much better for residual additions



# Profile resadds with SoC counters

```
cd /root/chipyard/generators/gemmini/software/gemmini-rocc-tests  
vim imagenet/profile_resadd_tutorial.c  
  
# Add code to configure and read bandwidth and stall counters.  
# Search for “TUTORIAL”.  
  
.build.sh  
  
cd ~/chipyard/generators/gemmini/  
  
.scripts/run-spike.sh profile_resadd_tutorial
```

# Profile resadds with SoC counters

```
# Actual results from Firesim

# Single-core:
#   Resadd cycles: 2,033,152
#   RDMA bandwidth: 13 bytes/cycle
#   RDMA stall for L2 cycles: 1,603,959
#
# Dual-core (average):
#   Resadd cycles: 3,353,037
#   RDMA bandwidth: 8 bytes/cycle
#   RDMA stall for L2 cycles: 2,856,820
```

# Profile resadds with SoC counters

```
# Actual results from Firesim

# Single-core:
#   Resadd cycles: 2,033,152
#   RDMA bandwidth: 13 bytes/cycle
#   RDMA stall for L2 cycles: 1,603,959
#
# Dual-core (average):
#   Resadd cycles: 3,353,037 (65% slowdown)
#   RDMA bandwidth: 8 bytes/cycle (63% slowdown)
#   RDMA stall for L2 cycles: 2,856,820 (78% increase)
```

# *Interactive Activity: Simulate Many Workloads in Parallel*

# Parallel Simulation

- CPU-based cycle-accurate simulation is **slow**
  - Simulating ResNet50 on Verilator takes days!
  - Not everyone has access to FPGA-accelerated simulation (e.g. FireSim)
- *Alternative:* Run *many* workloads in parallel
  - E.g. simulate each layer in ResNet50 in Verilator on a separate core of your workstation/server

# Create Workload Template

```
cd /root/chipyard/generators/gemmini/software/gemmini-rocc-
tests/gemmini-data-collection

# The README.md has all the instructions we describe here

cp ../../bareMetalC/conv_first_layer.c templates/
vim templates/conv_first_layer.c

# Replace the BATCH_SIZE and OUT_CHANNELS parameters with:
#     #define BATCH_SIZE %BATCH_SIZE%
#     #define OUT_CHANNELS %OUT_CHANNELS%
```

# Specify Workloads to Run

```
vim tests.py
```

```
# Comment out existing tests.  
#  
# Add new tests:  
#     GemminiTest([“BATCH_SIZE”, “OUT_CHANNELS”],  
#                 [“1”, “4”],  
#                 “conv_first_layer”, “conv_first_layer_1_4”),  
#  
#     GemminiTest([“BATCH_SIZE”, “OUT_CHANNELS”],  
#                 [“4”, “64”],  
#                 “conv_first_layer”, “conv_first_layer_4_64”)
```

# Run Workloads

```
bash config_gen_data.sh baselineInferenceConfig verilator  
  
# Parameter 1: Choose the hardware config from  
#           “configs/CustomConfigs.scala”  
#  
# Parameter 2: Can be “vcs”, “verilator”, or “midas”  
  
# Ctrl-C; this will take a while to run  
  
cd /root/chipyard/generators/gemmini/  
./tutorial/checkpoint.sh data-collection
```

# Inspect Outputs

```
cd chipyard/generators/gemmini
```

```
# View cycle counts:  
vim data-collection-output-configs/data-collection-output-cycles-  
verilator-baselineInferenceConfig/conv_first_layer_1_4-  
verilator.txt  
  
vim data-collection-output-configs/data-collection-output-cycles-  
verilator-baselineInferenceConfig/conv_first_layer_4_64-  
verilator.txt
```

# Inspect Outputs

```
cd chipyard/generators/gemmini
```

```
# View tiling factors:
```

```
vim data-collection-output-configs/data-collection-output-tiling-  
factors-baselineInferenceConfig/conv_first_layer_1_4-spike.txt
```

```
vim data-collection-output-configs/data-collection-output-tiling-  
factors-baselineInferenceConfig/conv_first_layer_4_64-spike.txt
```

```
# Gemmini's matmul and conv functions can print tiling factors  
# when performing parallel data collection
```

# Clean Intermediate Files

```
cd /root/chipyard/generators/gemmini/software/gemmini-rocc-
tests/gemmini-data-collection

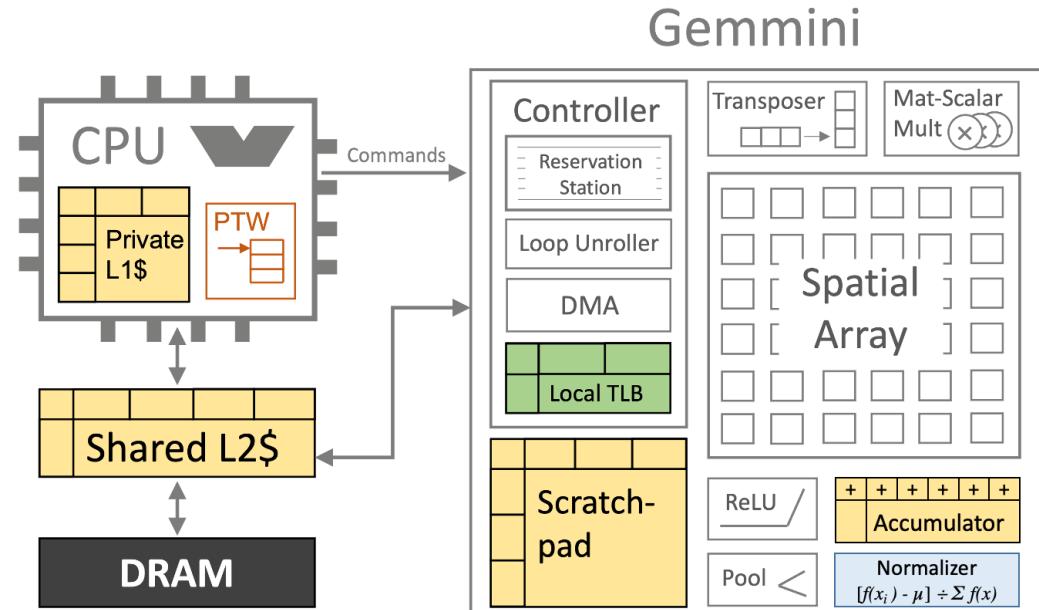
# Remove intermediate files, but keep the simulation outputs

bash clean.sh
```

# Parting Thoughts

# Summary

- Gemmini is a full-system, full-stack DNN accelerator evaluation platform
- Tune different components of full SoC
- Different levels of the programming stack exposed for different use-cases



# Future Plans

- Incorporate sparse accelerators
  - Sparse accelerators cover a much wider domain than dense accelerators
    - Sparse data formats, sparse NoCs, workload distribution schemes, etc.
- Continue performance improvements
  - Tune for training workloads
    - Offload batch-norms, etc. off of CPU
- Continue power/area improvements

# Thanks for Attending!



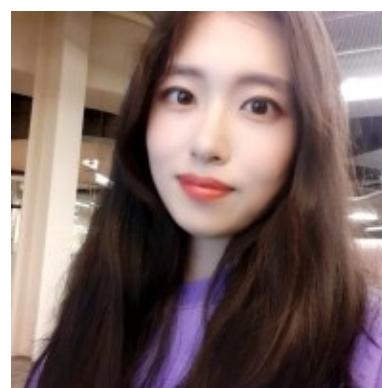
Hasan Nazim Genc

[hngenc@berkeley.edu](mailto:hngenc@berkeley.edu)



Simon Guo

[simonguo@berkeley.edu](mailto:simonguo@berkeley.edu)



Seah Kim

[kimsea8209@berkeley.edu](mailto:kimsea8209@berkeley.edu)



Vadim (Dima) Nikiforov

[vnikiforov@berkeley.edu](mailto:vnikiforov@berkeley.edu)

# Questions?