


COMPUTER ARCHITECTURE AND SYSTEMS LAB

# NPUWattch Tutorial: Basic Usage and Application to PyTorchSim

**Sehyeon Kim and William J. Song**
School of Electrical and Electronic Engineering
Yonsei University



**MICRO 2026**

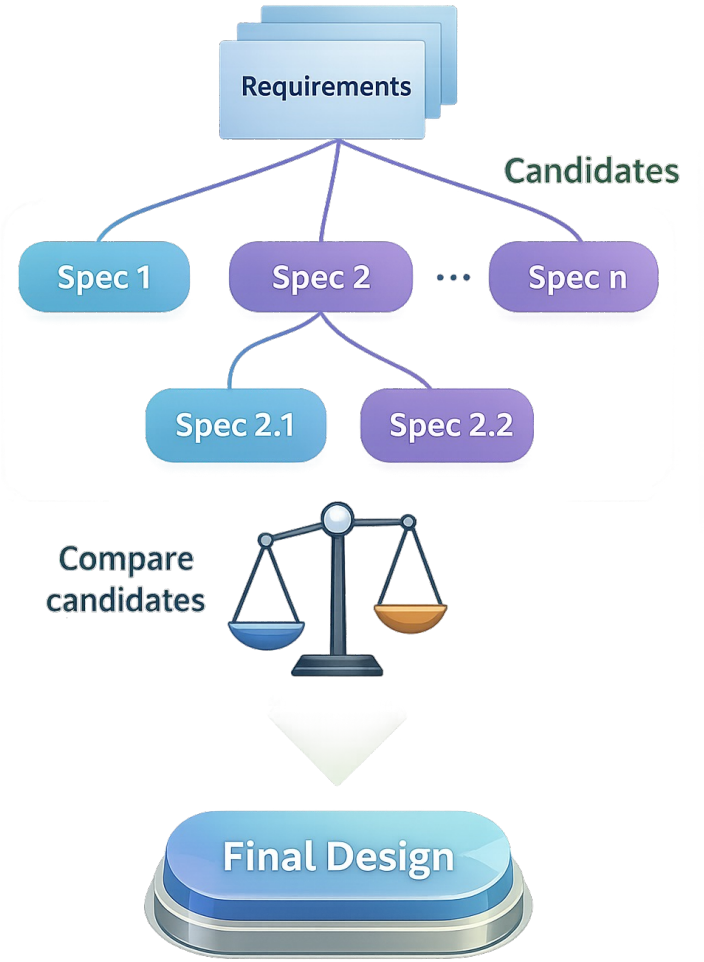
*59th IEEE/ACM International Symposium on Microarchitecture®*

InterContinental Athenaeum Athens Hotel
Sunday, November 1, 2026
8:00 AM EET – 12:00 PM EET
Arcade I (NPU Modeling Tutorial)

**The following introductory slides are adapted from our HPCA'26 presentation of the paper "NPUWattch: ML-based Power, Area, and Timing Modeling for Neural Accelerators."**

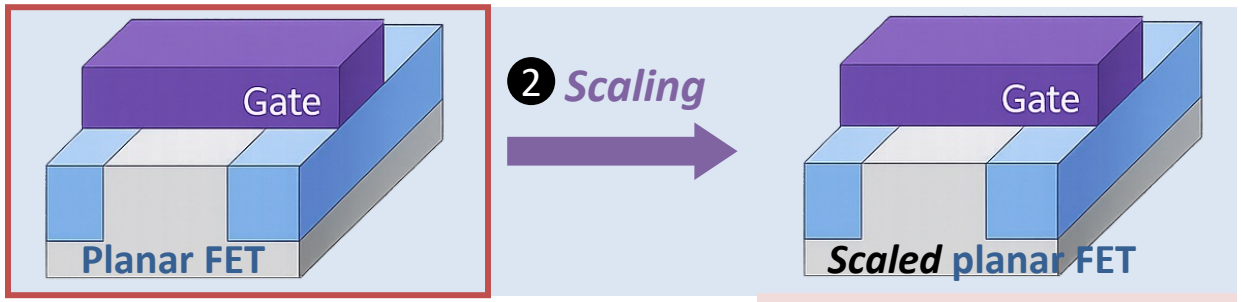
**A detailed tutorial on NPUWattch will follow this introduction.**

# Introduction: Modeling Neural Accelerators




- **Neural accelerators** have become one of the major architectural research topics.
- Full **tape-out** design procedure
  - ➡ Too costly for **early design space exploration** (DSE)
- **Power, area, and timing (PAT) models**
  *(Examples: CACTI, Aladdin, Accelergy)*
  - ➡ **Efficiency** and **accuracy** tradeoff
- Traditional **rule-based models** often lack sufficient details.
  - ➡ **Inaccurate** pre-silicon design analysis

# Background: Common Practice for Scaling Design Costs




❶ Design cost *measured* at an old process node *(e.g., 45nm)*

❸ Design cost *estimated* at a new process node *(e.g., 10nm)*

**!= Mismatch**

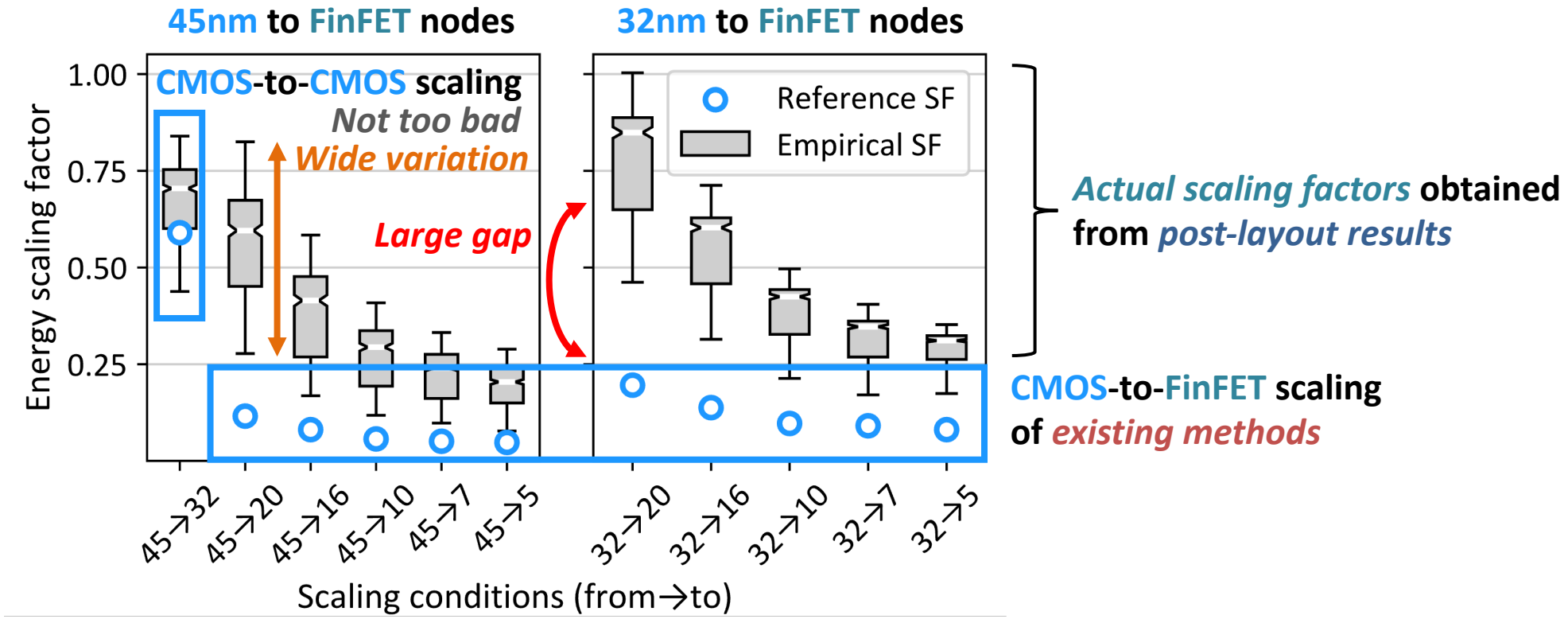
❹ *Actual* design cost at the new process node

**Simply scaling *CMOS design costs* to *emerging FinFET/GAAFET nodes* does not work!**

# Motivation: Three Pitfalls in Accelerator Modeling

## *Pitfall A.* ***Technology Scaling Factors: One Size Never Fits All***

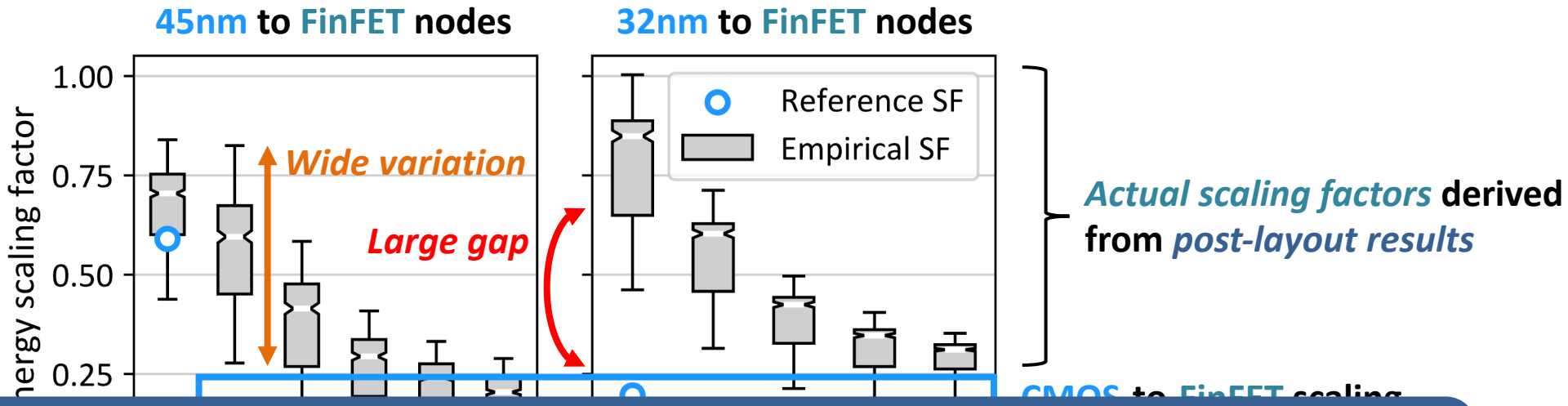
- A *single scaling factor* derived from a *small reference circuit* does not provide accurate estimations for *complex designs* at a *new technology node*.



# Motivation: Three Pitfalls in Accelerator Modeling

## *Pitfall A.* ***Technology Scaling Factors: One Size Never Fits All***

- A *single scaling factor* derived from a *small reference circuit* does not provide accurate estimations for *complex designs* at a *new technology node*.



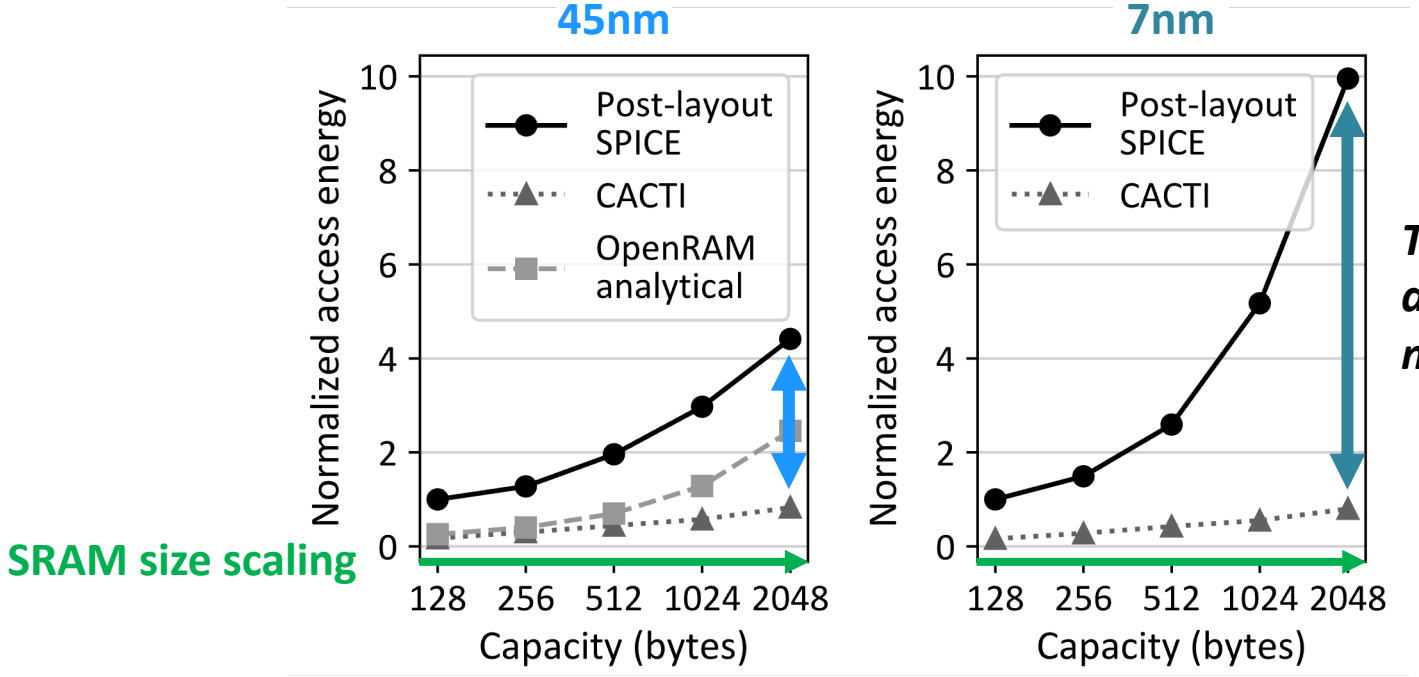

**A *single scaling factor* cannot correctly represent *design variations* and technology scaling to *newly emerging process nodes*.**

# Motivation: Three Pitfalls in Accelerator Modeling

## *Pitfall B.* ***Nonlinearity in Design Scaling***

- Traditional *analytical* or *rule-based models* fail to capture *nonlinearity* in design scaling.

  ➡ Even identical designs *do not scale the same way* at different technology nodes.



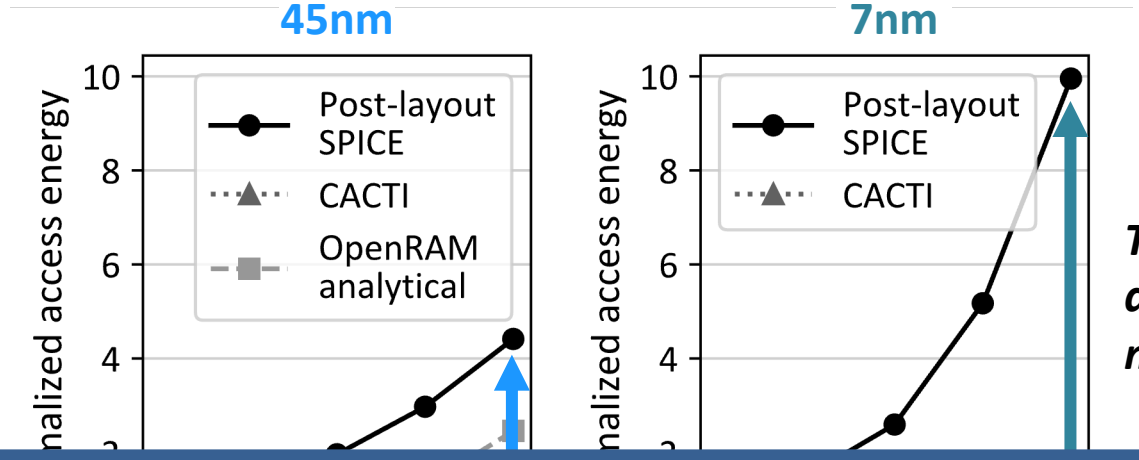

***The difference between post-layout results and rule-based models (e.g., CACTI) is more pronounced in the advanced nodes.***

# Motivation: Three Pitfalls in Accelerator Modeling

## *Pitfall B.* ***Nonlinearity in Design Scaling***

- Traditional *analytical* or *rule-based models* fail to capture *nonlinearity* in design scaling.

  ➡ Even identical designs *do not scale the same way* at different technology nodes.




***The difference between post-layout results and rule-based models (e.g., CACTI) is more pronounced in the advanced nodes.***

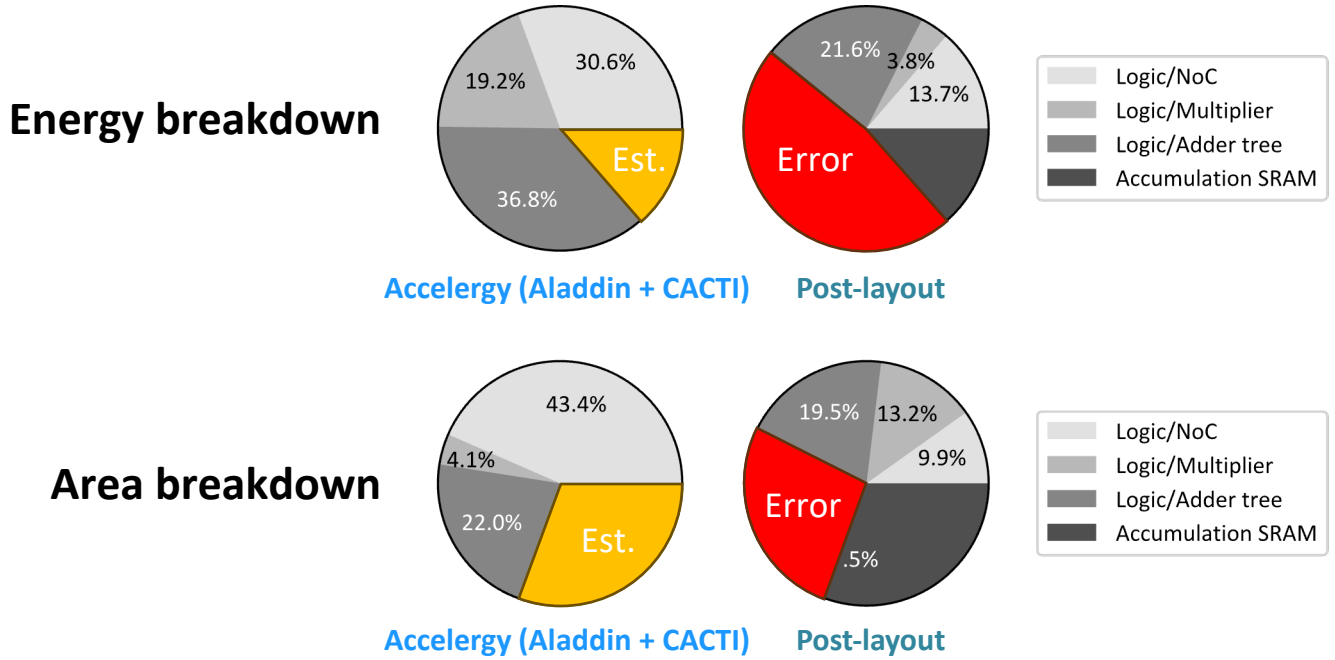
***Traditional rule-based models*** **cannot correctly capture** ***nonlinearity*** **in design and technology scaling.**

# Motivation: Three Pitfalls in Accelerator Modeling

## *Pitfall C.* ***Inconsistency in Heterogeneous Modeling***

- Monolithic-die modeling requires a *common calibration point* (i.e., transistor model) between **logic** and **SRAM** for *consistency*.

  ➡ The mixed use of heterogeneous models (e.g., Aladdin + CACTI) introduces *bias*.



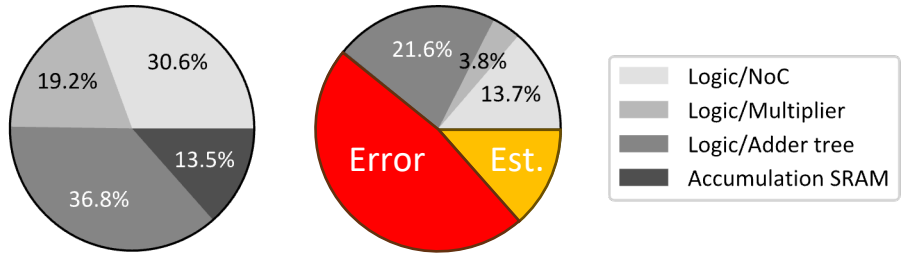
# Motivation: Three Pitfalls in Accelerator Modeling

## *Pitfall C.* ***Inconsistency in Heterogeneous Modeling***

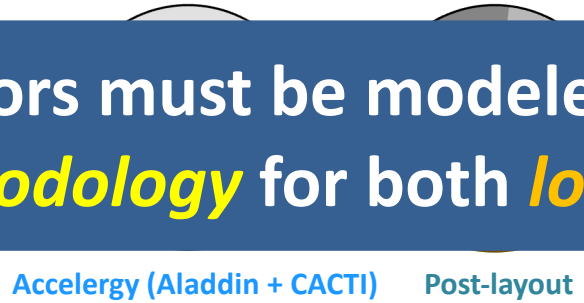
- Monolithic-die modeling requires a *common calibration point* (i.e., transistor model) between **logic** and **SRAM** for *consistency*.

  ➡ The mixed use of heterogeneous models (e.g., Aladdin + CACTI) introduces *bias*.

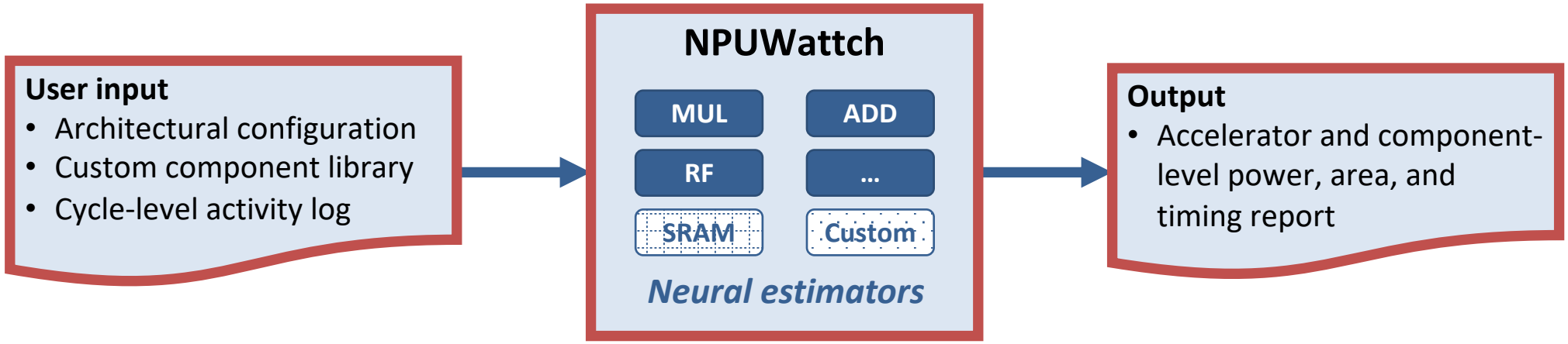
**Energy breakdown**




**Neural accelerators must be modeled using a *coherent design methodology* for both *logic* and *SRAM*.**

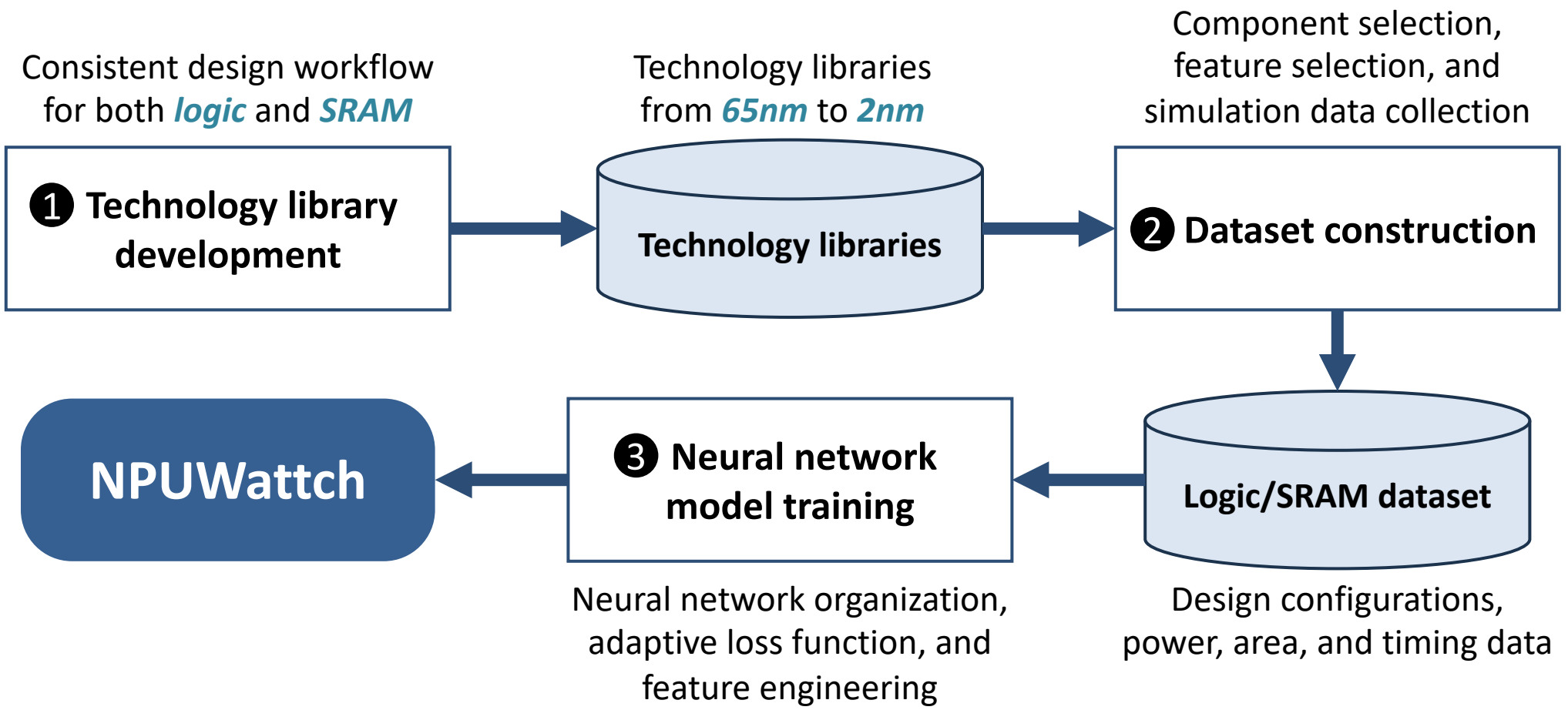


# NPUWattch: Framework Workflow




**NPUWattch provides accurate design cost estimates through *neural estimators* trained on carefully curated *post-layout data*.**

# NPUWattch: Overall Design Procedure

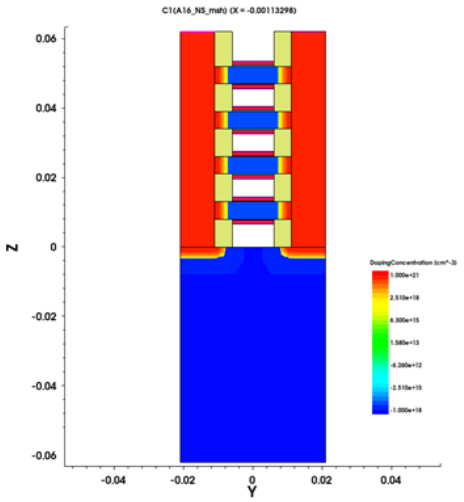


# ❶ Technology Library Development

**Transistor Modeling:**

- We *modeled 3D transistors* that conform to *scaling trends*, while considering doping concentration, thermal diffusion, and related physical effects.

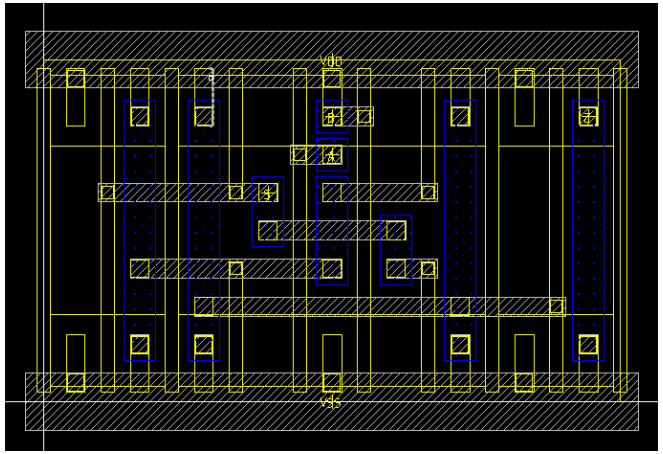
*Doping Concentration Map*




**Technology Library Building:**

- We developed *multi-corner*-characterized *technology libraries*, including design rules, logic cells, and SRAM periphery circuits.

*Standard Cell Layout*

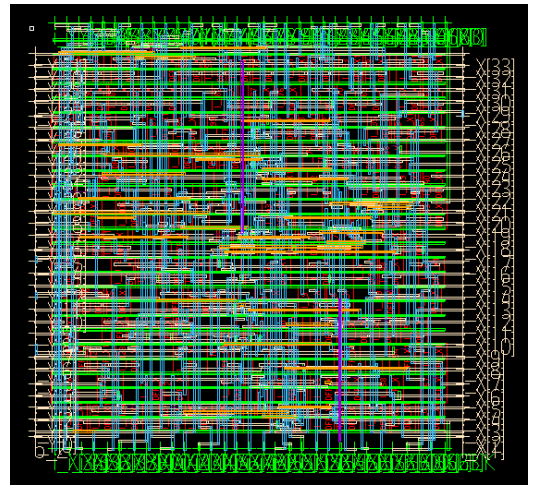


# ❷ Dataset Construction

**Digital Logic:**

- We performed *post-layout simulations* on functional RTL designs to extract power, area, and timing information.

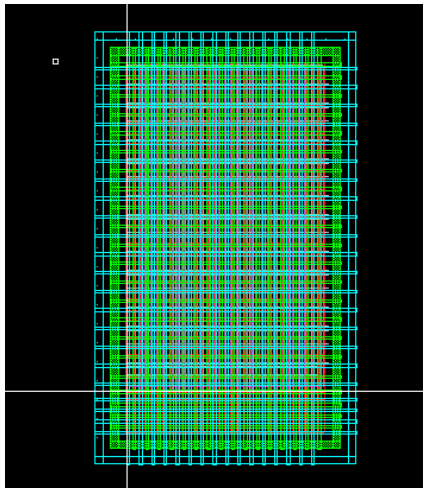
*Digital Circuit Layout*



**SRAM:**

- We compiled SRAM arrays of various sizes and performed *SPICE simulations* to evaluate their characteristics.

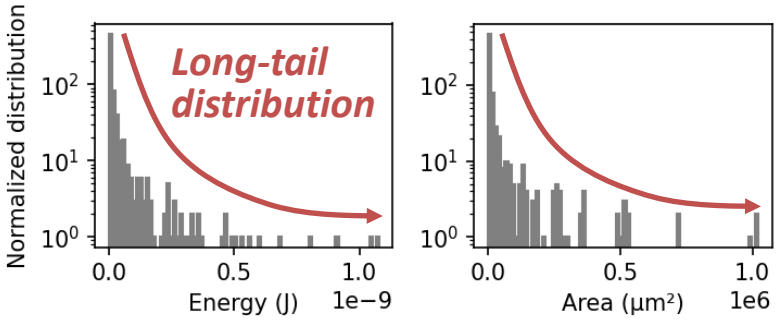
*SRAM Array Layout*



# ❸ Neural Network Model Training

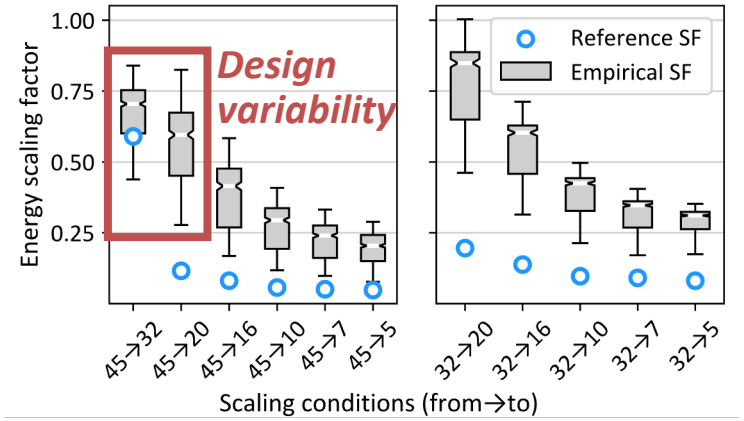
**Neural Network Model:**

- *Lightweight MLPs* for fast runtime inference (~0.4ms)
- A *dedicated neural network cluster* for each component




**Neural Network Training:**

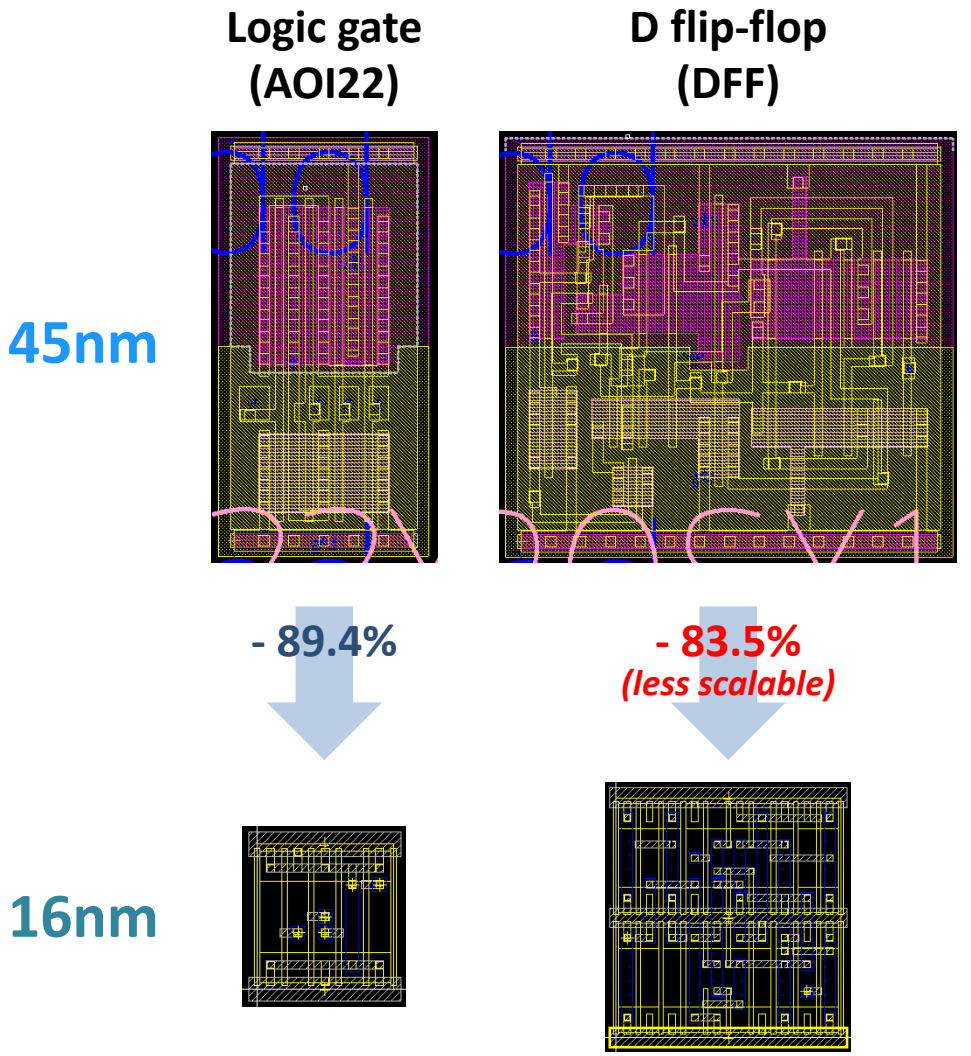
- An *adaptive loss function*, which reweights uneven sample distributions to address the *long-tail distribution problem*
- New *design-aware input features** to address *variability* in technology scaling.



# Neural Network Modeling and Training

## Design-Aware Input Features:

- NPUWattch introduces two new *design-aware input features*, ***S**equential cell **C**ount **R**atio (**SCR**)* and ***S**equential cell **A**rea **R**atio (**SAR**)*.
- This approach is motivated by the observation that ***flip-flops*** exhibit different power and area scaling than ***logic gates*** due to *clock trees*, *latch analog effects*, and *wiring complexity*.



# Experimental Setup

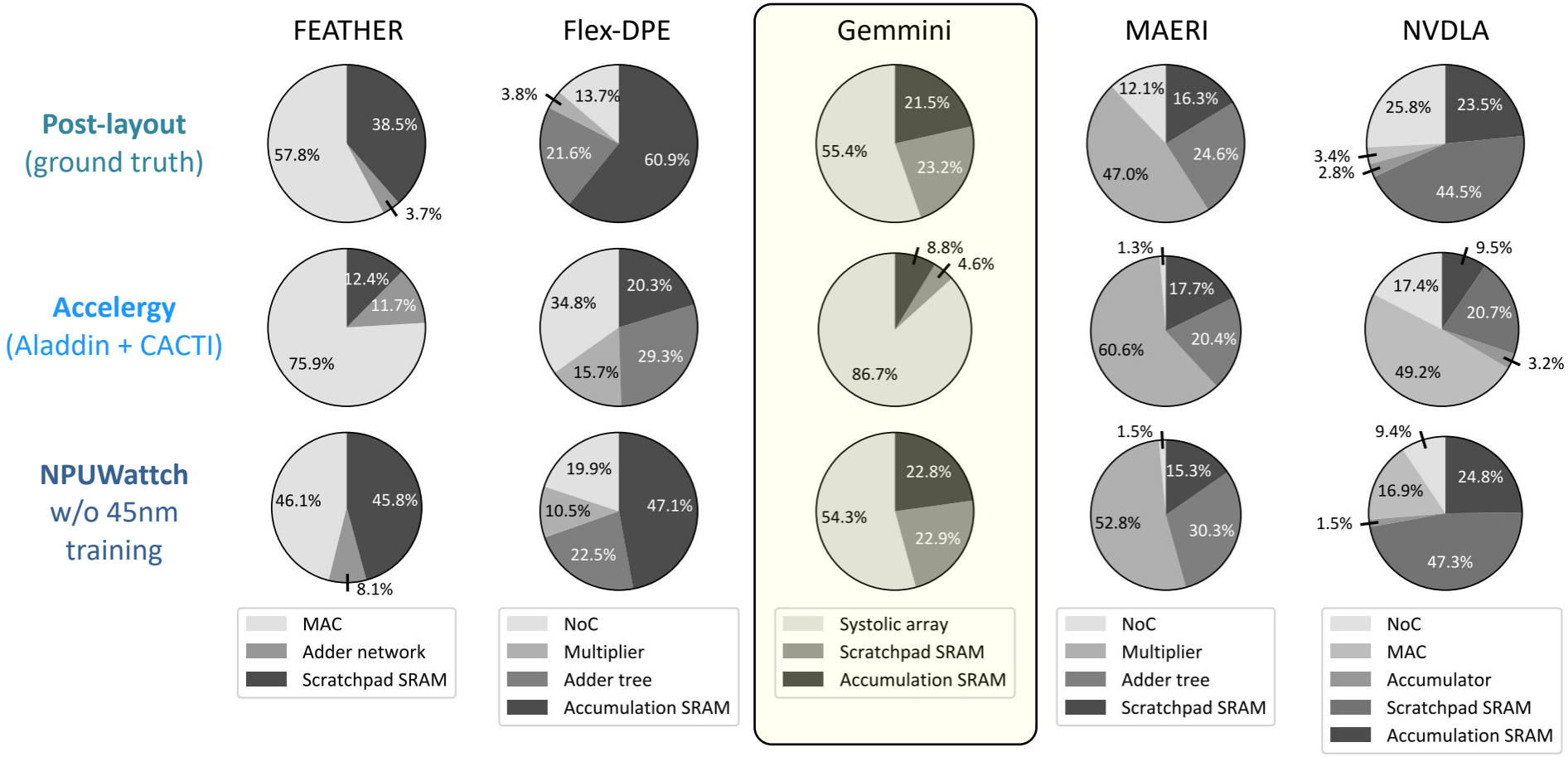
- **Modeling Frameworks:**
  **Accelergy** (Aladdin table + CACTI) at 45nm vs. **NPUWattch** without 45nm training
- **Post-layout Simulation Setup:** Nangate 45nm cell library + OpenRAM with FreePDK45
- **Accelerator Designs:** five *open-source RTL* accelerators

| Accelerator | Data format | PEs | Network topology | Global buffer |
|---|---|---|---|---|
| FEATHER | int8 | 16 x 16 | Spatial/adder tree | 256KB |
| Flex-DPE | bfloat16 | 1 x 128 | Adder tree | 128KB |
| Gemmini | int8 | 16 x 16 | Systolic | 320KB |
| MAERI | int16 | 1 x 128 | Adder tree | 80KB |
| NVDLA | int8 | 8 x 8 | Adder tree | 65KB |

# Evaluation Results: Comparative Analysis

## Accelerator-Level Energy Breakdown at 45nm:

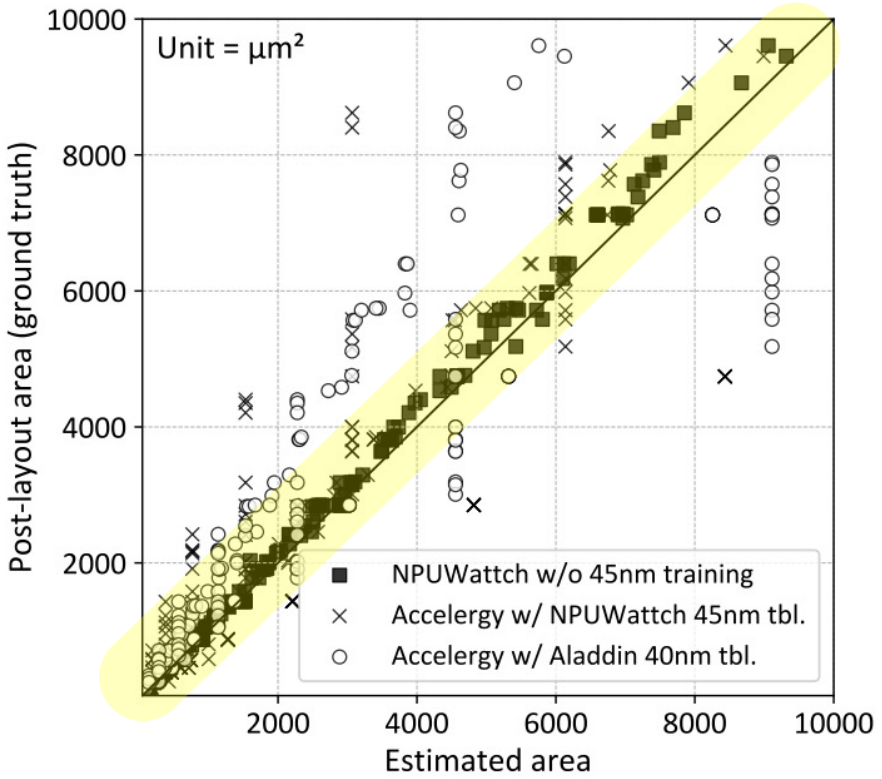
- *2.5x improvement* in mean absolute percentage error (MAPE) over Accelergy



# Evaluation Results: Modeling Accuracy

**Estimated vs. Ground Truth Area for Primitive Components:**

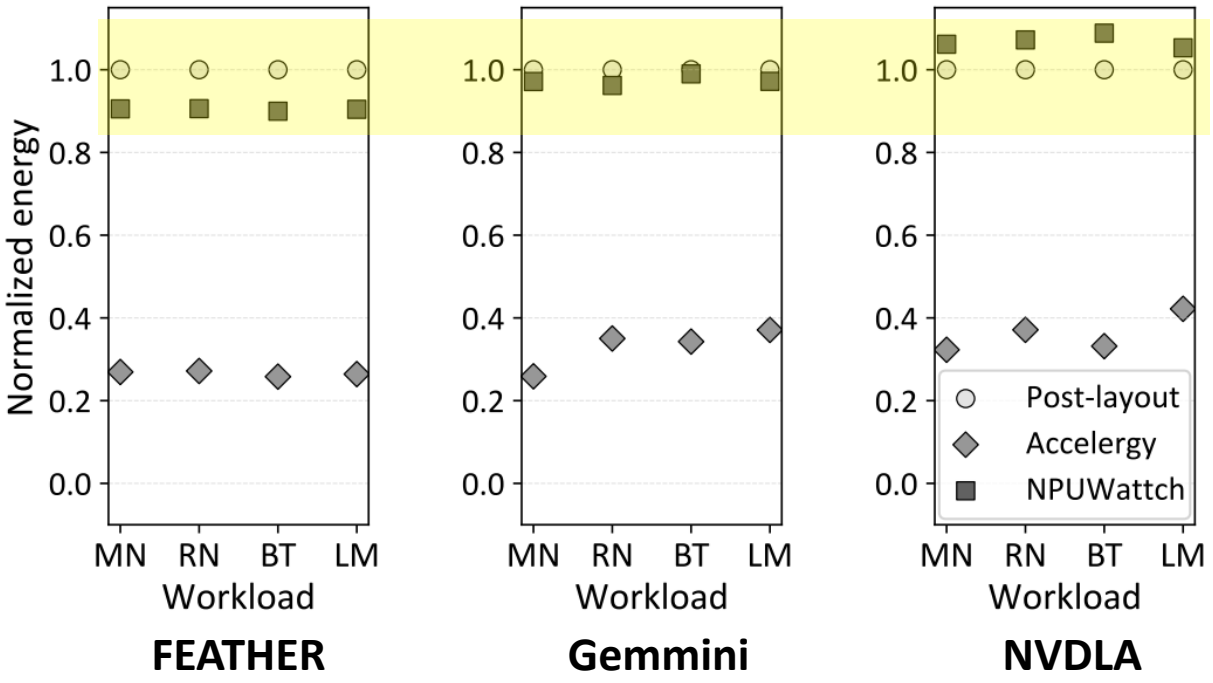
- *2.7% estimation error* on average
- NPUWattch (■) closely matches the post-layout data (diagonal line).



# Evaluation Results: Workload-Level Analysis

**Workload-Level Energy Estimation:**

- *6.3% error* (NPUWattch) vs. *68.1% error* (Accelergy)
- NPUWattch (■) closely follows the post-layout results (●).




| Networks | Abbr. | Features |
|---|---|---|
| MobileNetV3 | MN | Depthwise convolution |
| ResNet-50 | RN | Dense convolution |
| BERT | BT | Encoder-only transformer |
| Llama-3 | LM | Generative AI |

# NPUWattch Tutorial: Basic Usage

**Sehyeon Kim**
School of Electrical and Electronic Engineering
Yonsei University

# Content

- **Setup:** Install NPUWattch and Check the Environment
- **PyTorchSim + TPU-like NPU:** Run a Matmul and Read the Report
- **PyTorchSim Workloads:** Interpret Training, Utilization, and LLM Results

# NPUWattch: What the Tool Does

1. **What it is.** NPUWattch is a fast and accurate PPA estimation framework for NPUs: energy and power, area, and an optional timing check, on top of the cycles your simulator reports.

2. **What it reads.** The output of a simulator run you already have: a PyTorchSim run folder or two native files from any simulator.

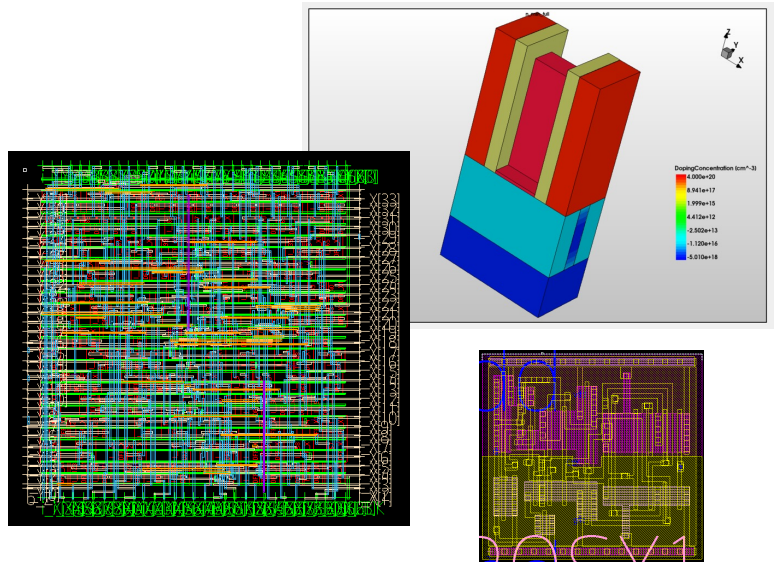
3. **How it prices.** Every component is costed by models trained on post-layout logic data and SRAM characterization, at the technology node you choose.

4. **What you get.** One command, seconds per run, no EDA tools: energy by component and by kernel, area, average power, and every assumption printed as a message.

# NPUWattch: We Handle the Complexity, You Run the Framework

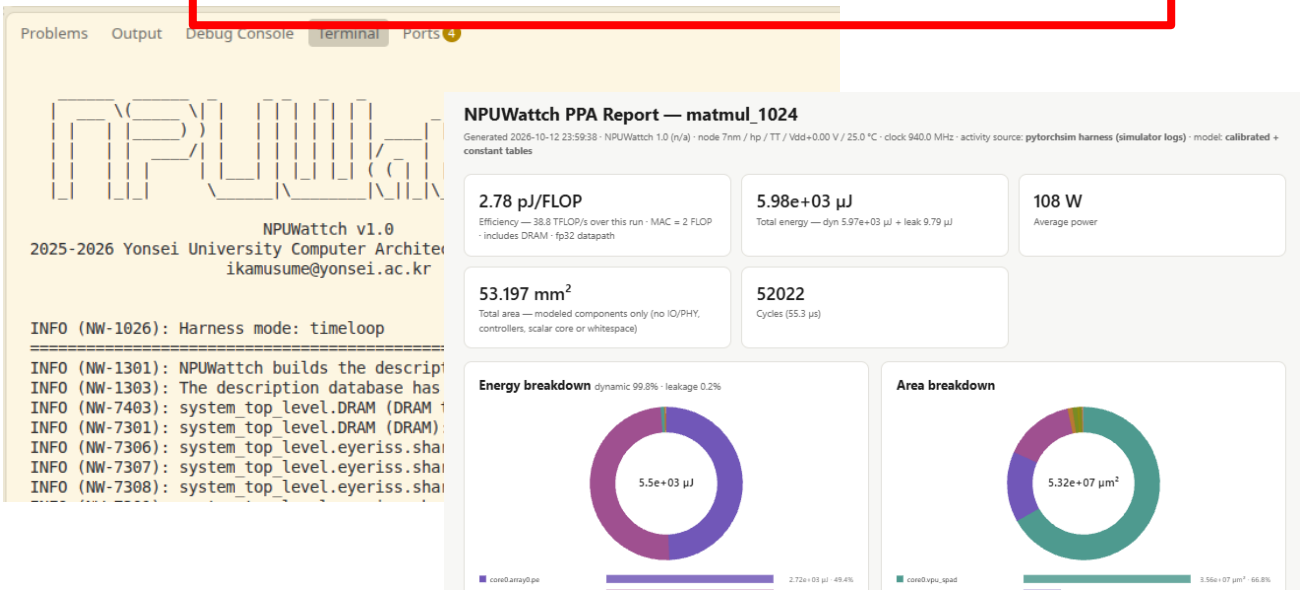
## Our side: Framework Development

Technology libraries (cells, SRAM)
Post-layout design cost dataset
Estimator models and solvers
Harnesses: PyTorchSim, Timeloop



## Your side: Framework Execution

Run one command on simulator output
Read the console and HTML report
Edit a YAML file, run again
No EDA tools, no characterization

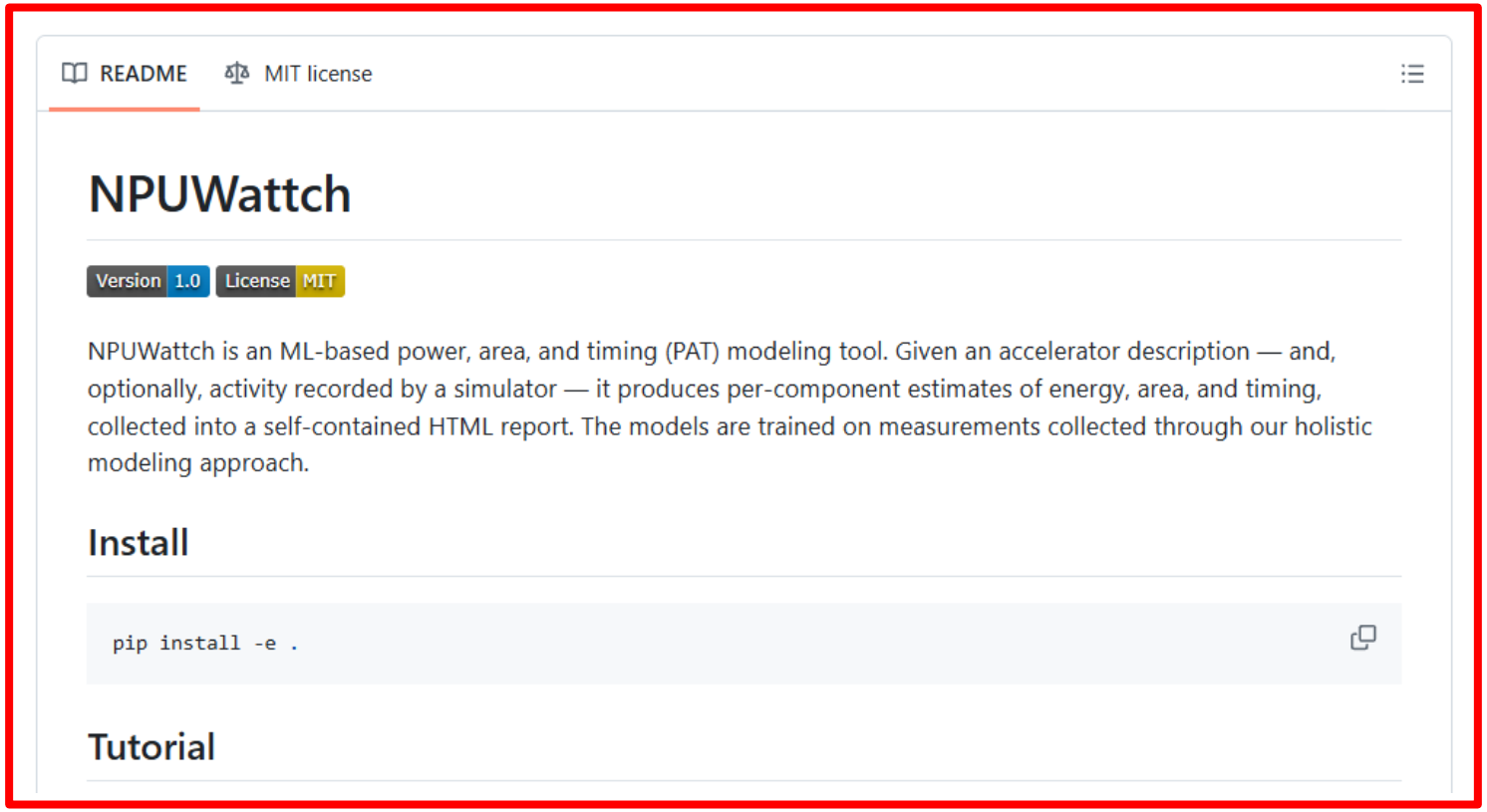



**More estimators, harnesses and artifacts are on the way**

# NPUWattch GitHub Repository

**https://github.com/yonseicasl/NPUWattch**

**Pull requests and feedback are welcome. More tutorials and material are on the way.**


README MIT license

## NPUWattch

Version 1.0 License MIT

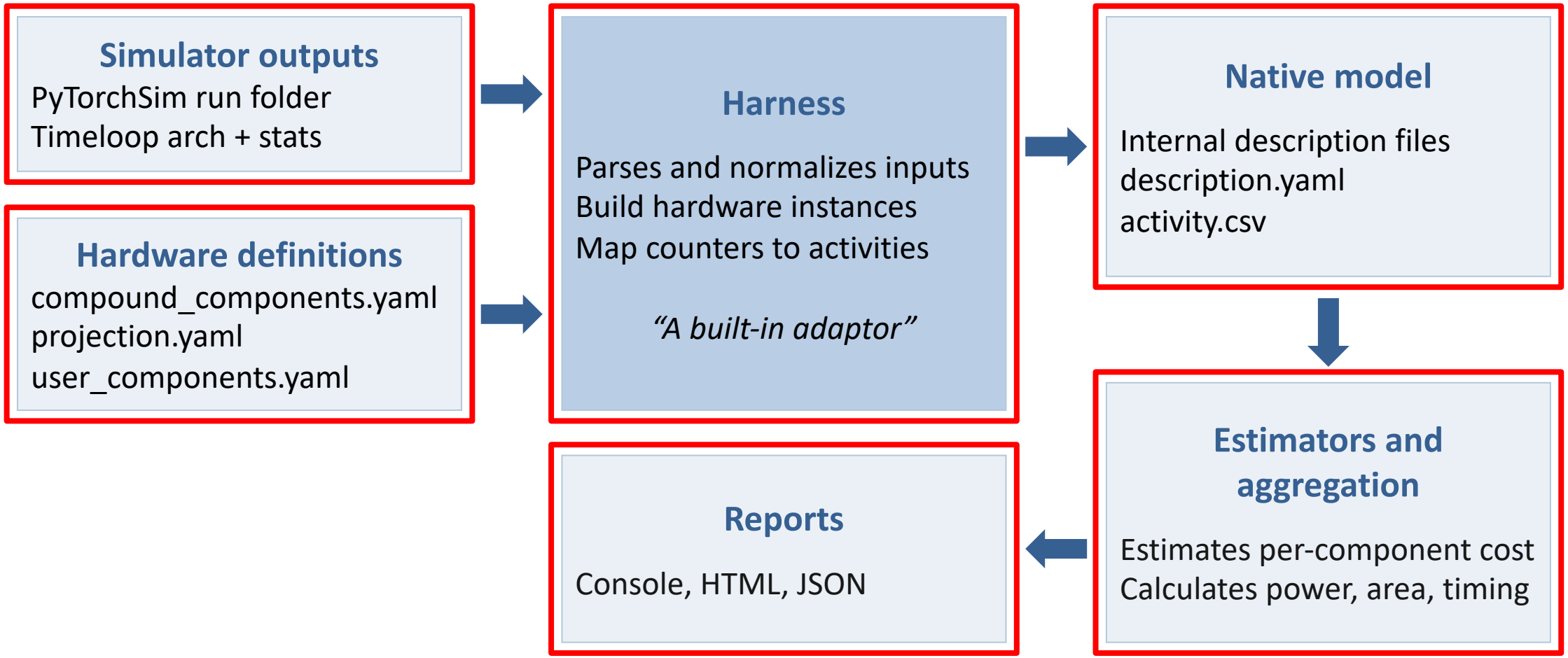
NPUWattch is an ML-based power, area, and timing (PAT) modeling tool. Given an accelerator description — and, optionally, activity recorded by a simulator — it produces per-component estimates of energy, area, and timing, collected into a self-contained HTML report. The models are trained on measurements collected through our holistic modeling approach.

### Install

```
pip install -e .
```

### Tutorial

# NPUWattch: End-to-End Program Structure




**We provide *harnesses* to simplify framework usage**

# Where the Program Lives

| Location | Role |
|---|---|
| src/npuwattch/ | CLI, native model, energy aggregation, reports |
| src/npuwattch_harness/ | PyTorchSim and Timeloop readers; compound expansion |
| src/npuwattch_estimators/ | Trained logic / SRAM models and custom estimators |
| tutorial/hands_on/ | Notebook flow and tutorial_helpers.py |
| tutorial/pytorchsim/ and tutorial/timeloop/ | Hardware definitions, saved simulator runs, reports |

**The files we edit during the hands-on**

compound_components.yaml · projection.yaml · user_components.yaml

# The Native Model: Hardware and Activity

**description.yaml: what the hardware is**

```
- name: core0.array0.pe
  class: fpmac
  count: 16384
  attributes:
    exponent_bits: 8
    mantissa_bits: 23
    pipeline_stages: 4
```

**activity.csv: fields in one recorded row**

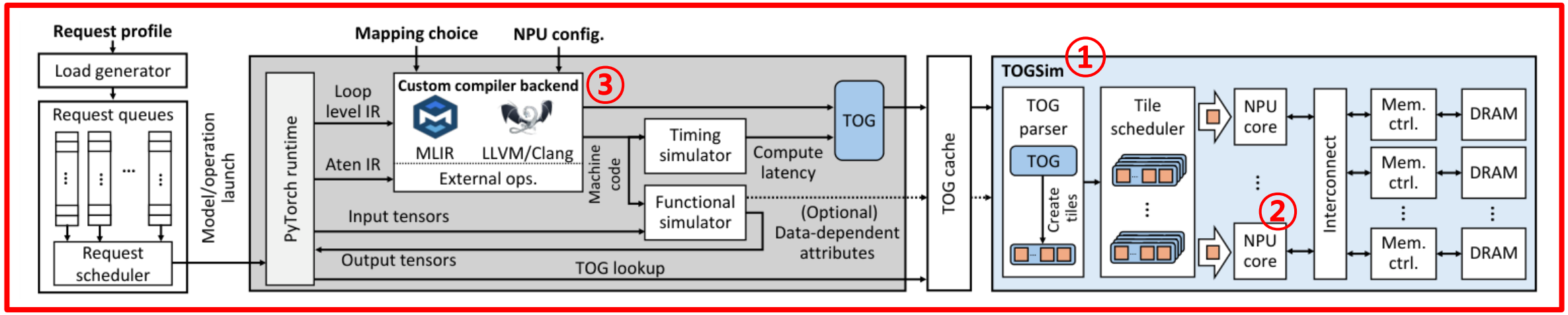
```
component: core0.array0.pe
event: op
mode: hold_b
count: 536870912
window: 0, cycles 0–52021
  # one window = one kernel
```

mode selects the characterized stimulus.
Activity count sums events over all instances.

Dynamic energy = Σ(event count × energy per event)
Leakage energy = instance count × leakage power × time

**Hardware count controls area; activity count controls dynamic energy**

# PyTorchSim: The Execution Flow Behind Our Inputs




| PyTorchSim produces | NPUWattch uses |
|---|---|
| ① TOGSim counters, per kernel | Machine shape, elapsed cycles, memory and NoC activity |
| ② gem5 instruction classes × tile graph | Systolic and SFU events of each kernel |
| ③ Config block + kernel metadata | Hardware parameters, operand formats, useful MACs |

One export file per run carries all three (format agreed with the PyTorchSim authors); NPUWattch reads it, not the raw logs.

Figure: Yang et al., PyTorchSim, MICRO 2025, Fig. 1

# Which PyTorchSim: the v1.1.0 Release, TPUv3-like Configuration

| PyTorchSim v1.1.0 feature | Where it shows up in this hands-on |
|---|---|
| Inference and torch.compile | One matmul, reconstructed machine, energy breakdown |
| Training and autograd | Forward / grad_b / grad_a as three kernels in the report |
| Mapping and TOGSim configuration | Tile occupancy, memory traffic, one vs. two cores |
| Execution modes and log analysis | Which logs provide cycles, counters, and data types |
| Internals and custom instructions | Counter-to-component projection and custom blocks |

Validated against a real TPU v3; every run used here is saved in the repository.

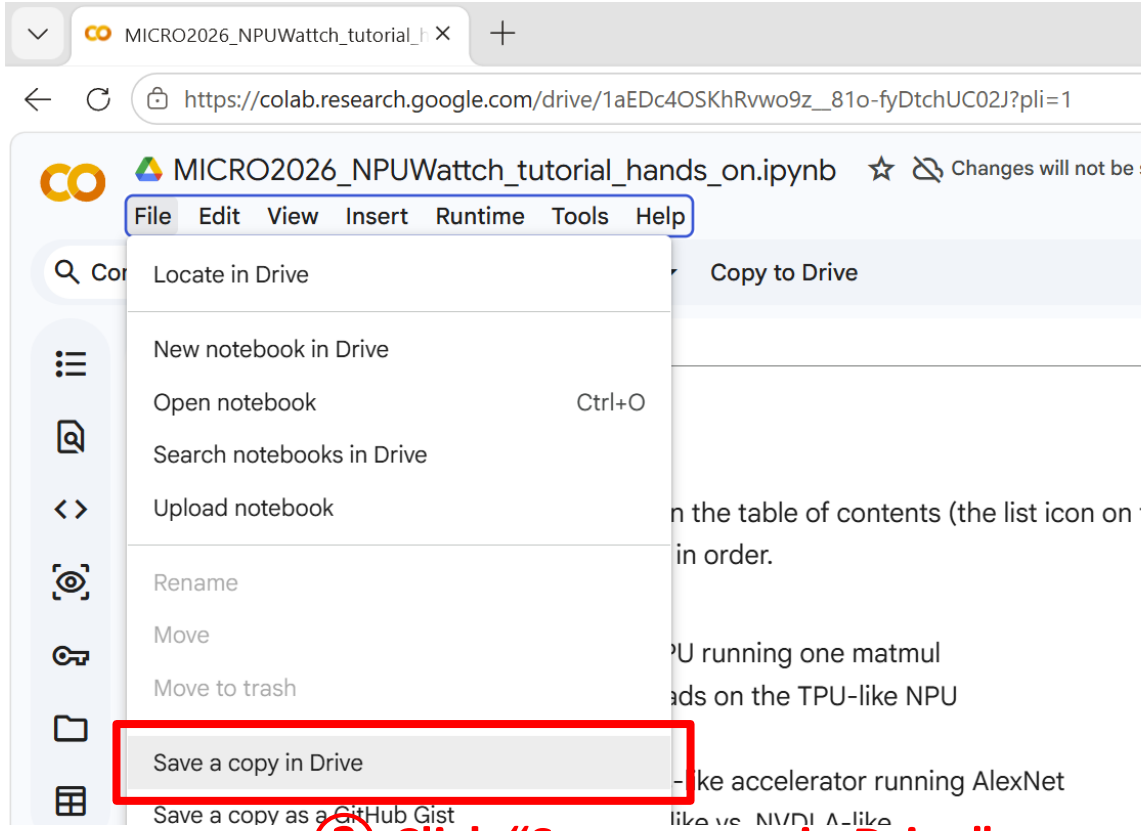
**The same workload and machine context carries into the energy analysis**

# Part 0
# Installing NPUWattch on Colab

# Setup: Open the Notebook in Colab

**https://tinyurl.com/micro2026npuwattch**

① Open this link

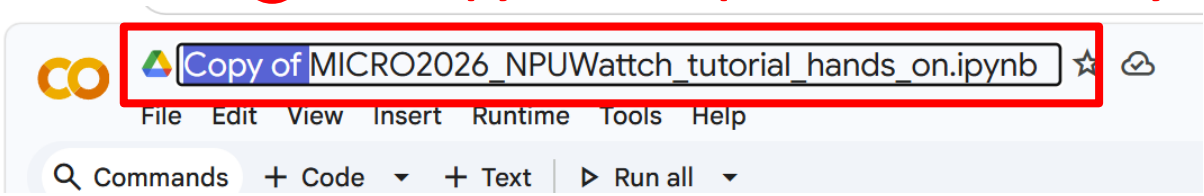



② Click “Save a copy in Drive”

1. Sign in to your Google account.
2. Open the link on the left.
3. In Colab, click File → Save a copy in Drive.
4. Colab opens your own copy in a new tab.

Work in that copy.

③ The copy will be opened automatically




Part 0 · Setup · Everyone works in their own copy of the shared notebook

# Setup: Run the Colab Setup Cell

### In Colab: run the setup cell first (outside Colab it does nothing)

```
import os, sys, subprocess
if "google.colab" in sys.modules:
    repo = "/content/NPUWattch"
    if not os.path.isdir(repo):
        subprocess.run(["git", "clone", "-q", "--depth", "1", "--branch", "main",
                        "https://github.com/yonseicasl/NPUWattch", repo], check=True)
        subprocess.run([sys.executable, "-m", "pip", "install", "-q", "-e", repo], check=True)
    os.chdir(repo + "/tutorial/hands_on")
```

No laptop? Every example folder keeps the report of a 7 nm run in out/, and every notebook keeps its output.

# Check the Environment

## Cell input

```
import sys; sys.path.insert(0, ".")
from tutorial_helpers import *
print("npuwattch:", npuwattch_binary())
print(subprocess.run([npuwattch_binary(), "--version"], capture_output=True, text=True).stdout ...)
print("tutorial root:", TUTORIAL_ROOT)
for p in ["timeloop/eyeriss_like/alexnet", ..., "pytorchsim/tpu_like_bf16/matmul_1024"]:
    d = example(p)
    print("ok " if (d / "run").exists() else "?? ", p,
          "report already present:", (d / "out/report.html").exists())
```

prints the path

version, root, 7 folders

## Cell output

```
npuwattch: .../envs/npuwattch/bin/npuwattch
npuwattch 1.0
tutorial root: .../NPUWattch/tutorial
  ok  timeloop/eyeriss_like/alexnet         report already present: True
  ok  timeloop/gemmini_like/resnet50        report already present: True
  ok  timeloop/gemmini_like/llama3_8b       report already present: True
  ok  timeloop/nvdla_like/resnet50          report already present: True
  ok  timeloop/nvdla_like/llama3_8b         report already present: True
  ok  pytorchsim/tpu_like_fp32/matmul_1024  report already present: True
  ok  pytorchsim/tpu_like_bf16/matmul_1024  report already present: True
```

the path

ok × 7

Seven example folders, each with its saved 7 nm report

# One Notebook Pattern for Every Example

```
ex = example("pytorchsim/tpu_like_fp32/matmul_1024")
text = run(ex)
show_tree(text)
show_messages(text)
rep = report(ex)
display(components_df(rep))
show_report(ex)
```

one run = one call

one piece at a time

| Object | What it contains |
|---|---|
| text | Console output |
| rep | report.json data |
| show_report(ex) | Embedded HTML report |

tables from report.json

Every example folder also keeps its run script and out/ report files.

Read every run in the same order:

1. the machine (tree and Machine: line)
2. the messages
3. the numbers, next to time and MAC util %

machine first

messages second

numbers third

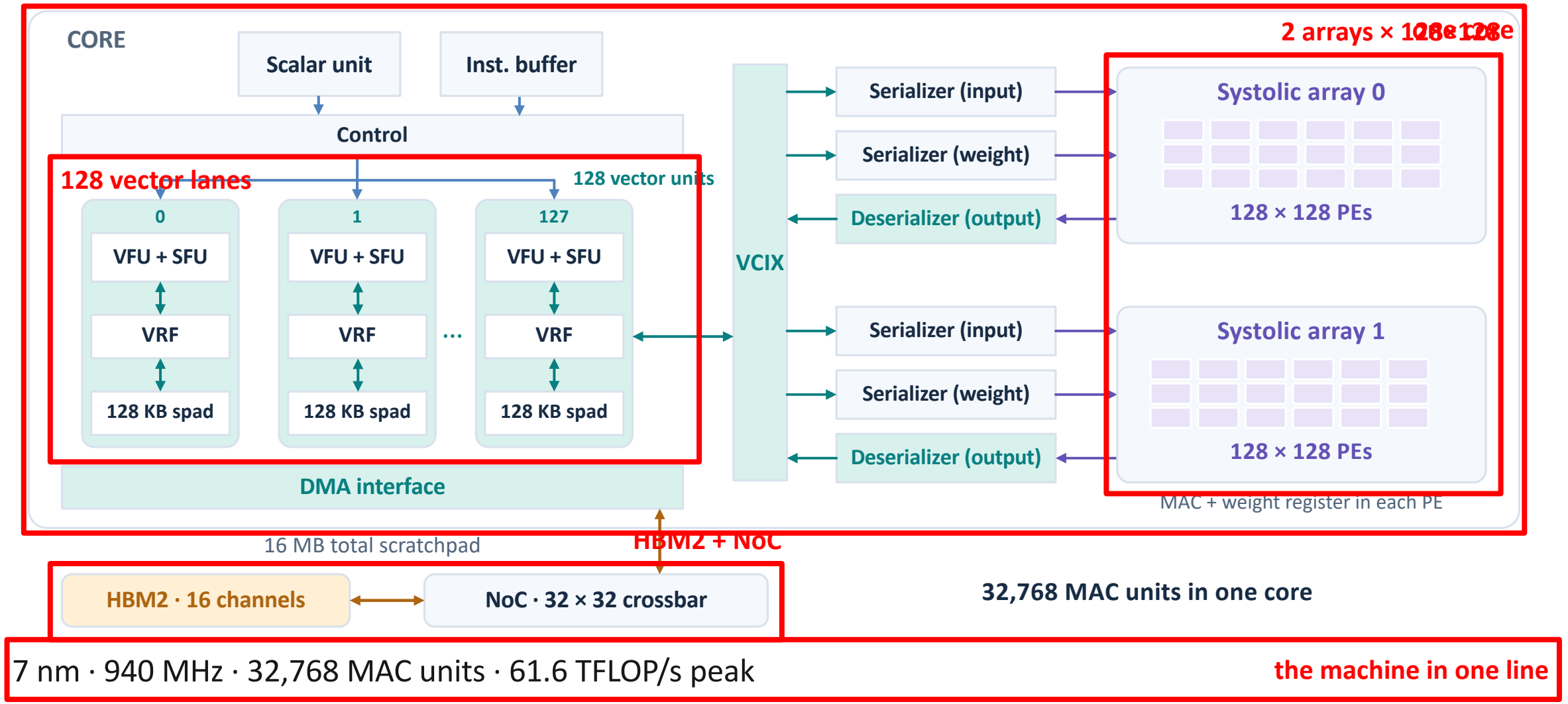
# Part 1
# PyTorchSim + TPU-like NPU: One Matmul, End to End

# Which TPU, and Why

PyTorchSim models several TPU generations: v3 and v6e.
This hands-on uses the v1.1.0 release with its TPU v3-like configuration: 1 core, 2 × 128×128 MXUs, 940 MHz, 16 HBM2 channels.

The richest public efficiency data is for TPUv4i, and TPU v3 is its closest relative.
Part 1 ends by putting our number next to the TPUv4i.

# PyTorchSim + TPU-v3-like NPU: The Baseline Machine




Part 1 · Hardware · Adapted from PyTorchSim, Figure 2; parameters from the tutorial

# First Run: Reconstructing the Machine

```
ex32 = example("pytorchsim/tpu_like_fp32/matmul_1024")
text32 = run(ex32)              # runs the folder's ./run script, returns the console text
show_tree(text32)
machine_card(report(ex32))
```

the run command

```
INFO (NW-1001): Instance hierarchy (reconstructed from the run's model):
chip
 ├── core0
 │    ├── array0
 │    │    ├── pe [×16384]  (class: fpmac, exponent_bits=8, mantissa_bits=23, pipeline_stages=4)
 │    │    └── w_reg [×16384]  (class: register_file, data_width=32, mem_depth_per_bank=1 = 4 B)
 │    ├── array1
 │    │    ├── pe [×16384]  (class: fpmac, exponent_bits=8, mantissa_bits=23, pipeline_stages=4)
 │    │    └── w_reg [×16384]  (class: register_file, data_width=32, mem_depth_per_bank=1 = 4 B)
 │    ├── vpu_spad [×128]  (class: sram, mem_template=sram_256k, mem_banks=1, mem_depth_per_bank=8192, data_width=128 = 128 KB)
 │    ├── vrf [×128]  (class: register_file, data_width=256, mem_depth_per_bank=32 = 1 KB)
 │    ├── vfu [×128]  (class: fpmac, exponent_bits=8, mantissa_bits=23, pipeline_stages=4)
 │    ├── dma_q  (class: fifo)
 │    ├── dma_addr  (class: intadd)
 │    └── sfu_pipe [×128]  (class: fpsfu)
 ├── noc
 │    ├── icnt_xbar  (class: crossbar, net_inputs=32, net_outputs=32, data_width=256)
 │    └── icnt_buf [×32]  (class: sram, mem_template=sram_64k, mem_banks=1, mem_depth_per_bank=1024, data_width=64 = 8 KB)
 └── dram
      └── dram_chan [×16]  (class: DRAM)
Machine: 7 nm, 940 MHz. 1 core, 2 arrays of 128×128 fp32 PEs = 32,768 MAC units (peak 61.6 TFLOP/s).
      128 lanes × 128 KB scratchpad. NoC 32×32 crossbar. DRAM 16 channels at 3.48 pJ/bit.
```

tree: two arrays

128 vector lanes

NoC + 16 HBM channels

clock + peak rate

Part 1 §1 · Run it, and read the rebuilt machine · Saved notebook output

# The Messages: Four WARNING Lines

```
show_messages(text32, kinds=("WARNING",))
```

```
WARNING (NW-8212): vpu_spad: The SRAM of 1048576 bits uses the macro templates 4x sram_256k in banks of at most
        16 macros (utilization 100.0%).
WARNING (NW-8212): icnt_buf: The SRAM of 65536 bits uses the macro templates 1x sram_64k in banks of at most
        16 macros (utilization 100.0%).
WARNING (NW-6607): CustomMatMulwVpush: NPUWattch divides the kernel total between the arrays by their active cycles
WARNING (NW-8111): core0.vrf: width=256 is outside the characterized range 8-128 of regfile.
```

NW-8212: built from SRAM macros

NW-6607: split by active cycles

the same, in plain words

| Code | What NPUWattch filled in |
|---|---|
| NW-8212 (×2) | vpu_spad and icnt_buf were given as a capacity; NPUWattch built them from characterized SRAM macros |
| NW-6607 | The weight-push count is a kernel total; it is split between the two arrays by their active cycles |
| NW-8111 | The vector register file is 256 bits wide, beyond the 8 - 128-bit characterized range: extrapolated |

**These are not errors: NPUWattch filled a gap and tells you how, so you can check it**

Part 1 §2 · Messages · Saved notebook output

# INFO Lines, One Explanation, and the Summary

```
show_messages(text32, kinds=("INFO",))       # 23 INFO lines: scope, waived counters, clock gating
show_messages(text32, codes=["NW-8111"])
explain("NW-8111")                           # = npuwattch --explain NW-8111
show_summary(text32)
```

```
INFO (NW-6008): Out of scope: scalar core (with its L1 I-cache / D-cache, branch predictor and TLBs) is excluded
        from the energy model; gem5 stats are read only for numCycles and the instruction counts ...
INFO (NW-6008): Out of scope: DRAM background/standby power and the NPU-side memory controller / HBM PHY are not charged
INFO (NW-2209): Activity stat 'CustomMatMul' (total 512 in 1 window(s)) is not charged, because projection 'pytorchsim'
        waives it: compute is charged as useful MACs (useful_macs action) plus idle PE cycles
INFO (NW-3107): Clock gating is assumed: idle cycles of 8 logic component(s) cost leakage only.
```

out of scope

not charged

clock gating assumed

```
NW-8111  WARNING
 Message:   {param}={value:g} is outside the characterized range {lo:g}-{hi:g} of {primitive}.
 The model of this primitive has training data only in this range. NPUWattch extrapolates the value.
 The result of this component is less accurate.
```

--explain NW-8111

```
INFO (NW-1902): Message summary: 0 CRITICAL, 0 ERROR, 4 WARNING, 23 INFO.
   NW-6607   WARNING  x1
   NW-8111   WARNING  x1
   NW-8212   WARNING  x2
```

message summary

INFO is a fact, WARNING is approximate, ERROR is your input, CRITICAL is our bug

four levels

Part 1 §2 · Message codes · Any code: npuwattch --explain NW-nnnn; full catalog in MESSAGES.md

# Matmul $1024^3$: The Reference Result

```
rep32 = report(ex32)
display(components_df(rep32, top=12))
show_totals(text32)
print(rep32["efficiency"]["pJ_per_flop_str"])
show_utilization(rep32)
```

per component

arrays

DRAM

the rest

| component | model | energy | share % |
|---|---|---|---|
| core0.array0.pe | cal | 2.72e+03 µJ | 45.4 |
| core0.array1.pe | cal | 2.72e+03 µJ | 45.4 |
| dram_chan | const | 485 µJ | 8.1 |
| core0.vpu_spad | cal | 41.8 µJ | 0.7 |
| icnt_buf | cal | 17.3 µJ | 0.3 |
| icnt_xbar, dma_q, vfu, vrf, … | cal | < 3 µJ each | 0.0 |

```
total energy = 5.981e+09 pJ (dyn 5.972e+09 + leak 9.79e+06); avg power = 1.081e+05 mW; exec = 5.534e-05 s
cycles: 52,022 at 940 MHz = 55.3 µs
useful MACs: 1.07 G on 32,768 MAC units in 52,022 cycles = 63.0% MAC utilization
```

the three lines

| Total energy | pJ / FLOP | Runtime | Average power | MAC utilization |
|---|---|---|---|---|
| 5.98 mJ | 2.78 pJ/FLOP | 55.3 µs | 108 W | 63.0% |

totals

runtime, power

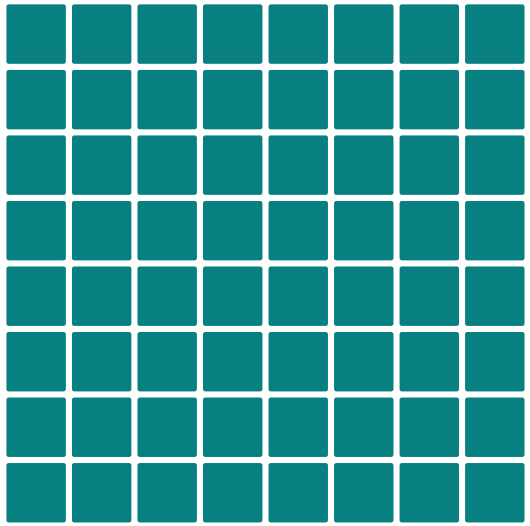
High weight reuse makes the arrays dominate this matmul

Part 1 §3 · Where the energy went · 7 nm, 940 MHz, fp32 PEs · pJ / FLOP includes DRAM · Saved notebook output

# Occupancy and MAC Utilization Measure Different Things

space: one tile

**SPACE: an active weight tile**

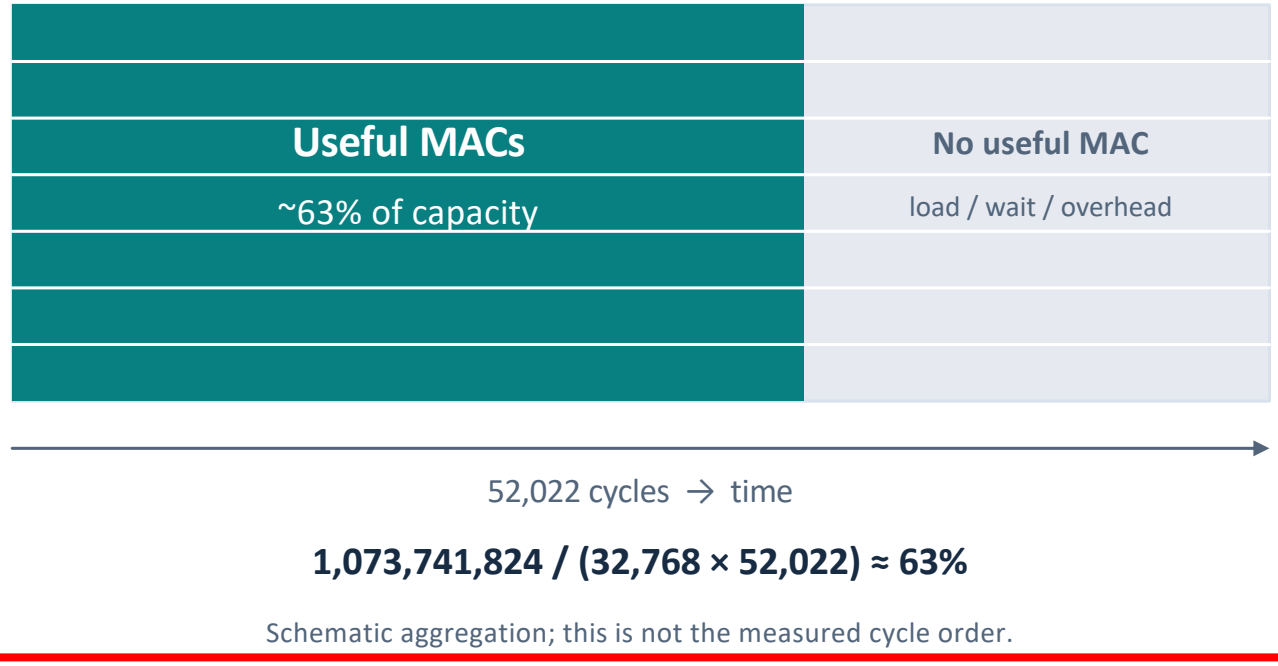


**100% occupied**

All 128 × 128 weights are real.

space × time: the run

**SPACE × TIME: the complete run**




**1,073,741,824 / (32,768 × 52,022) ≈ 63%**

Schematic aggregation; this is not the measured cycle order.

INFO (NW-6222): 1073741824 useful MACs of 1073741824 active PE cycles (100.0%) → occupancy 100% **occupancy line**

1.07 G useful MACs / (32,768 MAC units × 52,022 cycles) = 63% → MAC utilization **utilization line**

Part 1 · Reading utilization · Diagram adapted from the supplied SVG

# The Same Matmul in bf16: Set Up Like the TPU

## compound_components.yaml · systolic_mac (condensed)

```
systolic_mac:
 default_mode: gated     # unnamed elements: gated
 elements:
  pe:                    # bf16 x bf16 multiplier
   primitive: fpmul
   config: {exponent_bits: 8, mantissa_bits: 7, pipeline_stages: 2}
  acc:                   # fp32 adder: product + partial sum
   primitive: fpadd
   config: {exponent_bits: 8, mantissa_bits: 23, pipeline_stages: 2}
  acc_reg:               # 32-bit partial-sum register
   primitive: regfile
   config: {data_width: 32, mem_depth_per_bank: 1}
```

default: gated

the TPU's PE

## Matching the TPU datapath

- A TPU MXU multiplies in bf16 and accumulates in fp32. The PE is written to that spec: a bf16 multiplier, an fp32 adder, and a 32-bit partial-sum register; the weight register is 16 bits.
- One useful MAC = one multiply + one add + one register write. The report counts the multiplier events as the MACs.
- PyTorchSim runs fp32 by default. For the 7 nm TPU comparison we ran this matmul in bf16, so the cycles and the DRAM bytes are bf16 too.

the setup

```
ex16 = example("pytorchsim/tpu_like_bf16/matmul_1024")
text16 = run(ex16)                          # the bf16 run
show_totals(text16); show_utilization(report(ex16))
```

the run

```
total energy = 3.019e+09 pJ (dyn 3.013e+09 + leak 6.667e+06); avg power = 6.28e+04 mW; exec = 4.808e-05 s
cycles: 45,195 at 940 MHz = 48.1 µs
useful MACs: 1.07 G on 32,768 MAC units in 45,195 cycles = 72.5% MAC utilization
```

3.0 mJ, 45,195 cycles
72% utilization

Same machine, bf16 end to end like the TPU: 45,195 cycles and half the DRAM bytes

Part 1 §4 · The same matmul in bf16 · 7 nm, 940 MHz · PyTorchSim run in bf16 for the TPU comparison · Saved notebook output

# bf16 vs fp32, and the Published TPUv4i Figure

## The two designs on the 1024³ matmul, 7 nm

Efficiency with DRAM / chip only

| design | cycles | time | total energy | avg power | DRAM | MAC util | pJ/FLOP, with DRAM | pJ/FLOP, chip only |
|---|---|---|---|---|---|---|---|---|
| fp32 PE | 52,022 | 55.3 µs | 5.98 mJ | 108 W | 485 µJ | 63.0% | **2.78** | **2.56** |
| bf16 PE | 45,195 | 48.1 µs | 3.02 mJ | 62.8 W | 236 µJ | 72.5% | **1.41** | **1.30** |

fp32 / bf16

## TPUv4i, and this design at the same node and precision

TPUv4i chip only efficiency

| chip | node | MXUs | pJ/FLOP, chip only |
|---|---|---|---|
| TPUv4i | 7 nm | 4 × 128×128, 1 core, 1050 MHz | **1.27** |

TPUv4i: 7 nm, bf16 MXUs. The pJ/FLOP is derived from the chip figures published in Jouppi et al., ISCA 2021, Table 1.
This design: 7 nm, bf16 PEs, one 1024³ matmul.

same setup

within 10%

**The model lands within 10% of the published figure**

Part 1 §4 · The same matmul in bf16 · Both designs at 7 nm, 940 MHz · TPUv4i figure derived from Jouppi et al., ISCA 2021, Table 1

# Projection: From a Simulator Counter to a PE Action

```
src = (example("pytorchsim/tpu_like_fp32") / "projection.yaml").read_text().splitlines()
start = next(i for i, l in enumerate(src) if l.strip().startswith("systolic_mac:"))
print("\n".join(src[start:start + 19]))
```

```
  systolic_mac:
    CustomMatMul:                # weight-stationary compute: the useful MACs
      count_from:
        stat: useful_macs        # M·K·N of the operator (log operation + meta.txt shapes)
      elements:
        pe: hold_b               # PE computes with b latched
        w_reg: idle              # the weight stays and is clock-gated: leakage only
    CustomMatMulIdle:            # active PE cycles without a real weight
      count_from:
        stat: array_idle_slots   # systolic_active_cycles × lanes² − useful_macs
      elements:
        pe: idle                 # the PE is clock-gated: leakage only
        w_reg: idle
```

useful MACs → PE op

empty cycles → idle

useful_macs selects PE work; hold_b describes weight-stationary activity

the rule in one line

Part 1 §5 · Read projection.yaml · Saved notebook output (first 13 of 19 lines)

# The Native Files: What the Harness Built

```
_ = run(ex32, "-o", "native/", log_name="console_native.txt")    # -o DIR writes the two native files
show_file(ex32 / "native" / "description.yaml", max_lines=30)
show_file(ex32 / "native" / "activity.csv")
```

**-o writes the two files**

## native/description.yaml (first lines)

```
npuwattch:
 version: '1.0'
 technology:
  node: 7nm
  ...
 clock:
  frequency_MHz: 940.0
 components:
 - name: core0.array0.pe
  class: fpmac
  count: 16384
  attributes:
   data_width: 32
   exponent_bits: 8
   mantissa_bits: 23
   pipeline_stages: 4
 - name: core0.array0.w_reg
  class: register_file
  count: 16384
  ...
```

**description.yaml**

## native/activity.csv (23 rows)

```
window,cycle_start,cycle_end,component,event,mode,count
0,0,52021,core0.array0.pe,op,hold_b,536870912
0,0,52021,core0.array1.pe,op,hold_b,536870912
0,0,52021,core0.array0.w_reg,idle,idle,536870912
0,0,52021,core0.array0.pe,idle,idle,524288
0,0,52021,core0.array0.w_reg,write,write,524288
0,0,52021,core0.vpu_spad,write,write,2621440
0,0,52021,core0.vpu_spad,read,read,2359296
0,0,52021,core0.vfu,op,random,139648
0,0,52021,core0.vrf,read,read,279296
0,0,52021,core0.dma_q,stream,stream,393216
0,0,52021,icnt_xbar,transfer,valid25,98304
0,0,52021,icnt_buf,write,write,3145728
0,0,52021,dram_chan,activate,activate,118959
0,0,52021,dram_chan,read,read,262144
0,0,52021,dram_chan,write,write,131072
0,0,52021,dram_chan,refresh,refresh,448
...
,,,__meta__,total_cycles,,52022
```

**hold_b**

**activity.csv**

Part 1 §6 · The native files · Excerpts of the saved notebook output (array1 rows omitted)

# Replay the Native Files Without the Harness

## Cell input

```
text_native = run_npuwattch(["-d", "native/description.yaml", "-l", "native/activity.csv"], cwd=ex32)
show_totals(text_native)
print("same total as the harness run:",
    abs(total_energy_pJ(text_native) - total_energy_pJ(text32)) / total_energy_pJ(text32) < 1e-6)
```

**-d and -l, no harness**

## Cell output

```
total energy = 5.981e+09 pJ (dyn 5.972e+09 + leak 9.79e+06); avg power = 1.081e+05 mW; exec = 5.534e-05 s
calibrated primitives available: fpadd, fpmul, fpmac, intadd, intmul, intmac, fpsfu, mxfpmac, fifo, regfile, ...
same total as the harness run: True
```

**same total**

## The same run in a terminal

```
npuwattch -d native/description.yaml -l native/activity.csv
```

**the terminal form**

**Same total to the last digit: the model only ever sees these two files**

Part 1 §6 · The native format is also the integration path for another simulator

# The Report

```
show_report(ex32)
# Also open: out/report.html
```

show_report

## On-chip component energy

charts




| Report section | What to check |
|---|---|
| Totals and kernels | Energy, time, per-kernel variation |
| Breakdowns | On-chip, DRAM, area, leakage |
| Tree and provenance | Inputs, assumptions, warnings |

Chart excerpt from the saved HTML report.
Shares here use on-chip energy as the denominator.

Part 1 §7 · report.json provides the same numbers for scripts

# Part 2
# PyTorchSim Workloads: Training, Utilization, and LLM Results

# PyTorchSim Workloads: Questions for Each Run

| Workload | What changes | What to inspect |
|---|---|---|
| Training | Forward + backward kernels | Per-kernel energy |
| $2048^3$, one vs. two cores | Machine capacity and elapsed time | Energy, power, area |
| Softmax + exp2 | Vector and special-function work | VFU/SFU and DRAM activity |
| ResNet-50 subset | Convolution shapes and padding | Occupancy and bytes per useful MAC |
| LLM decoder blocks | Prefill/decode, dense/MoE | Weight reuse and memory traffic |

Keep the $1024^3$ matmul as the reference: 5.98 mJ, 55.3 µs, 63% MAC utilization.

Part 2 · All results on the recorded TPU-like runs; the 2-core experiment has a larger machine

# The Twelve PyTorchSim Workloads

```
base = "pytorchsim/tpu_like_fp32/"
WORKLOADS = {
    "matmul 1024³ (nb 1)": base + "matmul_1024",
    "matmul train": base + "matmul_train",
    "matmul 2048³ 1 core": base + "matmul_2048_1core",
    "matmul 2048³ 2 cores": base + "matmul_2048_2core",
    "softmax + exp2": base + "softmax",
    "ResNet-50": base + "resnet50",
    "Llama-3-8B prefill": base + "llama3_8b/prefill",
    "Llama-3-8B decode": base + "llama3_8b/decode",
    "Gemma-2-9B prefill": base + "gemma2_9b/prefill",
    "Gemma-2-9B decode": base + "gemma2_9b/decode",
    "DeepSeek-V2-Lite prefill": base + "deepseek_v2_lite/prefill",
    "DeepSeek-V2-Lite decode": base + "deepseek_v2_lite/decode",
}
```

Keep the run labels consistent: later cells use them as dictionary keys.

Part 2 §1 · Run them all

# Run the Twelve Workloads and Keep Their Reports

run all twelve

```
texts, reps = {}, {}
for label, path in WORKLOADS.items():
    if not (example(path) / "run").exists():
        print(f"{label:28s} (folder not present, skipped)"); continue
    texts[label] = run(example(path))                # console text of each run
    reps[label] = report(example(path))               # its report.json
    print(f"{label:28s} {reps[label]['totals']['energy_str']:>12s}   {len(reps[label]['windows']):2d} window(s)")
machine_card(reps["matmul 1024³ (nb 1)"])
```

one line per run

```
matmul 1024³ (nb 1)          5.98e+03 µJ    1 window(s)
matmul train                 1.8e+04 µJ    3 window(s)
matmul 2048³ 1 core          4.79e+04 µJ    1 window(s)
matmul 2048³ 2 cores         4.76e+04 µJ    1 window(s)
softmax + exp2               1.16e+03 µJ    4 window(s)
ResNet-50                    6.1e+03 µJ    19 window(s)
Llama-3-8B prefill           6.66e+05 µJ    13 window(s)
Llama-3-8B decode            4.29e+04 µJ    13 window(s)
Gemma-2-9B prefill           6.72e+05 µJ    13 window(s)
Gemma-2-9B decode            3.85e+04 µJ    13 window(s)
DeepSeek-V2-Lite prefill     3.37e+05 µJ    17 window(s)
DeepSeek-V2-Lite decode      1.43e+04 µJ    17 window(s)
Machine: 7 nm, 940 MHz. 1 core, 2 arrays of 128×128 fp32 PEs = 32,768 MAC units (peak 61.6 TFLOP/s). ...
```

Part 2 §1 · Run them all · Saved notebook output

# Twelve Runs in One Table: Time and MAC Utilization Next to Every Number

```
def row(label, rep):            # summary_row() + cycles and pJ/FLOP; "—" when the run has no GEMM kernel
    ...
display(pd.DataFrame([row(k, v) for k, v in reps.items()]).drop(columns=["total_pJ"]))
```

| design / workload | total energy | DRAM % | time | avg power | MAC util % | pJ / MAC | pJ / FLOP |
|---|---|---|---|---|---|---|---|
| matmul 1024³ (nb 1) | 5.98e+03 µJ | 8.1 | 55.3 µs | 108 W | 63.0 | 5.57 | 2.78 pJ/FLOP |
| matmul train | 1.8e+04 µJ | 8.6 | 167 µs | 108 W | 62.6 | 5.6 | 2.8 pJ/FLOP |
| matmul 2048³ 1 core | 4.79e+04 µJ | 8.2 | 334 µs | 143 W | 83.6 | 5.57 | 2.79 pJ/FLOP |
| matmul 2048³ 2 cores | 4.76e+04 µJ | 7.7 | 181 µs | 264 W | 77.2 | 5.55 | 2.77 pJ/FLOP |
| softmax + exp2 | 1.16e+03 µJ | 83.1 | 115 µs | 10 W | 0.3 | — | — |
| ResNet-50 | 6.1e+03 µJ | 46.0 | 565 µs | 10.8 W | 3.1 | 11.46 | 5.73 pJ/FLOP |
| Llama-3-8B prefill | 6.66e+05 µJ | 12.1 | 7.54 ms | 88.3 W | 49.0 | 5.85 | 2.92 pJ/FLOP |
| Llama-3-8B decode | 4.29e+04 µJ | 90.7 | 4.01 ms | 10.7 W | 0.2 | 192.45 | 96.2 pJ/FLOP |
| Gemma-2-9B prefill | 6.72e+05 µJ | 20.1 | 11 ms | 60.9 W | 30.5 | 6.48 | 3.24 pJ/FLOP |
| Gemma-2-9B decode | 3.85e+04 µJ | 90.8 | 3.07 ms | 12.5 W | 0.2 | 189.65 | 94.8 pJ/FLOP |
| DeepSeek-V2-Lite prefill | 3.37e+05 µJ | 31.4 | 9.7 ms | 34.8 W | 14.7 | 7.68 | 3.84 pJ/FLOP |
| DeepSeek-V2-Lite decode | 1.43e+04 µJ | 89.6 | 1.17 ms | 12.3 W | 0.2 | 165.56 | 82.8 pJ/FLOP |

**A low MAC utilization is a property of the run; read pJ per MAC with it**

Part 2 §1 · Saved notebook output · area is 53.197 mm² for every run except the 2-core run (106.437 mm²)

# Training: One Forward and Two Backward Matmuls

```
ex_train = example(
    WORKLOADS["matmul train"])
show_file(ex_train / "kernels.yaml")
display(windows_df(
    reps["matmul train"]))
```

Forward: $C = AB$
Input gradient: $dA = dC\ B^T$
Weight gradient: $dB = A^T\ dC$

one forward, two gradients

| Kernel | Cycles | Total energy |
|---|---|---|
| fwd | 52,058 | 5.98 mJ |
| grad_b | 52,848 | 6.03 mJ |
| grad_a | 52,061 | 6.03 mJ |

Similar shapes give three comparable kernels, about 18.0 mJ total

# One Core vs. Two Cores: Faster at Similar Energy

```
one, two = reps["matmul 2048³ 1 core"], reps["matmul 2048³ 2 cores"]
machine_card(one, "1 core"); machine_card(two, "2 cores")
pd.DataFrame([core_row("1 core", one), core_row("2 cores", two)])   # arrays, channels, ports, time, energy
```

Machine (1 core): 7 nm, 940 MHz. 1 core, 2 arrays of 128×128 fp32 PEs = 32,768 MAC units (peak 61.6 TFLOP/s).
128 lanes × 128 KB scratchpad. NoC 32×32 crossbar. DRAM 16 channels at 3.48 pJ/bit.

Machine (2 cores): 7 nm, 940 MHz. 2 cores, 4 arrays of 128×128 fp32 PEs = 65,536 MAC units (peak 123 TFLOP/s).
256 lanes × 128 KB scratchpad. NoC 64×64 crossbar. DRAM 32 channels at 3.48 pJ/bit.

| | 1 core | 2 cores | |
|---|---|---|---|
| Arrays / HBM channels / NoC ports | 2 / 16 / 32 | 4 / 32 / 64 | |
| Cycles → time | 313,749 → 334 µs | 169,707 → 181 µs | 0.54× the time |
| MAC util % | 83.6 | 77.2 | |
| Total energy (leakage) | 4.79e+04 µJ (59 µJ) | 4.76e+04 µJ (63.9 µJ) | leakage energy |
| DRAM % · area mm² · avg power | 8.2 · 53.2 · 143 W | 7.7 · 106.4 · 264 W | twice the power |

The useful MAC count stays fixed while the larger machine finishes sooner

Part 2 §3 · One core or two · Saved notebook output

# Softmax + exp2: Energy Beyond the Systolic Arrays

```
display(windows_df(reps["softmax + exp2"]))
display(components_df(reps["softmax + exp2"], top=10))
show_messages(texts["softmax + exp2"], codes=["NW-6201", "NW-6208", "NW-6002"])
```

| kernel | cycles | total |
|---|---|---|
| 01_softmax_p0 | 14102 | 188 µJ |
| 01_softmax_p1 | 15224 | 202 µJ |
| 01_softmax_p2 | 26837 | 378 µJ |
| 02_exp2 | 52301 | 389 µJ |

| component | class | model | energy | share % |
|---|---|---|---|---|
| dram_chan | DRAM | const | 962 µJ | 83.1 |
| core0.vfu | fpmac | cal | 61.6 µJ | 5.3 |
| icnt_buf | sram | cal | 34.7 µJ | 3.0 |
| core0.vrf | regfile | cal | 31.2 µJ | 2.7 |
| core0.sfu_pipe | fpsfu | cal | 24.5 µJ | 2.1 |
| core0.array0.pe | fpmac | cal | 4.49 µJ | 0.4 |

```
WARNING (NW-6201): ki6yhkrcpgm: the kernel has no linalg.matmul, so it has no MAC configuration.
WARNING (NW-6208): oqy5oqtice5: the kernel has SFU operations and an unknown operand dtype, so NPUWattch charges the SFU at fp32.
WARNING (NW-6002): The run has no MAC kernel, so non-MAC kernels use an fp32 datapath.
```

A kernel can consume energy even when it has no matrix MAC operation

Part 2 §4 · A kernel with no MAC · 1.16 mJ, 115 µs · Saved notebook output (component table: top 6 of 10)

# ResNet-50: Useful MACs Against Active PE Cycles

```
rep_r = reps["ResNet-50"]; ex_r = example("pytorchsim/tpu_like_fp32/resnet50")
w = windows_df(rep_r); display(w)                          # 19 windows: layout kernels, GEMMs, ReLU, BN
occ = occupancy_table(texts["ResNet-50"], kernel_names(ex_r))    # reads NW-6222 of each GEMM kernel
occ["pJ per useful MAC"] = [round(float(w.set_index("window").loc[k, "total_pJ"]) / m, 1)
                            for k, m in zip(occ["kernel"], occ["useful MACs"])]
display(occ)
```

| kernel | useful MACs | active PE cycles | occupancy % | pJ per useful MAC |
|---|---|---|---|---|
| 01_conv1_p2 | 118,013,952 | 10,074,734,592 | 1.2 | 13.7 |
| 02_res2_1x1_p1 | 12,845,056 | 51,380,224 | 25.0 | 9.7 |
| 03_res2_3x3_p2 | 115,605,504 | 462,684,160 | 25.0 | 6.3 |
| 04_res3_3x3_p2 | 115,605,504 | 115,605,504 | 100.0 | 5.8 |
| 05_res4_1x1_exp_p1 | 51,380,224 | 51,380,224 | 100.0 | 6.6 |
| 06_res5_3x3_p2 | 115,605,504 | 132,120,576 | 87.5 | 9.8 |
| 07_fc | 2,048,000 | 2,097,152 | 97.7 | 163.0 |

occupancy

conv1: 1.2% occupancy

full tiles

**Empty PE slots cost leakage only; fewer useful MACs share the data movement**

Part 2 §5 · ResNet-50 · conv1 tile is 112×8 @ 8×64: 3 real input channels in 8 of 128 rows · Saved output

# ResNet-50: conv1 Against res3, Component by Component

```
display(window_components(rep_r, "01_conv1_p2"))      # one window's per-component energy (report.json)
display(window_components(rep_r, "04_res3_3x3_p2"))
```

## 01_conv1_p2 · 315,300 cycles · 1.62 mJ · occupancy 1.2%

| component | energy |
|---|---|
| core0.array0.pe | 298 µJ |
| core0.array0.w_reg | 166 nJ |
| core0.array1.pe | 298 µJ |
| core0.array1.w_reg | 166 nJ |
| core0.vpu_spad | 299 µJ |
| core0.vrf | 684 nJ |
| core0.vfu | 1.37 µJ |
| core0.dma_q | 3.11 µJ |
| core0.dma_addr | 644 nJ |
| icnt_xbar | 3.61 µJ |
| icnt_buf | 22.1 µJ |
| dram_chan | 631 µJ |

conv1

## 04_res3_3x3_p2 · 9,612 cycles · 672 µJ · occupancy 100%

| component | energy |
|---|---|
| core0.array0.pe | 297 µJ |
| core0.array0.w_reg | 4.41 nJ |
| core0.array1.pe | 287 µJ |
| core0.array1.w_reg | 4.27 nJ |
| core0.vpu_spad | 5.03 µJ |
| core0.vrf | 53.9 nJ |
| core0.vfu | 108 nJ |
| core0.dma_q | 488 nJ |
| core0.dma_addr | 101 nJ |
| icnt_xbar | 565 nJ |
| icnt_buf | 3.46 µJ |
| dram_chan | 76.4 µJ |

res3

Same instance tree in every kernel: the difference is activity, not hardware

Part 2 §5 · Per-kernel component energy · Saved notebook output

# Three Decoder Blocks in Prefill and Decode

```
llm = pd.DataFrame([row(k, v) for k, v in reps.items() if "prefill" in k or "decode" in k])
display(llm.drop(columns=["total_pJ"]))
llm.set_index("design / workload")["total_pJ"].div(1e9).plot.bar(logy=True)   # one decoder block, mJ, log scale
```

| design / workload | total energy | DRAM % | time | avg power | MAC util % | pJ / MAC | pJ / FLOP |
|---|---|---|---|---|---|---|---|
| Llama-3-8B prefill | 6.66e+05 µJ | 12.1 | 7.54 ms | 88.3 W | 49.0 | 5.85 | 2.92 pJ/FLOP |
| Llama-3-8B decode | 4.29e+04 µJ | 90.7 | 4.01 ms | 10.7 W | 0.2 | 192.45 | 96.2 pJ/FLOP |
| Gemma-2-9B prefill | 6.72e+05 µJ | 20.1 | 11 ms | 60.9 W | 30.5 | 6.48 | 3.24 pJ/FLOP |
| Gemma-2-9B decode | 3.85e+04 µJ | 90.8 | 3.07 ms | 12.5 W | 0.2 | 189.65 | 94.8 pJ/FLOP |
| DeepSeek-V2-Lite prefill | 3.37e+05 µJ | 31.4 | 9.7 ms | 34.8 W | 14.7 | 7.68 | 3.84 pJ/FLOP |
| DeepSeek-V2-Lite decode | 1.43e+04 µJ | 89.6 | 1.17 ms | 12.3 W | 0.2 | 165.56 | 82.8 pJ/FLOP |

Each run is one decoder block: prefill processes 512 tokens, decode generates one token against a 512-entry KV cache.
Per token, decode costs about 33× more: 42.9 mJ for one token against 666 mJ for 512. **33× per token**

Prefill reuses every weight for 512 tokens; decode streams the same weights for one

Part 2 §6 · LLM decoder blocks · Saved notebook output (the notebook also draws a log-scale bar chart)

# Llama-3-8B: Prefill Against Decode, Two Counters

```
pre, dec = reps["Llama-3-8B prefill"], reps["Llama-3-8B decode"]
dram_rows = lambda rep: {r["label"]: r["pct"] for r in rep["dram_breakdown"]["rows"]}
pd.DataFrame([{"phase": ..., "useful MACs": ..., "DRAM %": ..., "arrays %": ...,
               **dram_rows(rep)} for rep in (pre, dec)])
```

| Llama-3-8B decoder block | prefill (512 tokens) | decode (1 token) | |
|---|---|---|---|
| useful MACs | 1.14e+11 | 2.23e+08 | **512× fewer MACs** |
| time | 7.54 ms | 4.01 ms | |
| total energy | 666 mJ | 42.9 mJ | |
| DRAM % | 12.1 | 90.7 | **DRAM share flips** |
| arrays % (the pe rows) | 86.4 | 3.4 | **arrays 86 → 3%** |
| DRAM breakdown: row activation (ACT+PRE) | 26.3% | 30.8% | |
| DRAM breakdown: transfer read / write | 62.3% / 7.0% | 64.2% / 0.2% | **DRAM breakdown** |
| DRAM breakdown: refresh | 4.5% | 4.9% | |
| energy per useful MAC | 5.85 pJ | 192.45 pJ | **250× per MAC** |

A much smaller MAC count does not produce a proportional runtime reduction

Part 2 §6.1 · Saved notebook output · block weights ≈ 870 MB fp32: decode reads them once, prefill about twice

# Three Decoder Blocks: Dense, MoE, and Mapping

```
occ_d = occupancy_table(texts["DeepSeek-V2-Lite prefill"], kernel_names(ex_d))   # + cycles, energy, pJ/MAC
g, l = reps["Gemma-2-9B prefill"], reps["Llama-3-8B prefill"]
pd.DataFrame([{"kernel": k, "Gemma cycles": ..., "Llama cycles": ..., "Gemma energy": ..., ...} for k in pairs])
```

## DeepSeek-V2-Lite prefill kernels (occupancy_table)

| kernel | occupancy % | cycles | pJ / useful MAC | |
|---|---|---|---|---|
| 1_q_proj | 99.8 | 139,312 | 5.5 | dense |
| 6_o_proj | 100.0 | 101,607 | 5.6 | |
| 7_router | 50.0 | 16,117 | 7.7 | router |
| 8_experts_up_gate | 99.9 | 4,625,367 | 8.5 | experts |
| 9_experts_down | 99.9 | 3,164,956 | 9.2 | |
| 10_shared_up_gate | 100.0 | 226,925 | 5.6 | |

## Gemma-2-9B against Llama-3-8B, prefill kernels

| kernel | Gemma cyc | Llama cyc | Gemma E | Llama E |
|---|---|---|---|---|
| 1_q_proj | 411,821 | 345,779 | 42.6 mJ | 47.8 mJ |
| 3_attn_qk | 100,824 | 152,117 | 6.65 mJ | 7.26 mJ |
| 6_ffn_up_gate | 5,575,632 | 3,267,126 | 350 mJ | 351 mJ |
| 7_ffn_down | 2,691,664 | 1,716,468 | 173 mJ | 170 mJ |

Gemma vs Llama

10× bytes per MAC

Expert GEMMs stream 48 rows per weight tile instead of 512: about 10× the weight bytes per MAC, same PE row.

Gemma's FFN mapping streams the weights 4.3× from DRAM (1.76 GB for 411 MB) against 2.1× on Llama: a mapping effect.

a mapping effect

Lower total compute can coexist with higher energy per useful MAC

Part 2 §6.2–6.3 · Saved notebook output (energies shown in mJ; the notebook prints µJ)

# Q&A and Tutorial Resources

### NPUWattch hands-on

github.com/yonseicasl/NPUWattch/tree/main/tutorial/hands_on

### Program structure and input formats

tutorial/npuwattch_basic/part3_structure.md
tutorial/npuwattch_basic/part4_data.md

**Sehyeon Kim · ikamusume@yonsei.ac.kr**