# Scalable system simulation and performance analysis for RISC-V architecture using machine learning workloads

Debjyoti Bhattacharjee
*Compute System Architecture (CSA) imec, Belgium*

Tommaso Marinelli
*Compute System Architecture (CSA) imec, Belgium*

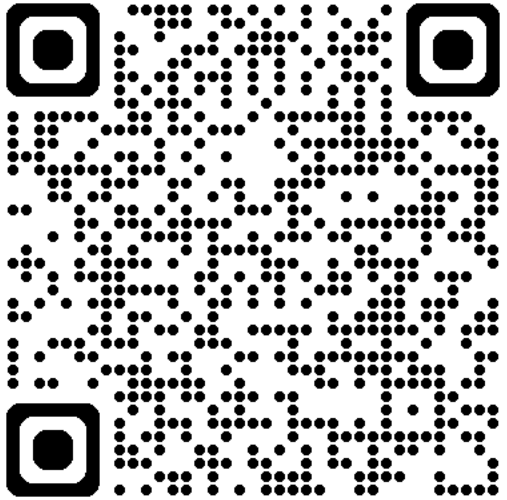
imec

*Learn. Simulate. Scale.*

ISPASS

# Workshop contents

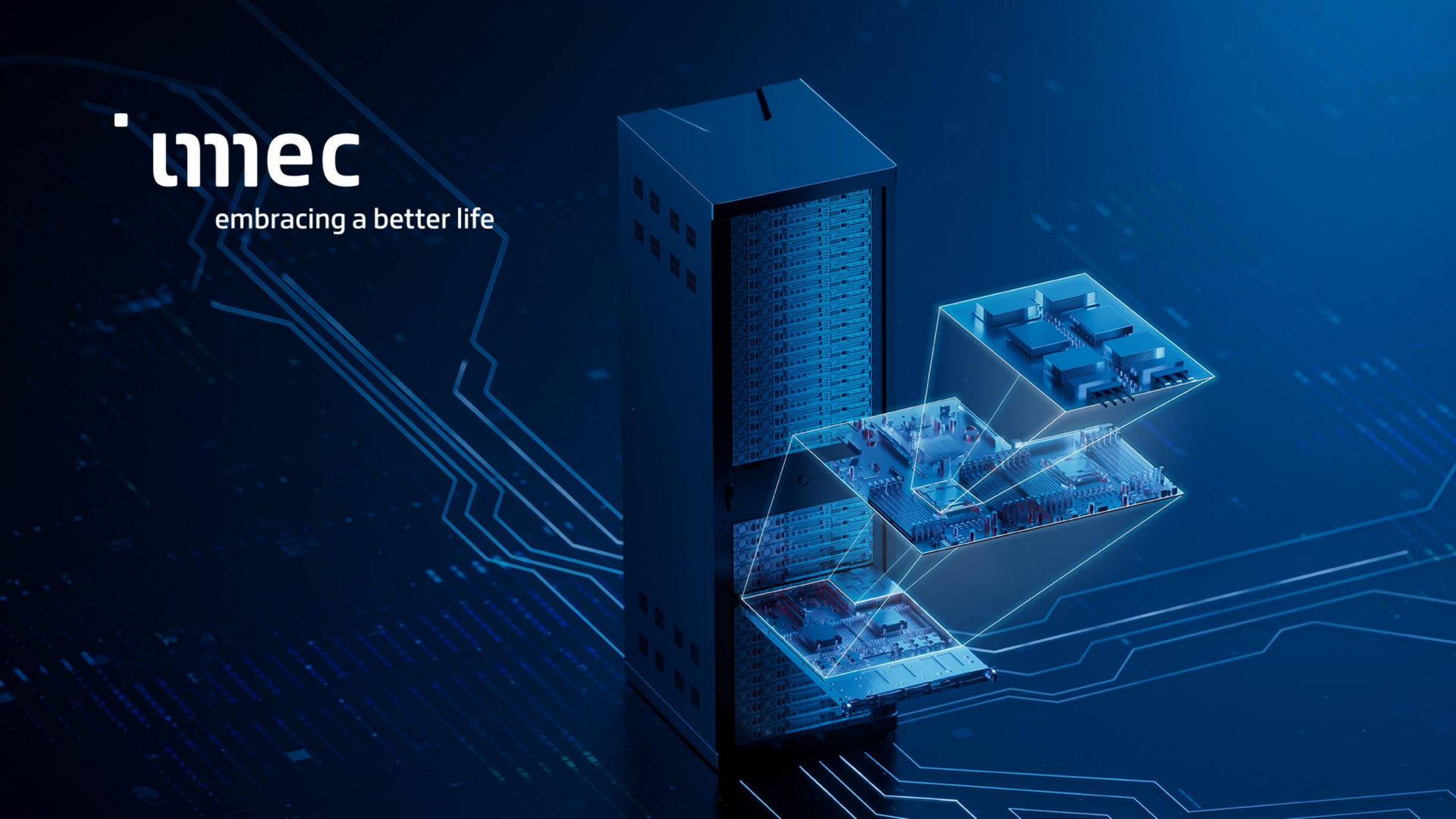
- All the materials are available online via github
  - Slides are available here
  - Docker environment is provided
  - Try it out yourself — *During or After the Workshop!*
  - File issues on the github if you need help



https://github.com/CSA-infra/RISCV-Scalable-Simulation-tutorial

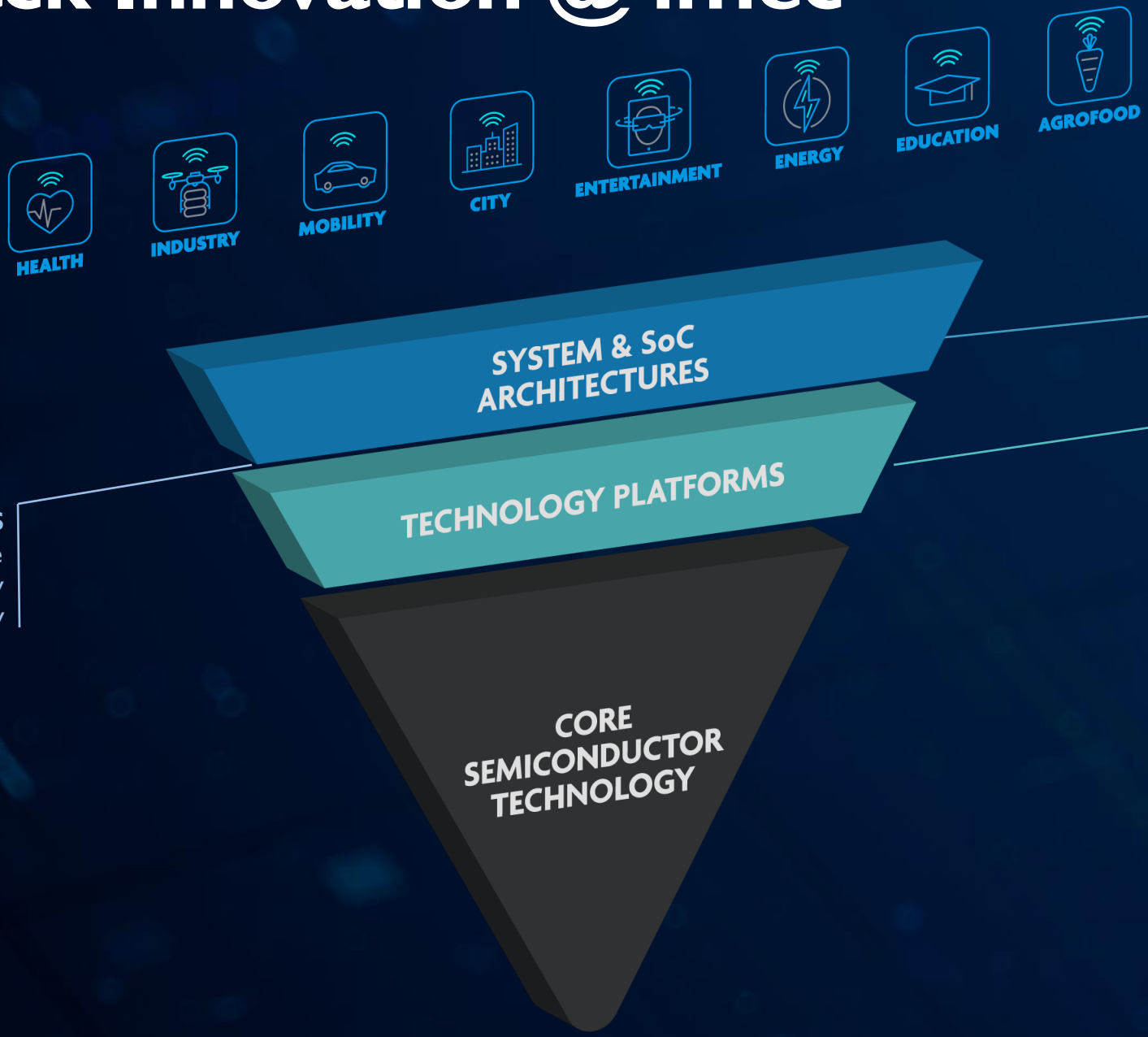
imec

embracing a better life




THE NETHERLANDS
EINDHOVEN, WAGENINGEN, NIJMEGEN
UK
CAMBRIDGE
BELGIUM
LEUVEN (HQ), GHENT
ANTWERP, BRUSSELS,
KORTRIJK & GENK
USA
MICHIGAN
USA
PURDUE
USA
BERKELEY
USA
BOSTON
USA
SAN FRANCISCO
USA
FLORIDA
USA
SAN JOSE
SPAIN
MÁLAGA
ITALY
NAPOLI
GERMANY
HEILBRONN
KOREA
SEOUL
JAPAN
TOKYO
TAIWAN
HSINCHU
INDIA
BANGALORE
R&D
Representative office

# Cross-Stack Innovation @ imec

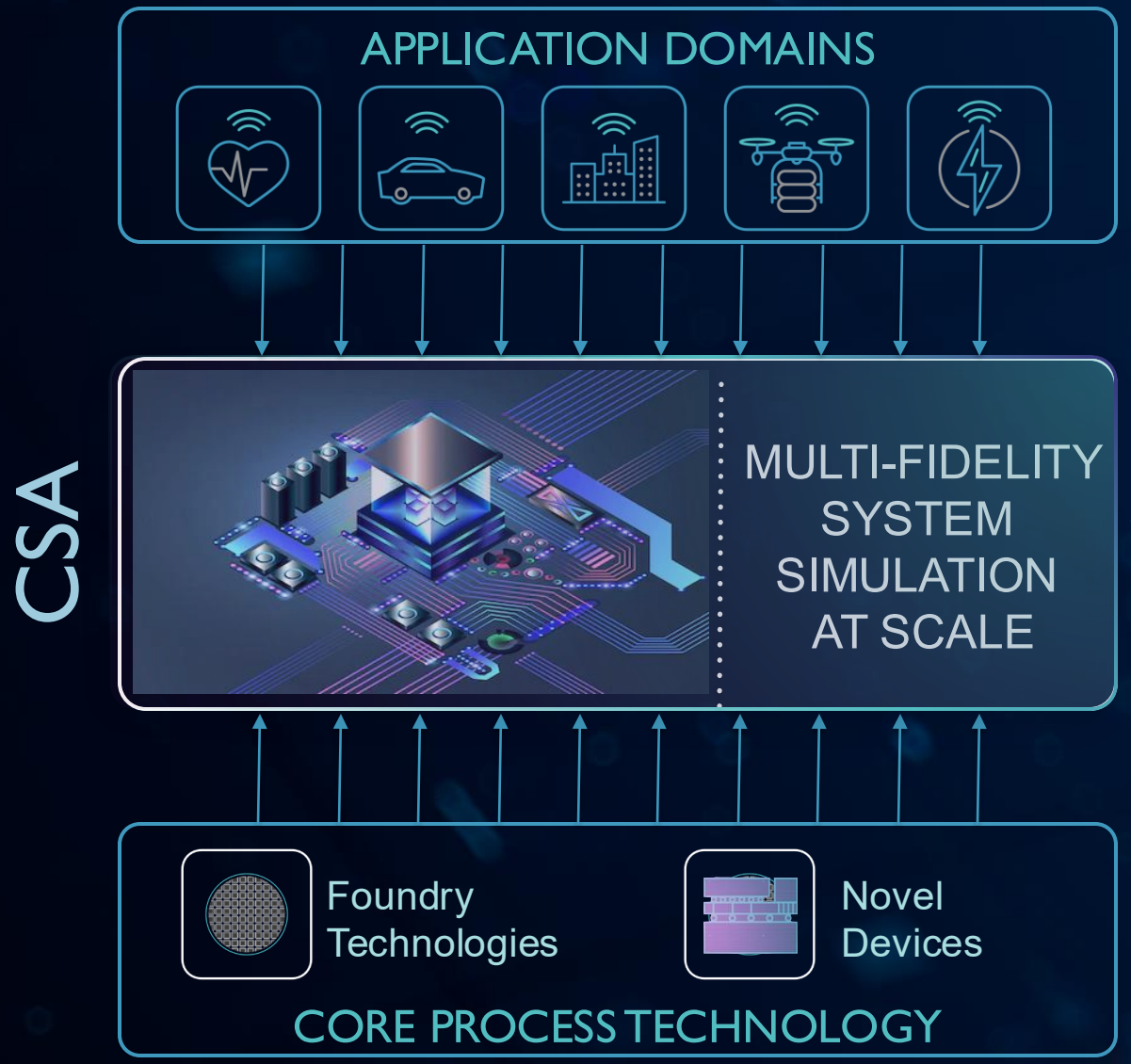



**SYSTEM PLATFORMS**
**Compute System Architecture**
Optical beamforming
DNA sequencing
Nanofluidics processor
RF technologies for (beyond) 5G
Hyperspectral imaging
…

**DIGITAL TECHNOLOGIES**
Artificial intelligence
Privacy / cybersecurity
(Beyond) 5G connectivity technology

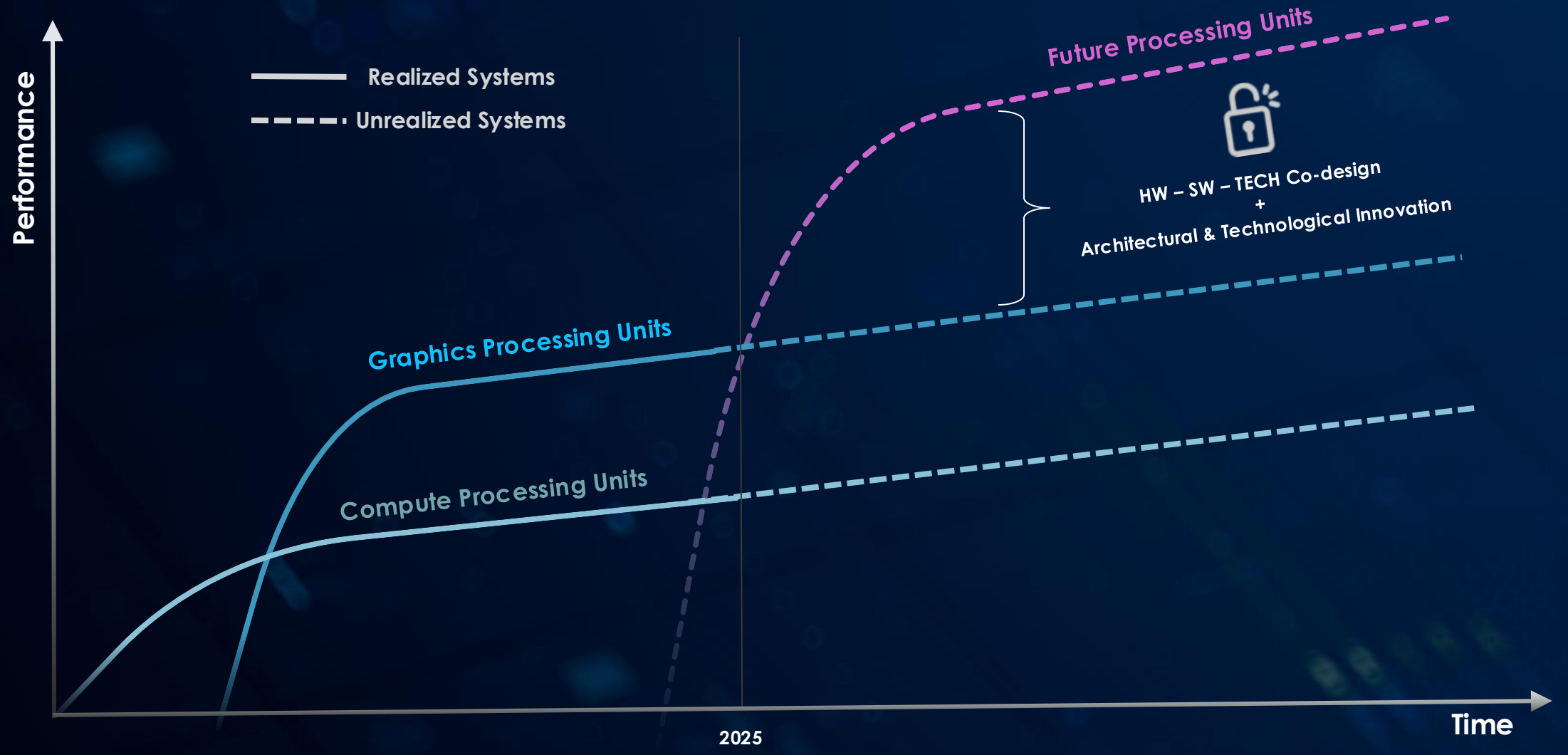
**TECHNOLOGY PLATFORMS**
Logic CMOS
Memory
3D heterogeneous integration
Photonics
MEMS & microfluidics
Sensor platform
Flexible technologies

# CSA : Areas of Expertise




- Domain-specific Architectures
- Performance and TCO Analysis
- HW Validation and Calibration
- Technology Impact Evaluation

# Enabling the next breakthrough in performance



# Agenda

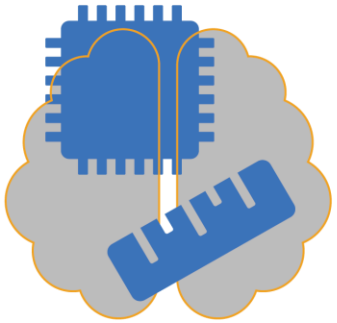
| Topic | Timeline |
|---|---|
| Introduction to System Simulation | 13:45-14:05 |
| Detailed Single Node simulation for ML workloads using gem5 x MLIR | 14:05-15:15 |
| Break | 15:15-15:25 |
| Scalable multi-node simulation using SST<br>▪ *Exploring multi-scale parallelism (MPI+OMP) with Instruction-level Simulation*<br>▪ *Exploring the scaling of LLM training with Packet-level Simulation* | 15:25-16:30 |

ML: Machine Learning MLIR: Multi-Level Intermediate Representation SST: Structural Simulation Toolkit LLM: Large Language Model

# System-Level Simulators

- Modern systems are increasingly complex
  - Requires accurate and scalable models to evaluate hardware-software interactions.
- Physical prototyping is expensive, time-consuming, and infeasible during early design stages.
- Simulators enable rapid exploration, performance analysis, and optimization of system architectures

gem5



SST

Astra-Sim

The Sparta Modeling Framework

AccelSim

https://www.gem5.org/ https://sst-simulator.org/
https://sparcians.github.io/map/index.html https://accel-sim.github.io/
https://scalesim-project.github.io/ https://astra-sim.github.io/

# System-Level Simulators

## Role of Simulation in Hardware-System Co-Design

- Use **system-level simulators** to model components (CPUs, memory, interconnects) and their interactions.
- Select/combine tools like **gem5** (detailed architectural modelling) and **SST** (scalable simulations) to analyse real-world scenarios
  - In most cases, a combination of multiple simulation tools are required to come to an "actionable" decision
- Use **representative benchmarks** and **design of experiments (DoE)** to ensure accuracy and scalability.

# System-Level Simulators

## Features and Limitations

- Typically allows modelling of a "hardware" component at high level of abstraction
  - Usually much less details than RTL
- Performance estimation and bottleneck analysis can be done
  - *Eg: Pipeline viewer to see stalls*
- Serves well to do "design space exploration" and answer "what-if" question(s)
  - *Eg: What if the processor clock could run at 30Ghz?*

How does "area", "energy" change with "technology"?

Direct estimation is not feasible.
Needs additional tools.

It does not say if it is feasible to design a CPU with 30Ghz clock.
RTL validation and possible full mapping to technology library would be necessary.

# Design of Experiments (DoE)

| | |
|---|---|
| **Purpose** | • Systematically explore design spaces to identify optimal configurations. |
| **Key Metrics** | • Define figures of merit (FoM)<br>• *Eg: Misses Per Kilo Instructions (MPKI) executed* |
| **Parameter Selection** | • Choose system variables to analyze.<br>• *Eg: cache size, memory latency* |
| **Experiment Planning** | • Use structured approaches (e.g., factorial design) to reduce simulation overhead. |
| **Result Analysis** | • Extract insights to understand potential bottlenecks and areas of improvement |

# Design of Experiments (DoE)

## Choosing "Figure of Merit"

- Choosing the "figure of merit" is a very critical aspect in the DoE
- If a simulator cannot be used to report or derive an "figure of merit"
  - Choose a different simulator
  - Extend it or combine it with some other simulator → This is mostly what happens in practice!
- Often a combination of multiple metrics might be interesting
  - *Eg: Area-Delay Product, Throughput/mm^2.*

# Design of Experiments (DoE)

## Choosing "Figure of Merit" → Machine Learning Context

- Systems built for machine learning are ubiquitous now
  - Powering everything from data centers to edge devices.
- Operational costs and energy efficiency is a well accepted FoM
  - TOPs/W → Measure of "energy efficiency"
  - TOPs/(W $mm^2$) → Ties performance, power, and area together
- Environmental impact is become an important FoM
  - Carbon Emissions related to manufacturing of the chips
  - https://netzero.imec-int.com/

# Design of Experiments (DoE)

## Benchmarks for Evaluation

- Choice of benchmarks is as important as choice of "FoM"
  - *Eg: If you want to evaluate a multi-node system, MPI workloads might be interesting*
- It always helps to use "well-accepted" benchmarks to compare across different systems of similar kind
  - *Eg: SPEC CPU® 2017, LINPACK benchmarks, etc.*

Benchmarks need to be represented/compiled in a way that the simulator/system can ingest!

# Case Study

## Mobile memory exploration

### Goals

- Analyse memory bottlenecks of emerging mobile SoCs executing GenAI workloads
- Estimate performance improvements with emerging low-power DRAM standards

### Main parameters

- DRAM frequency per pin, DRAM bus width

### Workloads

- Transformer models (e.g. BERT, GPT, Llama)

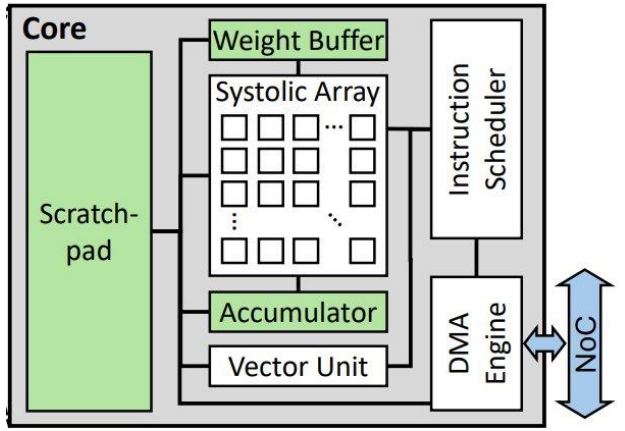
### Figures of merit

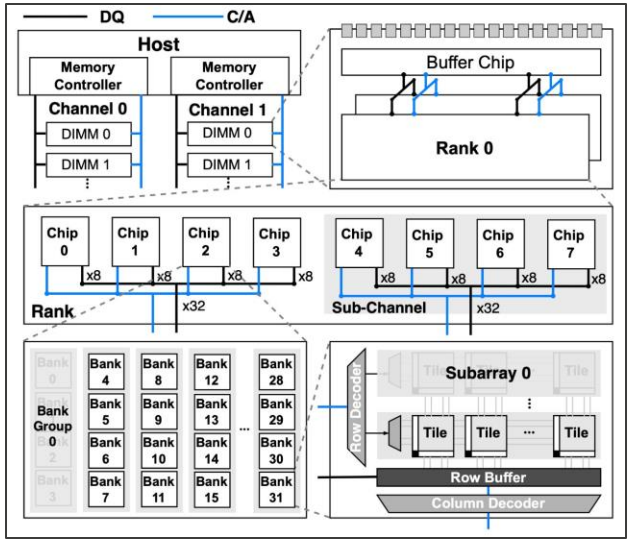
- Effective DRAM bandwidth
- Tokens per Second (TPS)

Performance data
(Energy and area need additional tools)

### Platform

- NPU simulator → ONNXim (cycle-level)
- DRAM simulator → Ramulator 2.0 (cycle-level)



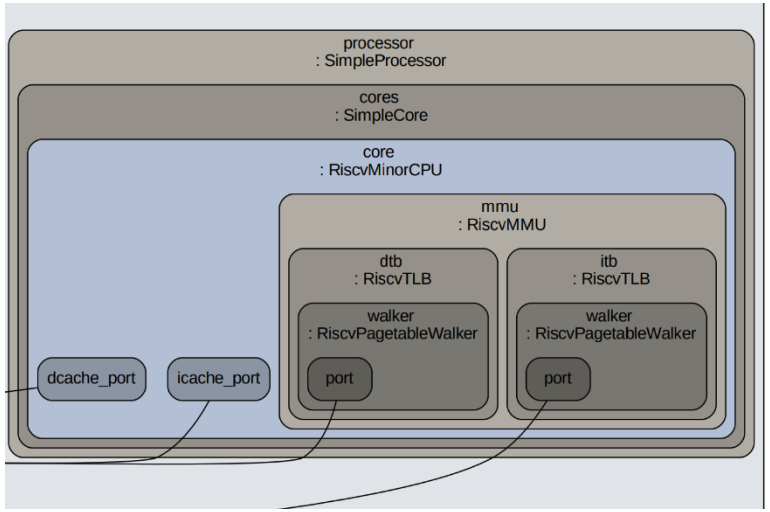
# Agenda

| Topic | Timeline |
|---|---|
| Introduction to System Simulation | 13:45-14:05 |
| Detailed Single Node simulation for ML workloads using gem5 x MLIR | 14:05-15:15 |
| Break | 15:15-15:25 |
| Scalable multi-node simulation using SST<br>▪ *Exploring multi-scale parallelism (MPI+OMP) with Instruction-level Simulation*<br>▪ *Exploring the scaling of LLM training with Packet-level Simulation* | 15:25-16:30 |

ML: Machine Learning MLIR: Multi-Level Intermediate Representation SST: Structural Simulation Toolkit LLM: Large Language Model

# gem5
## A brief introduction

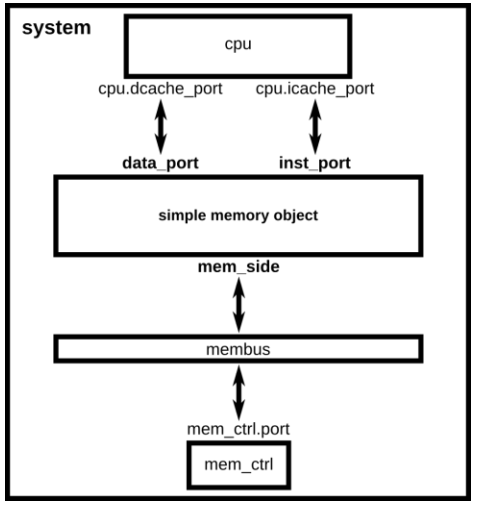
- Modular discrete event driven computer system simulator platform
  - Used widely in academia as well as industry
- Can model a full system: CPU + memory + devices
  - Potential support for GPU and accelerators too
- Ability to execute real workloads and even a full OS
  - Different modes of execution according to needed accuracy and speed

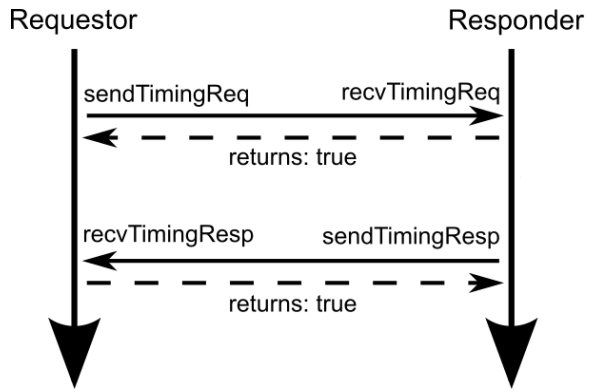



A part of a simulated system

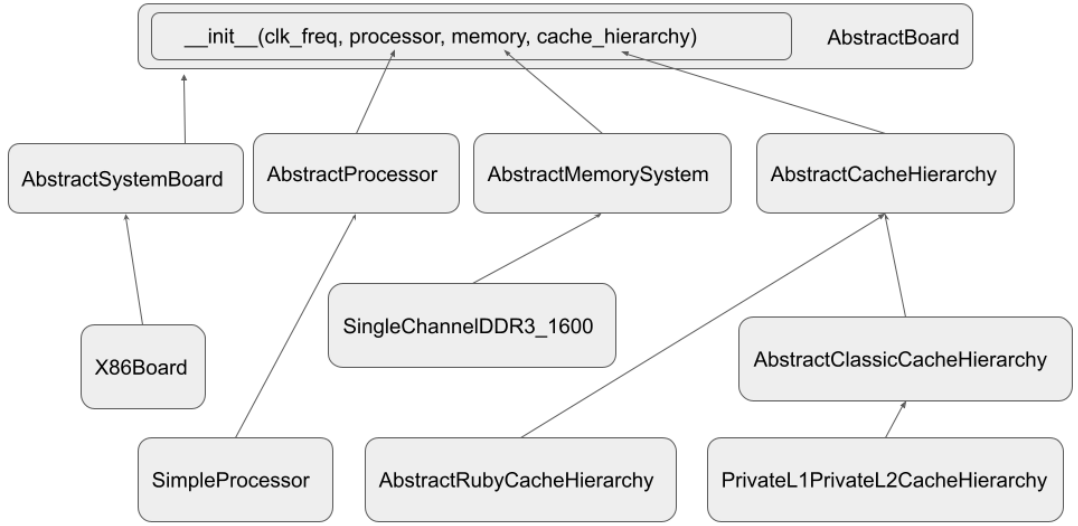
# gem5
## Simulation objects

- Describes a system as a collection of objects (*SimObjects*)
  - Generally implemented in C++
  - Port interface + packets for communication between memory objects
- Objects are connected via Python scripts
  - Each component is parameterized as well
  - Statistics per component is reported
- New: gem5 standard library defining a series of standard components
  - Board → processor + memory system + cache hierarchy




gem5: Creating SimObjects in the memory system




gem5: Developing Your Own Components Tutorial

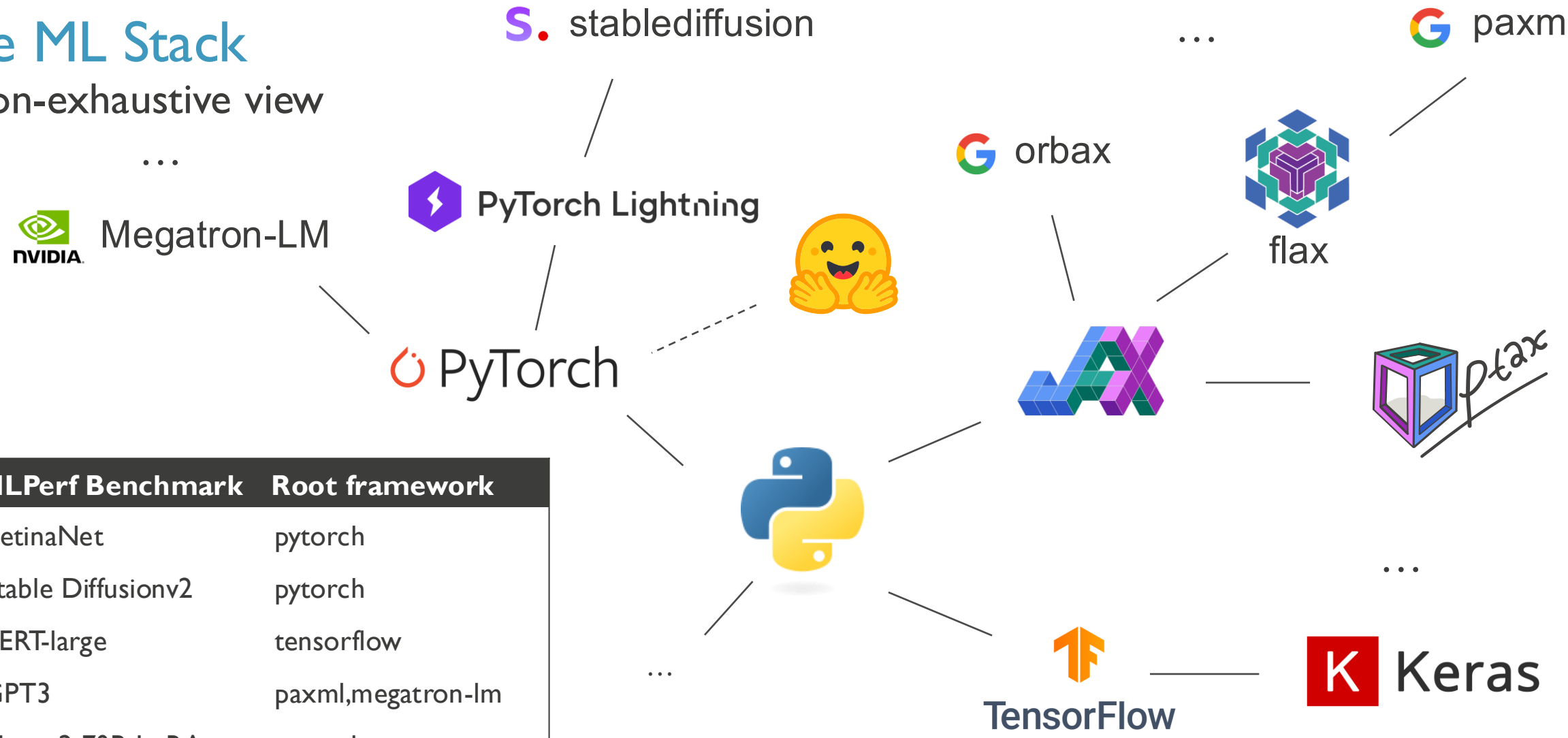
# gem5
## Flexibility

- gem5 core component micro-architecture is separate from the ISA
  - *Eg: MinorCPU models an in-order architecture →*
    *Can be combined with "RISCV64" or "AARCH64"*
- Different choices for main memory backend
  - Simple model, accurate DDR*/HBM, external simulator
- Different cache hierarchies and cache coherency protocols can be modelled as well
  - MOESI-only with classic memory system
  - More flexibility (e.g CHI) using Ruby memory system

| Attribute | Type/version |
|---|---|
| **Core Type** | MinorCPU, O3CPU |
| **Core Freq.** | 2 GHz |
| **L1 Cache** | 64KB, 4-way |
| **L2 Cache** | 8MB, 4-way |
| **DRAM Type** | simpleMem |
| **DRAM Size** | 3GB |
| **DRAM Freq.** | 1 GHz |

# The ML Stack

## A non-exhaustive view

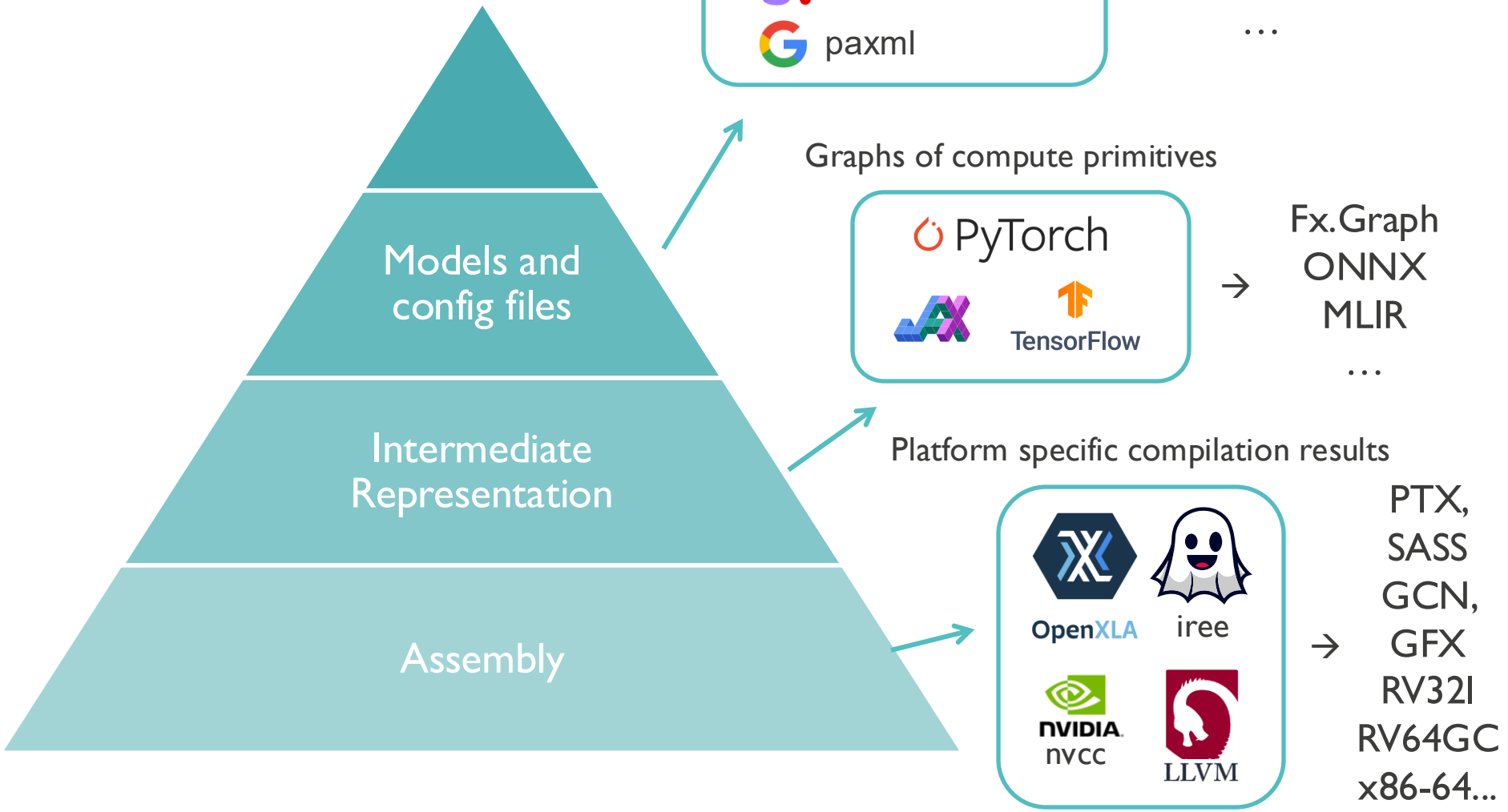



| MLPerf Benchmark | Root framework |
|---|---|
| RetinaNet | pytorch |
| Stable Diffusionv2 | pytorch |
| BERT-large | tensorflow |
| GPT3 | paxml,megatron-lm |
| LLama2 70B-LoRA | pytorch |
| DLRMv2 | torchrec |
| RGAT | pytorch |

**Complex, modular, and evolving stacks**

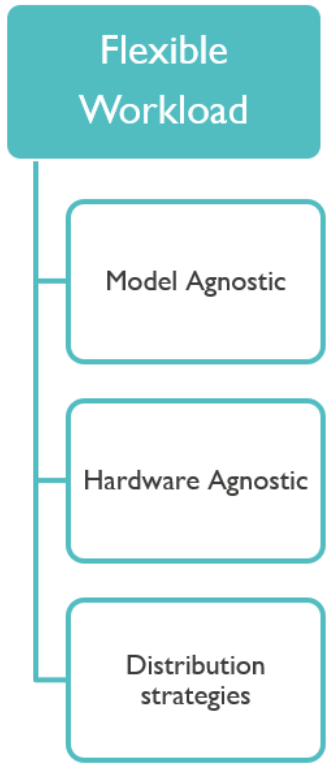
# Workload Representation

- We are considering Machine Learning Inference task as workload in this tutorial



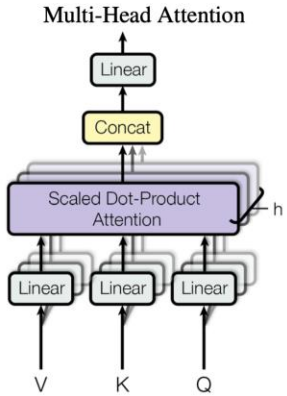
# Mapping Workloads to System Simulations

- Recall: Benchmarks need to be represented/compiled in a way that the simulator/system can ingest!



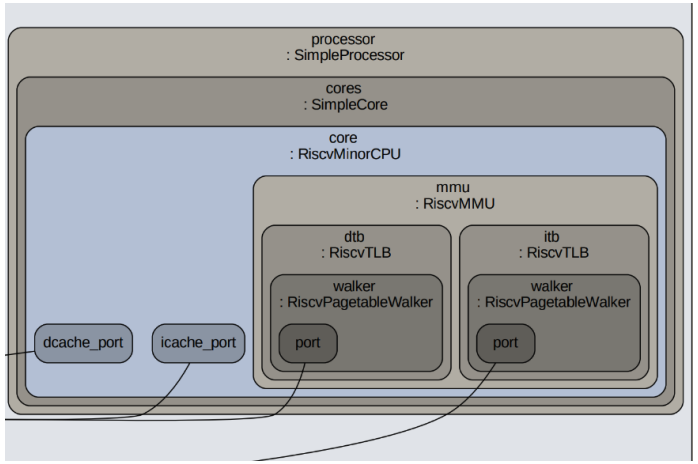

Multi-level IR (MLIR) provides a flexible intermediate representation for machine learning workloads

- MLIR can be exported from popular ML frameworks
- MLIR compilers and runtimes exist for different hardware targets







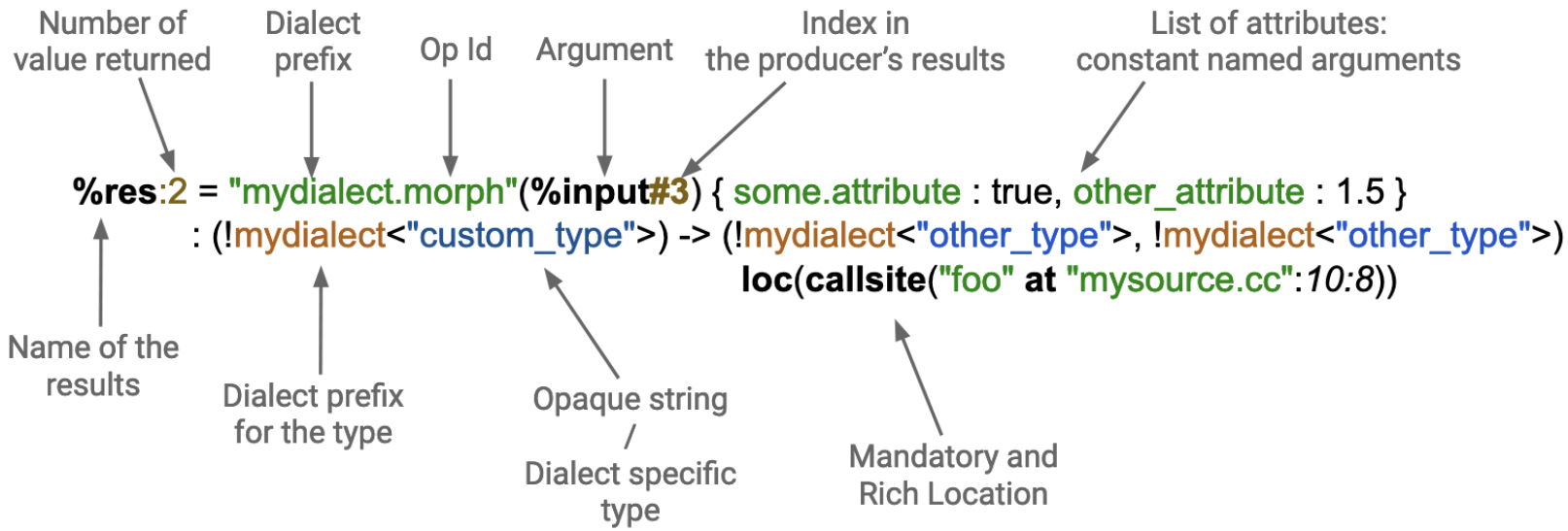


# Mapping Workloads to System Simulations

## A representative MLIR operation

### Operations, Not Instructions

- No predefined set of instructions
- Operations are like “opaque functions” to MLIR




*https://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf

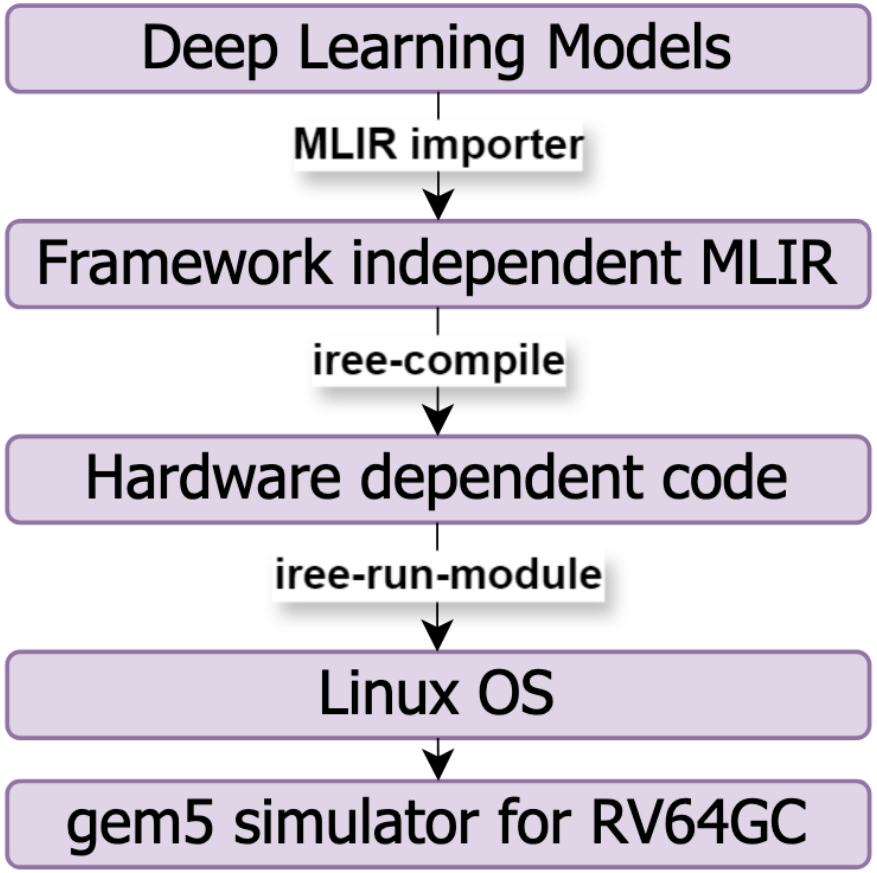
# Mapping Workloads to System Simulations

```
module @pmap__unnamed_wrapped_function_ attributes {mhlo.num_partitions = 1 : i32, mhlo.num_replicas = 4 : i32} {
func.func public @main(%arg0: tensor<i32> {mhlo.is_same_data_across_replicas = true}, %arg1: tensor<64xf32> ...
%1 = stablehlo.reshape %arg433 : (tensor<1x64xi32>) -> tensor<64xi32>
%c = stablehlo.constant dense<0> : tensor<i32>
%2 = stablehlo.subtract %arg0, %c : tensor<i32>
%c_0 = stablehlo.constant dense<0> : tensor<i32>
%c_1 = stablehlo.constant dense<1000> : tensor<i32>
%3 = call @clip(%2, %c_0, %c_1) : (tensor<i32>, tensor<i32>, tensor<i32>) -> tensor<i32>
%4 = stablehlo.convert %3 : (tensor<i32>) -> tensor<f32>
%cst = stablehlo.constant dense<1.000000e+03> : tensor<f32>
%5 = stablehlo.divide %4, %cst : tensor<f32>
%cst_2 = stablehlo.constant dense<1.000000e+00> : tensor<f32>
%6 = stablehlo.subtract %cst_2, %5 : tensor<f32>
%7 = call @integer_pow(%6) : (tensor<f32>) -> tensor<f32>
...
```

*https://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf

# Mapping Workloads to System Simulations

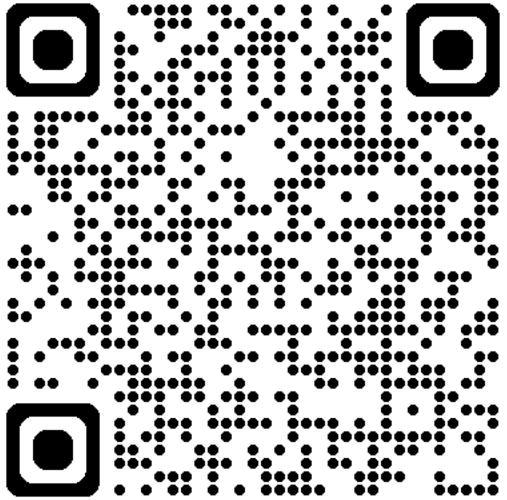
- **Dialect Usage**: Leverage MLIR's dialects to model ML computations effectively.
  - *Eg: Stablehlo*
- **IREE Compilation***: Use IREE to lower MLIR representations into RISC-V binaries (depending on chosen ISA)
  - Applies multiple MLIR's transformation passes to optimize workloads
  - Also, embeds scheduling information
- **End-to-End Flow:** Execute the generated code using *iree-run-module* on the simulated system




*https://iree.dev/

# Workshop Contents

- All the materials are available online via github
  - Slides are available here
  - Docker environment is provided
  - Try it out yourself — *During or After the Workshop!*
  - File issues on the github if you need help



https://github.com/CSA-infra/RISCV-Scalable-Simulation-tutorial

# Structure of a gem5 Experiment

## Simulation input

- System configuration and simulation behaviour is defined through Python scripts
- Different type of scripts:
  - Completely custom
    - gem5: Creating a simple configuration script
  - Legacy / do-it-all scripts
    - E.g. gem5/configs/deprecated/example/fs.py or gem5/configs/example/riscv/fs_linux.py
  - Standard library scripts
    - Attempt at making configuration more modular
    - Example provided in this slide

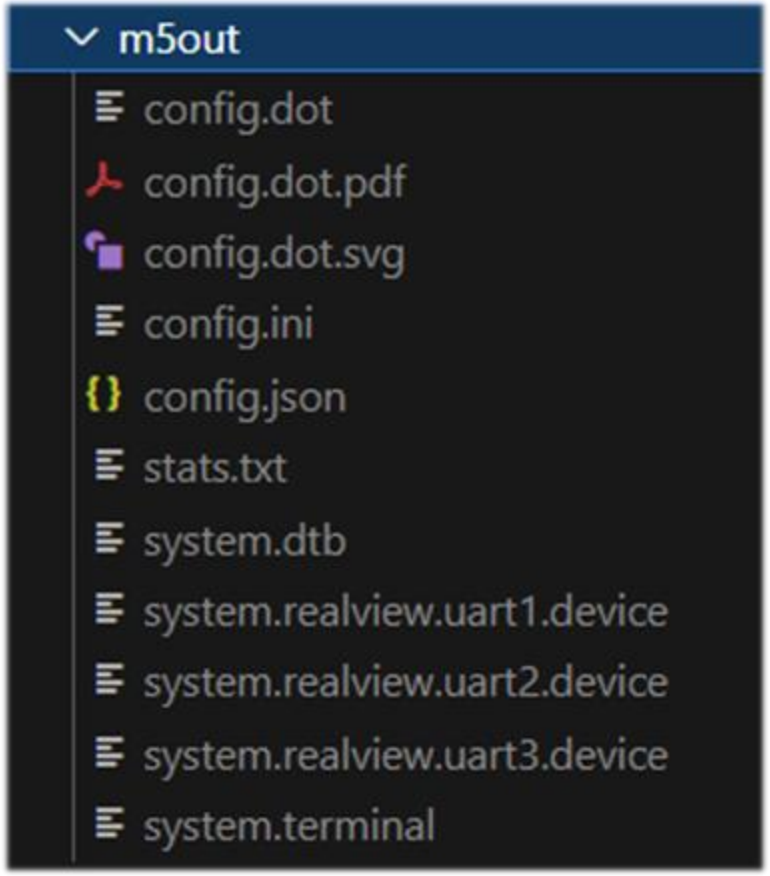
```python
from gem5.components.boards.simple_board import SimpleBoard
from gem5.components.cachehierarchies.classic.no_cache import NoCache
from gem5.components.memory import SingleChannelDDR3_1600
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.isas import ISA
from gem5.resources.resource import obtain_resource
from gem5.simulate.simulator import Simulator
from gem5.utils.requires import requires

requires(isa_required=ISA.ARM)

cache_hierarchy = NoCache()

memory = SingleChannelDDR3_1600(size="32MiB")

processor = SimpleProcessor(cpu_type=CPUTypes.TIMING, isa=ISA.ARM, num_cores=1)

board = SimpleBoard(
    clk_freq="3GHz",
    processor=processor,
    memory=memory,
    cache_hierarchy=cache_hierarchy,
)

board.set_se_binary_workload(
    obtain_resource("arm-hello64-static", resource_version="1.0.0")
)

simulator = Simulator(board=board)
simulator.run()

print(
    "Exiting @ tick {} because {}.".format(
        simulator.get_current_tick(), simulator.get_last_exit_event_cause()
    )
)
```

# Structure of a gem5 Experiment

## Simulation outputs

Several files are produced after simulation:

- Statistics file (*stats.txt*)
  - Contains all the simulation metrics
  - "*Performance counters on steroids*"
- System configuration report (*config.ini* / *config.json*)
  - List of instantiated components with their own parameters
  - Useful for cross-checking ("*was my parameter applied properly?*")
- System visual representation (*config.dot.pdf* / *config.dot.svg*)
  - Only generated if pydot+graphviz is installed
- Terminal output (e.g. *system.terminal*)
  - For FS mode simulations



# Structure of a gem5 Experiment

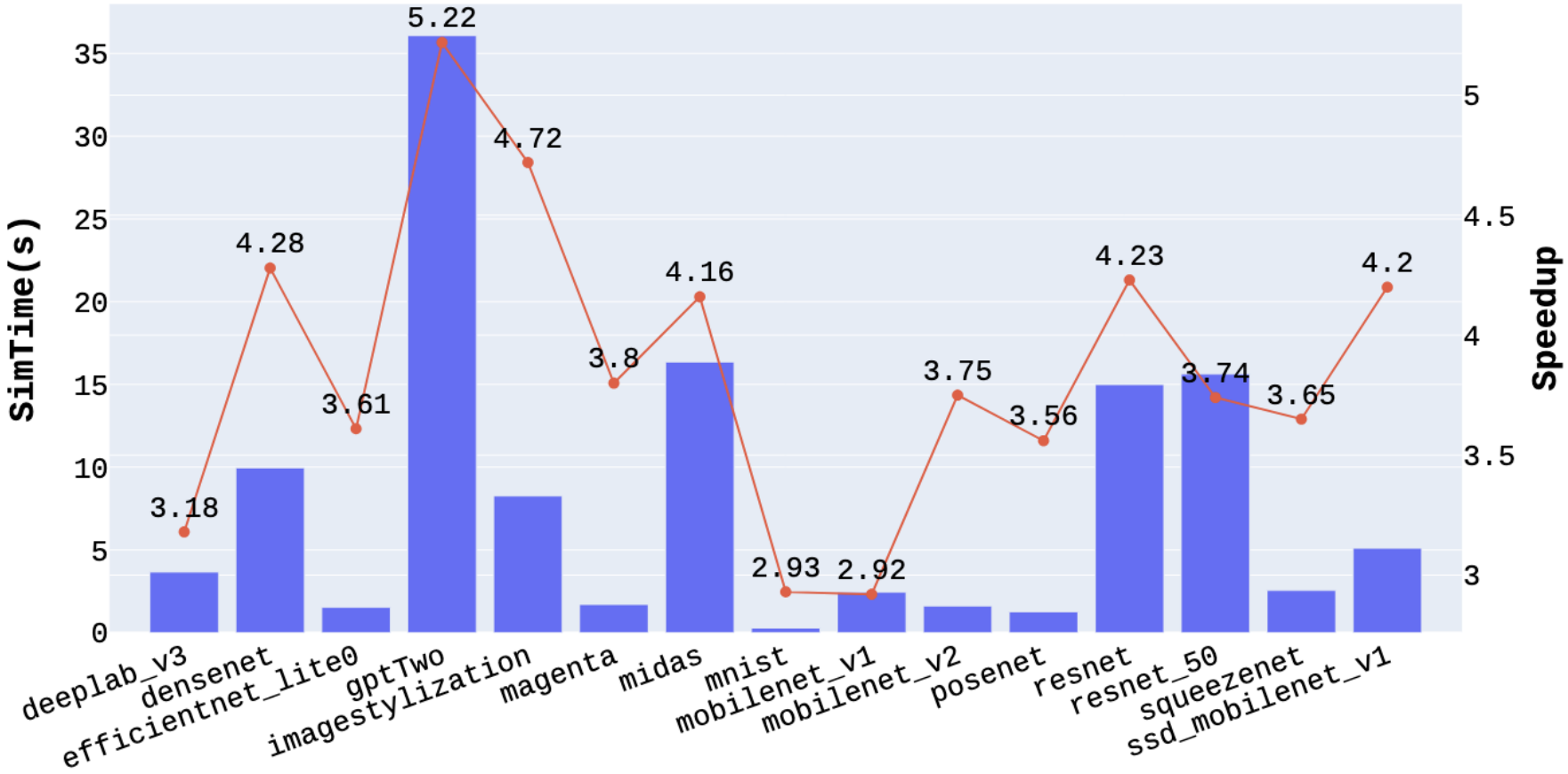
## Statistics file

- Huge text file (or HDF5)
  - Post-processing is your friend ;)
- Dividers to delimit sections (begin/end)
  - Default: single section for whole execution
  - Possible to have several windows using simulation helpers / m5ops
- Some examples of statistics:
  - Execution times (simulated system, host)
  - Caches miss rate
  - Instruction mix
  - Buses occupation
  - Branch mispredictions
  - ...

```
---------- Begin Simulation Statistics ----------
simSeconds                                   0.000032
simTicks                                     31574865
finalTick                                    31574865
simFreq                                1000000000000
hostSeconds                                      0.05
hostTickRate                                683660604
hostMemory                                   16962888
simInsts                                        6685
simOps                                          6694
hostInstRate                                  144534
hostOpRate                                    144716
[...]
board.cache_hierarchy.l1dcaches.demandHits::processor.cores.core.data         2339
board.cache_hierarchy.l1dcaches.demandHits::total         2339
board.cache_hierarchy.l1dcaches.overallHits::processor.cores.core.data         2339
board.cache_hierarchy.l1dcaches.overallHits::total         2339
board.cache_hierarchy.l1dcaches.demandMisses::processor.cores.core.data          132
board.cache_hierarchy.l1dcaches.demandMisses::total          132
board.cache_hierarchy.l1dcaches.overallMisses::processor.cores.core.data          132
board.cache_hierarchy.l1dcaches.overallMisses::total          132
[...]
board.cache_hierarchy.membus.reqLayer0.occupancy       393847
board.cache_hierarchy.membus.reqLayer0.utilization          0.0
board.cache_hierarchy.membus.respLayer1.occupancy      1107152
board.cache_hierarchy.membus.respLayer1.utilization          0.0
[...]
board.processor.cores.core.branchPred.committed_0::NoBranch            8      0.55%      0.55%
board.processor.cores.core.branchPred.committed_0::Return          121      8.32%      8.87%
board.processor.cores.core.branchPred.committed_0::CallDirect           96      6.60%     15.46%
board.processor.cores.core.branchPred.committed_0::CallIndirect           29      1.99%     17.46%
board.processor.cores.core.branchPred.committed_0::DirectCond         1095     75.26%     92.71%
board.processor.cores.core.branchPred.committed_0::DirectUncond           95      6.53%     99.24%
board.processor.cores.core.branchPred.committed_0::IndirectCond            0      0.00%     99.24%
board.processor.cores.core.branchPred.committed_0::IndirectUncond           11      0.76%    100.00%
board.processor.cores.core.branchPred.committed_0::total         1455
[...]
```

# Exp1: Simulating a RISC-V Core with gem5

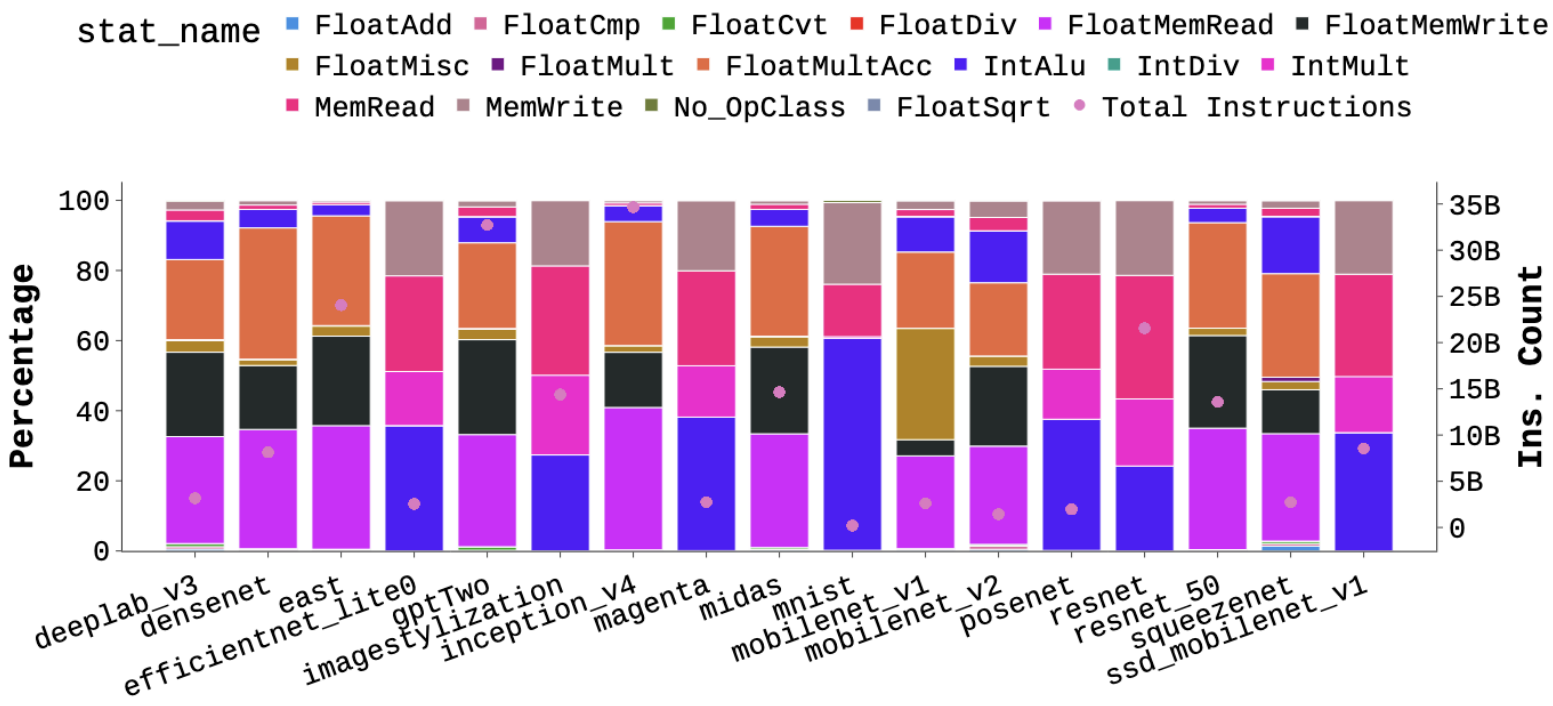
- Performance of the workloads running on
  - *in-order CPU* (Minor CPU)
  - Speedup of *Out-of-order(O3) CPU* vs *in-order CPU*




https://arxiv.org/pdf/2405.15380

# Exp2: Understanding the instruction mix of workloads

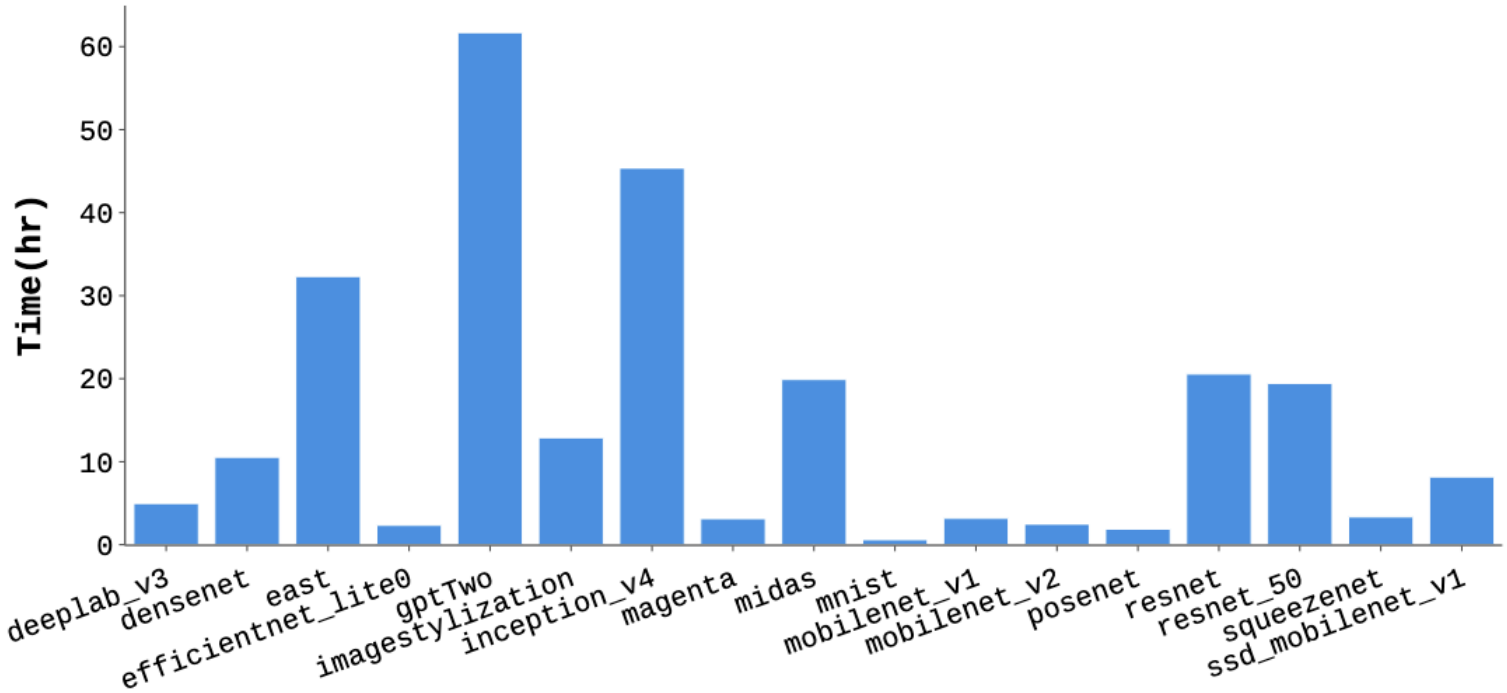
- Each run of gem5 dumps the different types of instructions executed by the CPU
- Can you spot which ones are “integer”/ “quantized” models by looking at this plot?




https://arxiv.org/pdf/2405.15380

# Exp3: How long does it take to simulate each workload?

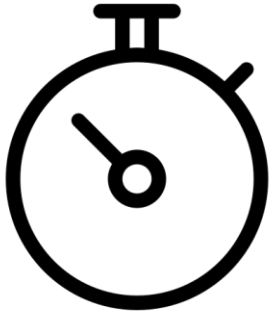
- Gem5 out-of-the-box is a "single-threaded" simulator
- For large workloads, e.g. gptTwo > 2 days to simulate
- For even larger LLMs, this would grow intractable!




https://arxiv.org/pdf/2405.15380

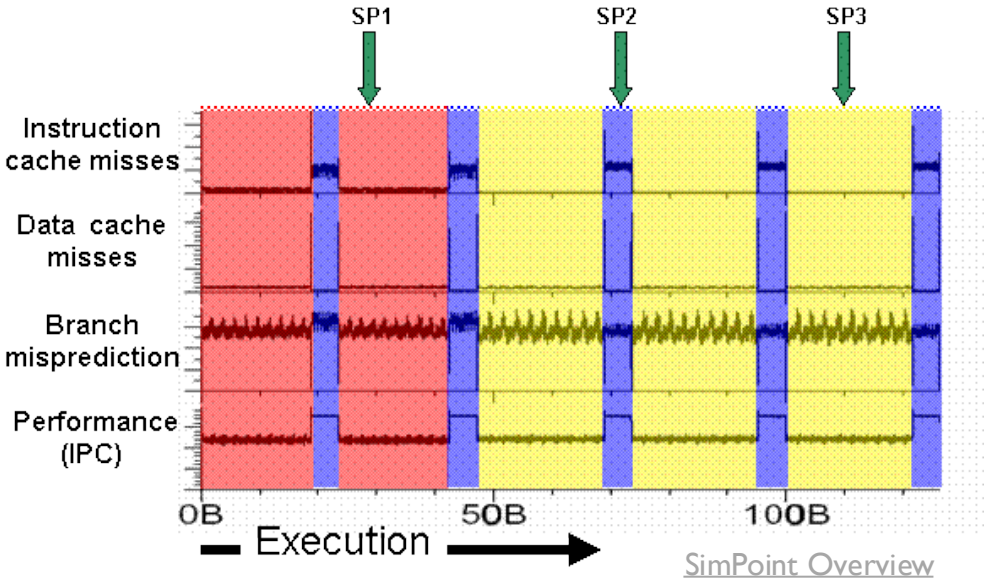
# How to speed up execution?

## Regions of interest (ROIs)



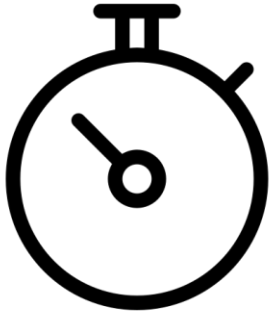
Identify regions of interest in applications

- *Is it really needed to simulate a full workload to assess system performance?*
  - Statistical analysis to identify relevant portions and discard redundant phases (e.g. SimPoint, LoopPoint) → *targeted sampling*
  - Manual workload annotation




SimPoint Overview

# How to speed up execution?
## Checkpoints

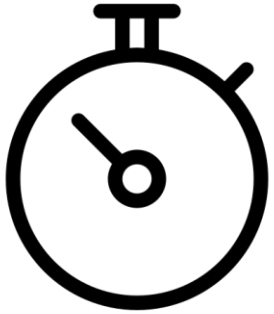


Use checkpoints in simulation

- Actual snapshots of a simulation execution at a certain point in time
  - Dump memory state + CPU state + device state + extra info
- Allow a new simulation to start from that point (e.g. skip boot)
- Allow modifications to the memory system
  - Caches always start from a fresh state
  - Warmup to mitigate the impact of cold misses
- Can align with ROIs

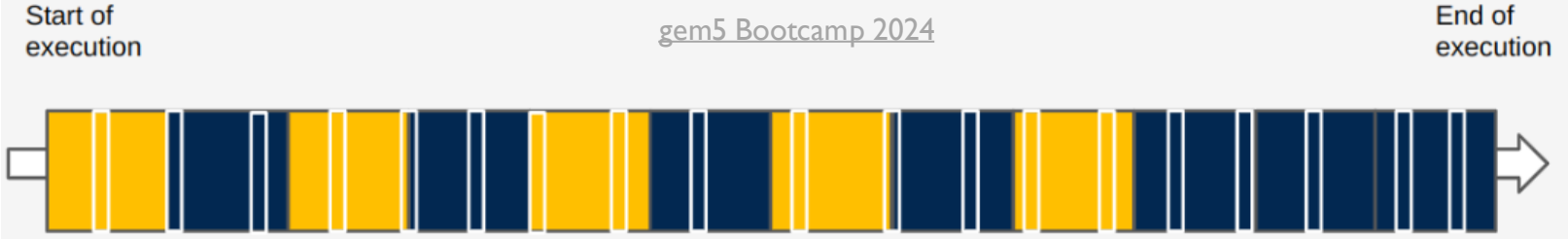
# How to speed up execution?

## Fast-forwarding



Use fast-forwarding between phases

- Start simulation with a faster, low accuracy processor, then switch to a more accurate one midway
- Can be used for *statistical sampling* (continued switching with defined time intervals)
- Can use hardware acceleration/KVM when supported
- Can align with ROIs



# Calibration and validation

Important question: can we *trust* simulation data? Are they realistic?

- Accurate behaviour models and advanced features may be missing in simulated models
  - Lack of interest, effort to produce and maintain code
  - Information is not always public domain ☺
- Still, a lot can be done just properly *calibrating* timing parameters and element sizes
  - Need to *validate* with real hardware
  - Micro-benchmarks are usually used before bigger workloads

- Lot of interest in academia and industry
  - A. Akram and L. Sawalha, "Validation of the gem5 Simulator for x86 Architectures," 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Denver, CO, USA, 2019, pp. 53-58, doi: 10.1109/PMBS49563.2019.00012.
  - Kunal Pai, Zhantong Qiu, Jason Lowe-Power, "Validating Hardware and SimPoints with gem5: A RISC-V Board Case Study", ISCA 2023: gem5 Workshop.
  - U. Shahid, A. Ahmad and S. Wasim, "Gem5-Based Evaluation of CVA6 SoC: Insights into the Architectural Design," 2024 (ISPASS),

†Roughly speaking, errors < 15% are considered to be good enough in the simulation world



†Speaker believes so!

*Scaling out the simulation is essential to model and analyse the complex large, interconnected systems efficiently*

# 10 Min Break

# Agenda

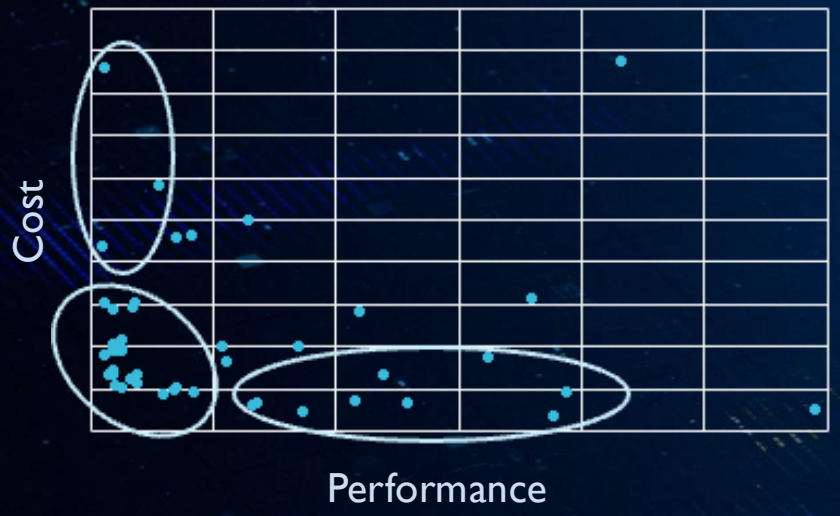
| Topic | Timeline |
|---|---|
| Introduction to System Simulation | 13:45-14:05 |
| Detailed Single Node simulation for ML workloads using gem5 x MLIR | 14:05-15:15 |
| Break | 15:15-15:25 |
| Scalable multi-node simulation using SST<br>▪ *Exploring multi-scale parallelism (MPI+OMP) with Instruction-level Simulation*<br>▪ *Exploring the scaling of LLM training with Packet-level Simulation* | 15:25-16:30 |

ML: Machine Learning MLIR: Multi-Level Intermediate Representation SST: Structural Simulation Toolkit LLM: Large Language Model

# Design Space Exploration

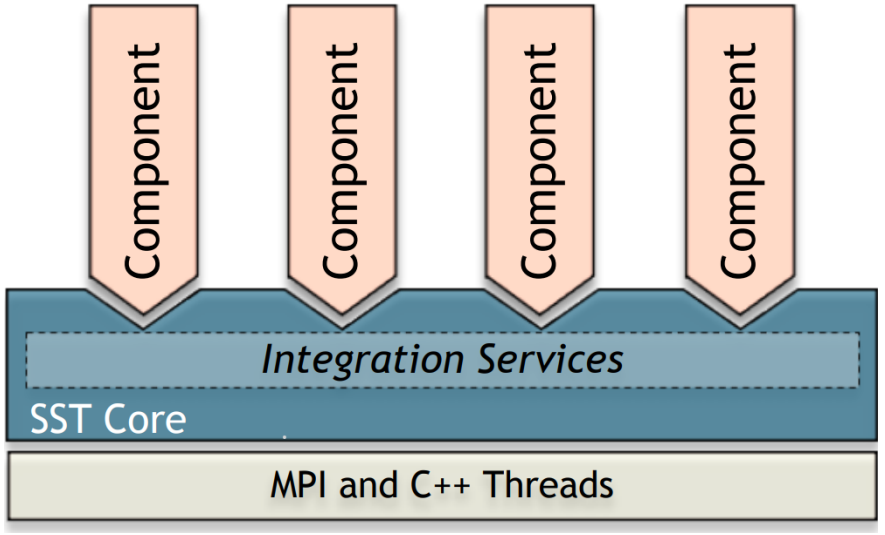
Profile real-life workloads for optimal design choices based on performance vs. PPAC analysis, across parameters from:

- ✓ Microarchitecture
- ✓ Memory and storage
- ✓ Network configuration and topology
- ✓ Many more

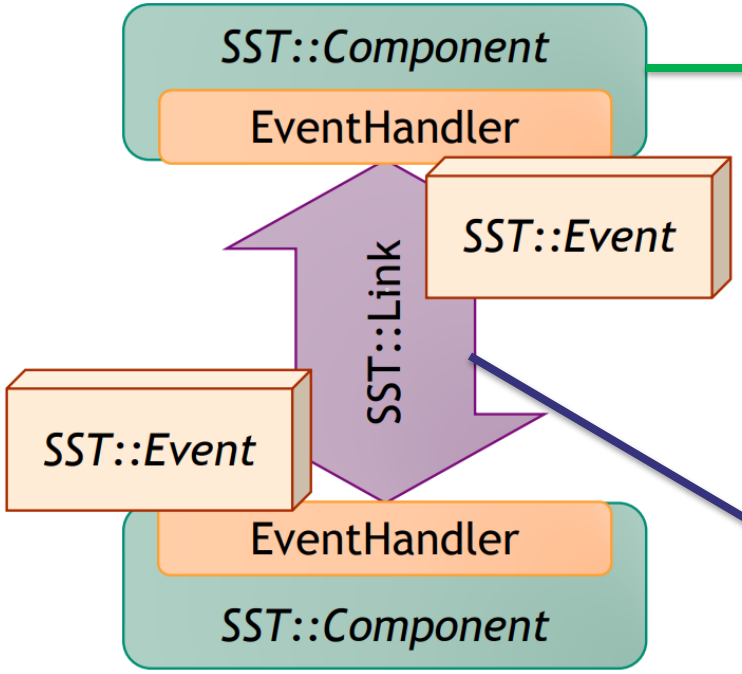



# Introduction to Structural Simulation Toolkit (SST):

- SST is a Parallel discrete-Event Simulator Framework
  - Developed by Sandia National Laboratories
  - Uses MPI for parallelization: **Allows large scale simulation**
- Divided into two projects:
  - **SST-core framework**:
    - The backbone of the simulator
    - Provides utilities and interfaces for components
    - **You don't need to edit it**
  - **SST-Elements Libraries**:
    - Libraries of components that model compute system
    - You can add new components or libraries
    - **More information on** https://sst-simulator.org
- Components are modeled in C++
- Simulations are built with python script



# Introduction to Structural Simulation Toolkit (SST): Building a system




```
### Create the components
component0 = sst.Component("c0", "simpleElementExample.example0")
component1 = sst.Component("c1", "simpleElementExample.example0")
                                        Library.Component


### Parameterize the components.
params = {
        "eventsToSend" : 50,
        "eventSize" : 32
}
component0.addParams(params)
component1.addParams(params)

# Link the components via their 'port' ports
link = sst.Link("component_link")
link.connect( (component0, "port", "1ns"), (component1, "port", "1ns") )
```

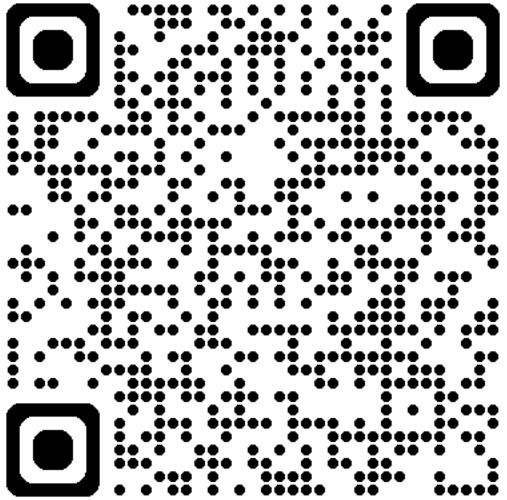
Parameters and port names must match information provided by sst-info

Source: A-SST Initial Specification

Python script snippet

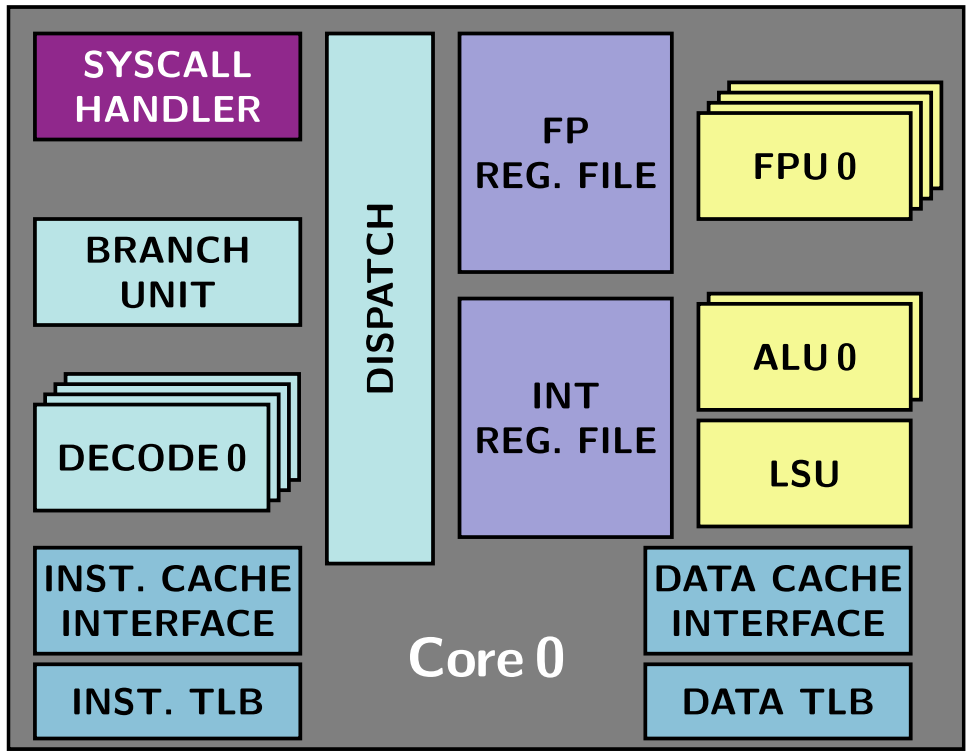
# Workshop contents

- All the materials are available online via github
  - Slides are available here
  - Docker environment is provided
  - Try it out yourself — *During or After the Workshop!*
  - File issues on the github if you need help
  - Instructions to build SST-core and SST-elements are provided in *external/INSTALL.rst*



https://github.com/CSA-infra/RISCV-Scalable-Simulation-tutorial

# Presentation of the system under exploration: Single-node system

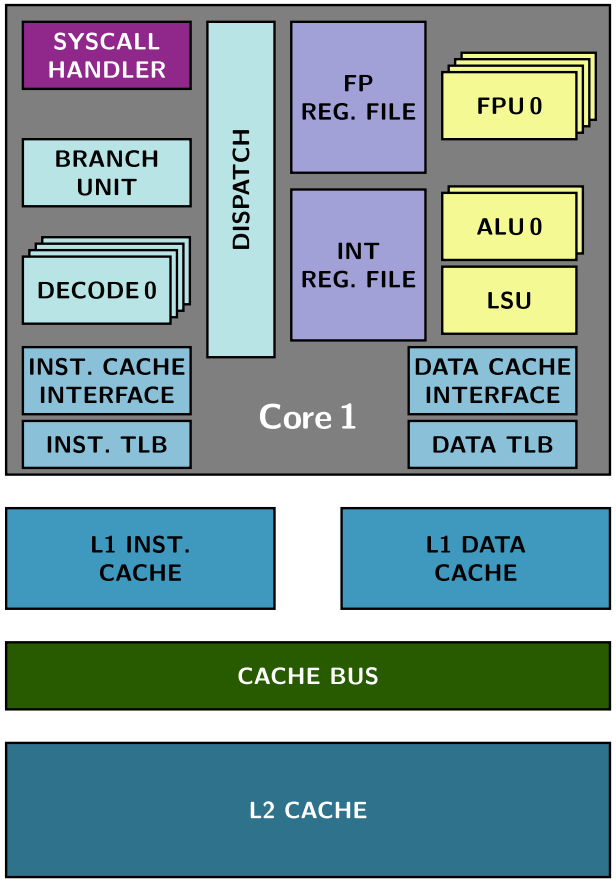



**RISC-V 64-bit CPU core**

The CPU core is simulated with ***Vanadis*** model:

- Vanadis is a processor with **detailed pipeline** model
- The ISA is configurable: MIPS32 or **RISCV64**
- Supports **multi-threading**
- **Does not support vector extensions**
- **Widely configurable:**
  - Register file size
  - Number of instructions issued per cycle
  - Number of ALU and FPU
  - Latencies of ALU and FPU operations
  - Etc ...

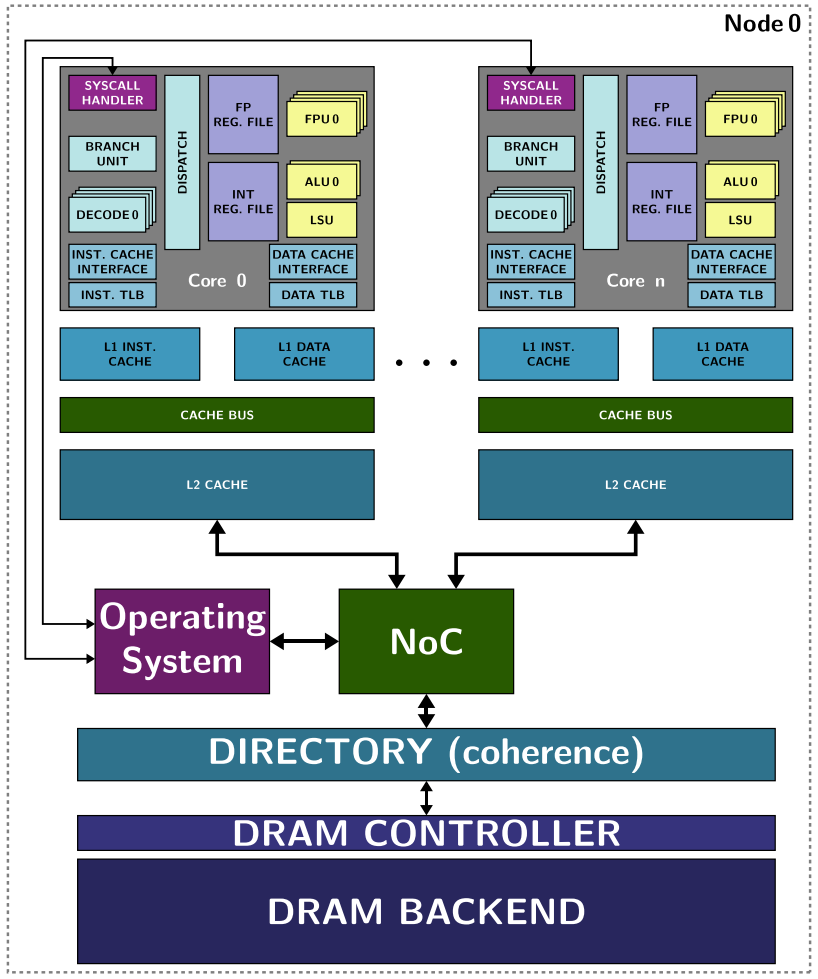
# Presentation of the system under exploration: Single-node system




**CPU complex**

- **Each CPU core is attached to cache subsystem**:
  - Private L1 Instruction cache with a prefetcher
  - Private L1 Data cache with a prefetcher
  - Shared L2 cache (inclusive)
  - MESI cache coherence protocol
  - A memory bus interconnect the caches
- **Every components are defined in memHierarchy**
- **Memory components are configurable**:
  - Capacity (cache & buffers)
  - Latency (ports & caches)
  - Bandwidth (link width and number of requests per cycle)

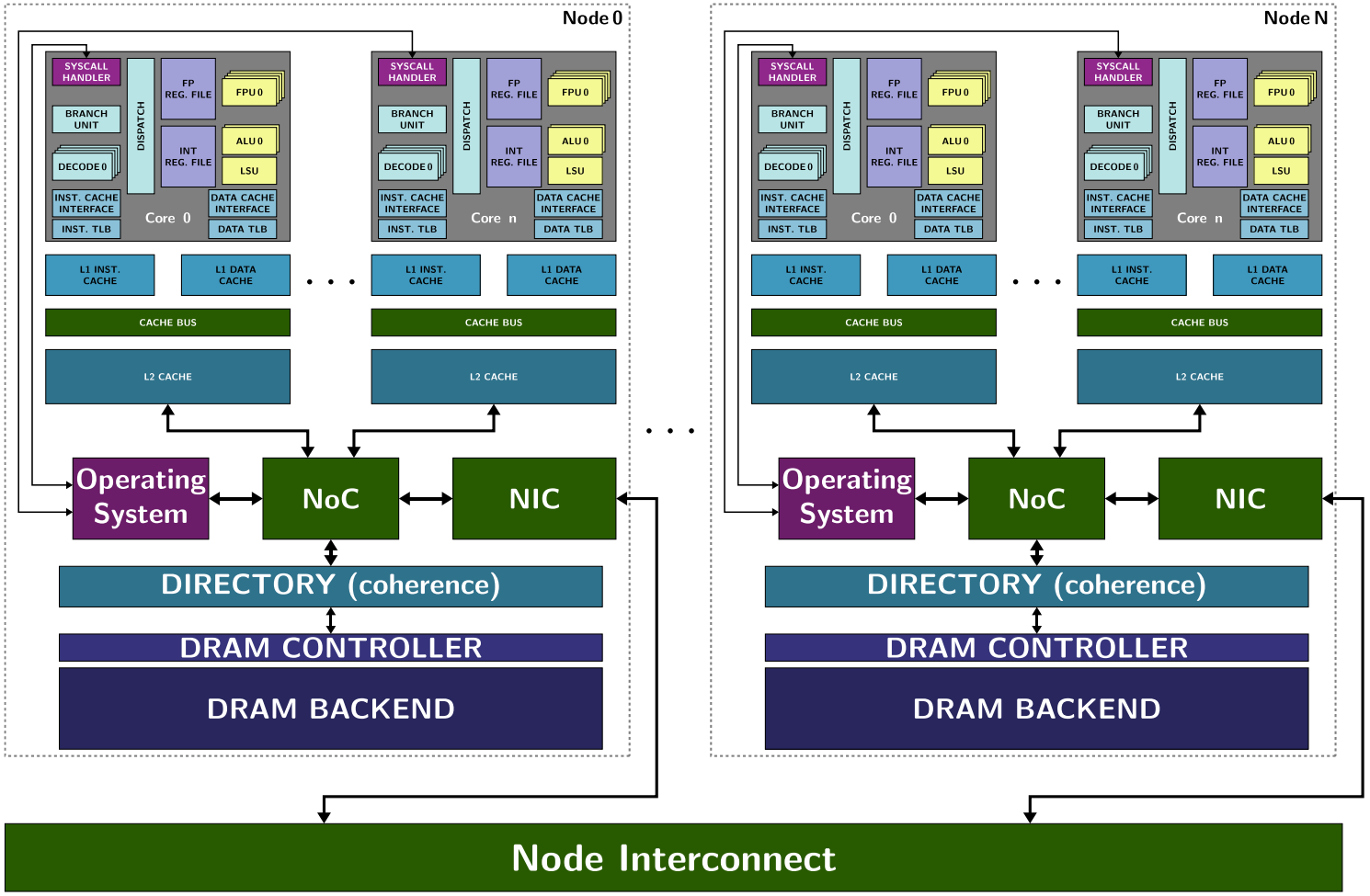
# Presentation of the system under exploration: Single-node system




**A single-node system is built by aggregating multiple CPU complex with a Network-on-Chip:**

- Integrates an **operating system** providing the minimal services to run applications including virtual memory:
  - Emulated with an SST-elements component:
    - Fast syscall emulation
    - Not extendable
- Integrates a **DRAM** attached to **a directory** which manages the coherency:
  - Many backend can be instantiated:
    - Naive memory model applying constant latency (simpleMemory)
    - External DRAM simulator accurately reproducing latencies (Ramulator from CMU-Safari)
- **A node can run multi-threaded applications**
- **Scaling-up compute power by increasing the number of CPU cores**

# Presentation of the system under exploration: Multi-node system

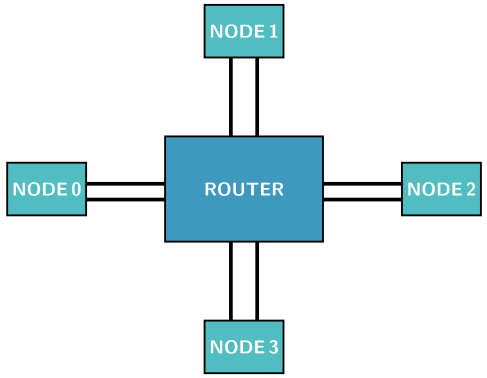



**A Multi-node system is built by aggregating node with a node interconnect:**

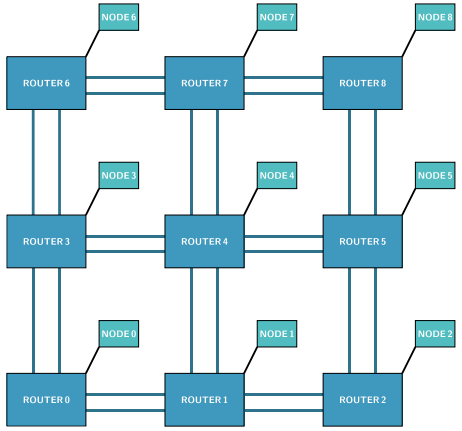
- A NIC is attached to each node:
  - **Allows to run MPI applications**
  - **Requires to edit the MPI library**
- **Many interconnect can be instantiated**:
  - Allows to do studies on network topologies

**Scaling-out compute power by increasing the number of nodes**

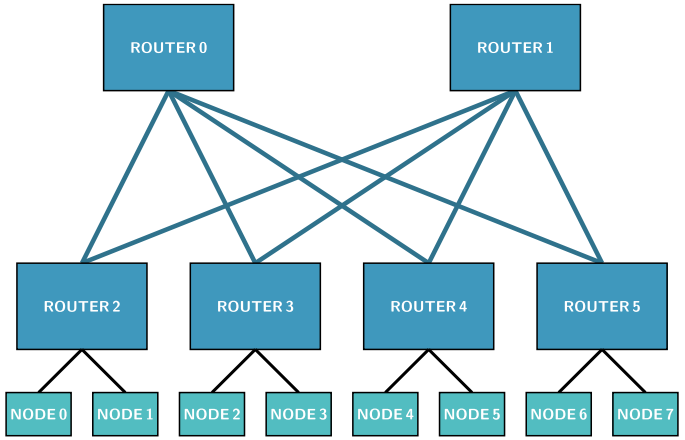
# Presentation of the workload under evaluation: Merlin topologies




**Single router**




**2D Mesh**

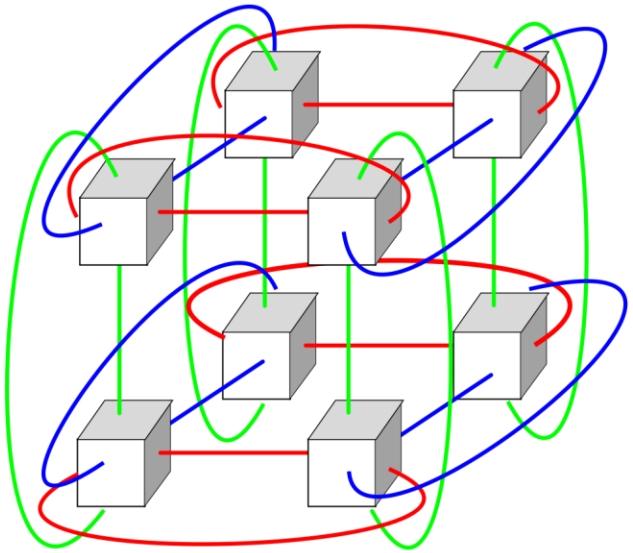



**Fat tree**

**SST Merlin allows to explore many topologies with minimal effort:**

Includes also:

- **Dragon fly topology**
- **Hyper X topology**



**2x2x2 Torus**

(from: Wikipedia/Torus interconnect )

# Workload under evaluation: Multi-Head Attention

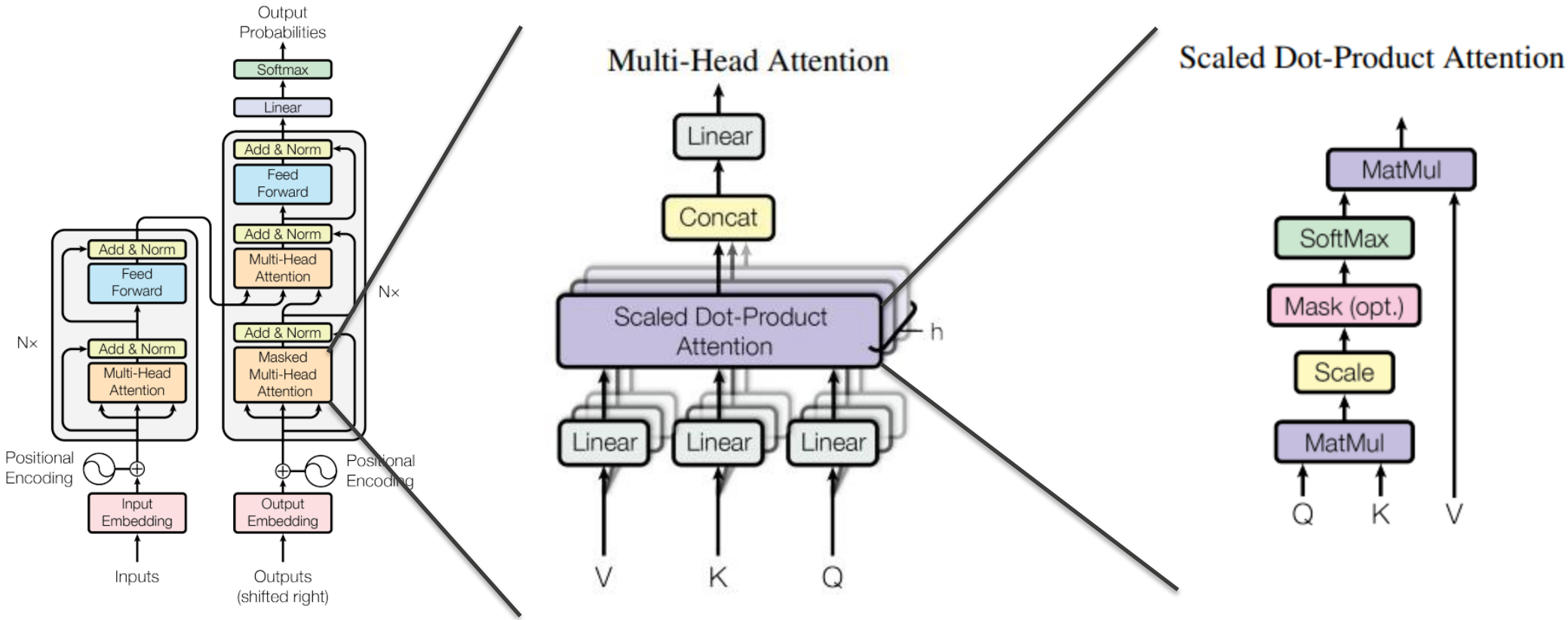
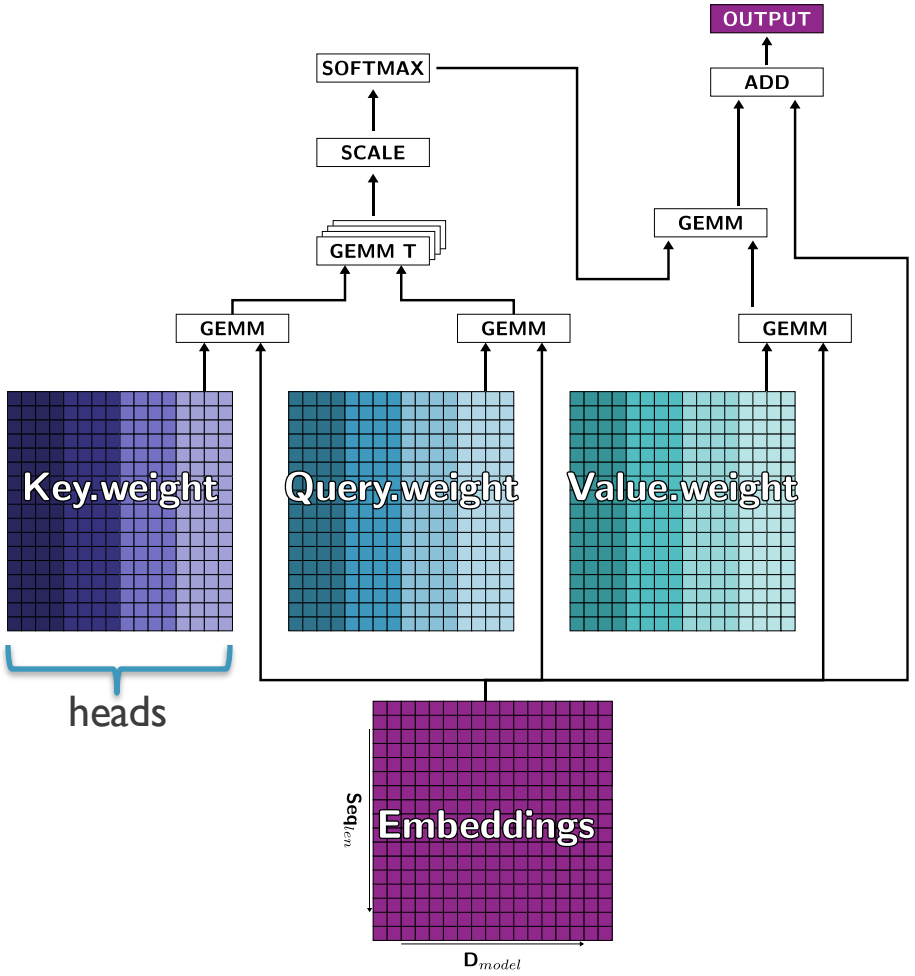



Illustration from: Attention Is All You Need

# Demo 1: Single-node simulation with an OpenMP application




- $Seq_{Len}$, $D_{model}$ and ***heads*** are passed as arguments of the main function and defined the number of FP operations (FLOP):
  - Q,K,V, and ATTNout GEMM: each requires $2 \times Seq_{len} \times D_{model} \times D_{model}$ FLOP
  - KQ and KQV GEMM: each require $2 \times Seq_{len} \times Seq_{len} \times D_{model}$ FLOP
  - The scale kernel requires $heads \times Seq_{len} \times Seq_{len}$ FLOP
  - The softmax kernel requires $7 \times heads \times Seq_{len} \times Seq_{len}$ FLOP
  - The ADD kernel requires $Seq_{len} \times D_{model}$ FLOP
  - **Total:** $\boldsymbol{Seq_{len} \times (D_{model} \times (8 \times D_{model} + 4 \times Seq_{len} + 1) + 8 \times heads \times Seq_{len})}$
- **Each compute kernel is parallelized with OpenMP directives**
- Matrix-Matrix multiplication is the heaviest workload:
  - **Tiled implementation** to leverage cache hierarchy and minimize data movement

# Demo 1: Single-node simulation with an OpenMP application

## 1st experiment: GEMM tile size

- 4 binaries are provided for this experiment:
  - mha_OMP_8 which uses **8x8 tiles**
  - mha_OMP_16 which uses **16x16 tiles**
  - mha_OMP_32 which uses **32x32 tiles**
  - mha_OMP_64 which uses **64x64 tiles**

- Run the two binaries and analyze the performance:
  - Run `**sst scale_up.py**` in *demo/sst/system* folder
  - You need to pass the correct **exe (--exe path)** as argument of the script

- **What can explain the difference?**
  - You can use the generated statistics to verify the explanation

## 2nd experiment: simulation scaling

Let's explore the scalability of the **simulated system** and of the **simulation**:

- Make sure to use **the most efficient binary**

- You can vary the number of simulated **threads** and **processors**
  - **--num_threads_per_cpu NUM**
  - **--num_cpu_per_node NUM**

- You can vary the number of simulation threads (i.e. parallelism of the simulation) using --num-threads= option:
  - **sst --num-threads=4 scale_up.py**

# Demo 1: Single-node simulation with an OpenMP application

**1st experiment: GEMM tile size**

| | SIMULATED TIME | | MHA TIME | | Number of instructions issued | | | Number of loads executed | | Number of branches | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| TILE SIZE | ms | Normalized | ms | Normalized | Millions of instructions | Normalized | Balance | Millions of loads | Normalized | Millions of branches | Normalized |
| **8x8** | 10.83 | 1.11 | 10.31 | 1.12 | 55.23 | 1.22 | 50%/50% | 10.92 | 1.06 | 6.58 | 1.11 |
| **16x16** | 10.08 | 1.03 | 9.57 | 1.04 | 48.30 | 1.07 | 50%/50% | 10.50 | 1.02 | 6.12 | 1.04 |
| **32x32** | 9.76 | 1.00 | 9.24 | 1.00 | 45.21 | 1.00 | 50%/50% | 10.34 | 1.00 | 5.91 | 1.00 |
| **64x64** | 12.86 | 1.32 | 12.34 | 1.34 | 54.61 | 1.21 | 54%/46% | 12.44 | 1.20 | 10.15 | 1.72 |

**Experiment with 2 CPU and 2 threads each. $Seq_{len}$ = 64, $D_{model}$=128, heads = 8.**

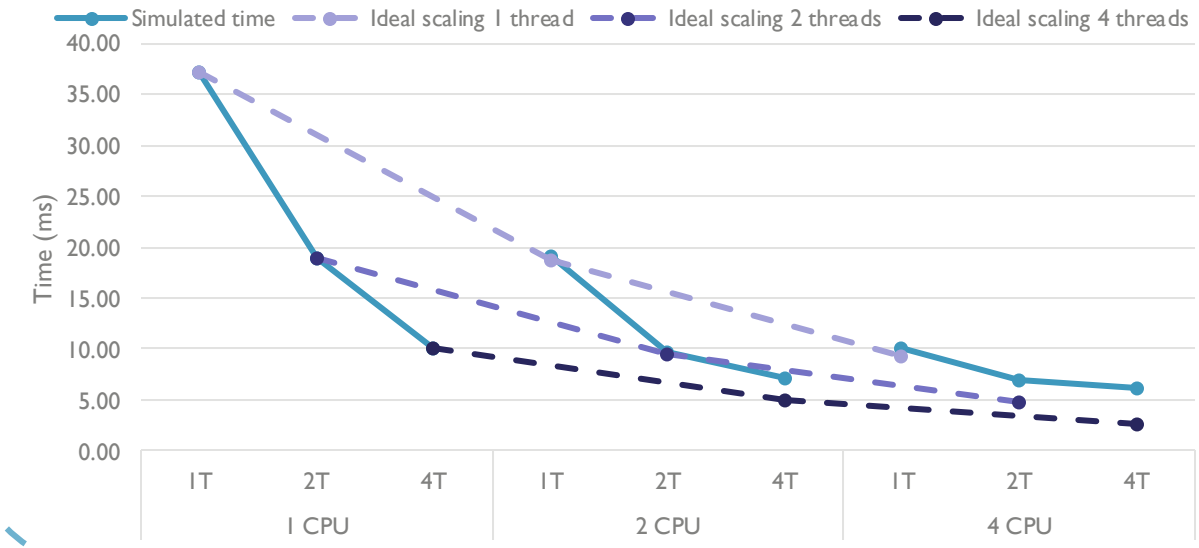
- Larger tiles improve the performance by increasing the arithmetic intensity (less loads and branches)
- Too large tiles lead to workload imbalance

# Demo 1: Single-node simulation with an OpenMP application

## 2nd experiment: simulation scaling

| | 1 CPU | | | 2 CPU | | | 4 CPU | | |
|---|---|---|---|---|---|---|---|---|---|
| **Number of simulated threads** | 1T | 2T | 4T | 1T | 2T | 4T | 1T | 2T | 4T |
| **Simulated time (ms)** | 37.26 | 18.84 | 10.13 | 19.21 | 9.76 | 7.06 | 10.09 | 6.88 | 6.14 |

**Scaling of the simulated system with $Seq_{len}$ = 64, $D_{model}$=128, heads = 8.**



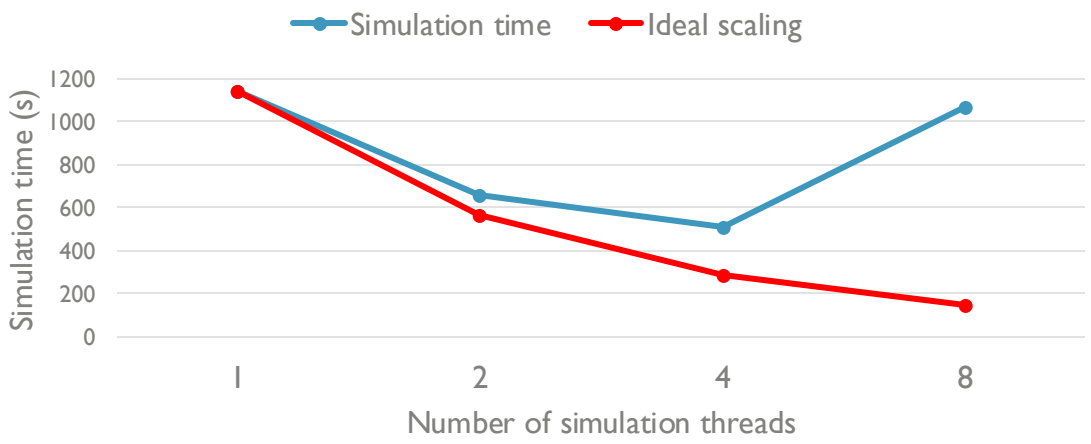

- Good scaling when the number of CPU increases with 1 thread:
  - Parallel cache subsystems
- Good scaling when the number of threads increases with 1 CPU:
  - Data shared from the same cache subsystem
- Poor scaling when the number of CPU increases with 4 threads:
  - Data shared from different cache subsystem
  - Not enough work for 16 SW threads

# Demo 1: Single-node simulation with an OpenMP application

## 2nd experiment: simulation scaling

| Number of simulation threads | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Simulation time (s) | 1140 | 655 | 510 | 1066 |
| Million of instr. per second | 0.08 | 0.13 | 0.17 | 0.08 |

**Scaling of the simulation with 4 CPU, 4 threads, $Seq_{len}$ = 64, $D_{model}$=128, heads = 8.**




**The simulation fails to scale for the same reasons as the simulated system!**

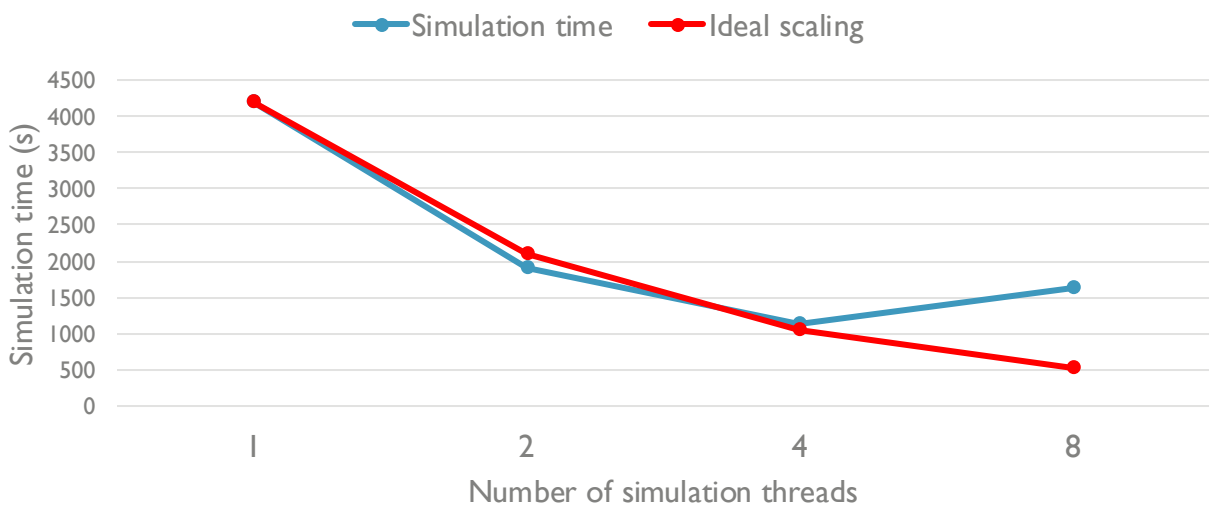
- Amdahl's Law applies for the simulation framework as well:
  - We can speed up the simulation of parallel components (CPU complex)
  - We cannot speed up the simulation of shared resources (NoC, DRAM, OS)
- Allocating too much threads unbalances the simulation

# Demo 1: Single-node simulation with an OpenMP application

## 2nd experiment: simulation scaling

| Number of simulation threads | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Simulation time (s) | 4196 | 1909 | 1133 | 1639 |

**Scaling of the simulation with 8 CPU, 1 thread, $Seq_{len}$ = 128, $D_{model}$=128, heads = 16.**




**Simulation scales better with more parallelism!**

- More CPU but same number of threads
- Larger workload

Allocating too much threads unbalances the simulation:

- Each thread simulates a set of components
- Better to overload the simulation threads (i.e., less threads than number of simulated CPU)

# Demo 2: Multi-node simulation with a hybrid OpenMP + MPI application

## 1st experiment: Network topologies

To get start with scale-out simulation we will experiment different inter-node network topology:

- 3 types of topology are defined in ***scale_out.py***:
  - **Simple**: instantiates ***num_node_per_router*** and interconnect them with a single router
  - **Torus:** instantiates a torus network
    - ***torus_width:*** Defines the number of links between two routers
    - ***torus_shape:*** Defines the shape of the network. Each item defines the number of routers in one dimension
  - **Fat tree:** instantiates a Fat tree network
    - ***Fattree_shape:*** defines the shape of the network.
      - Syntax is *"number of router in a level : number of port per router"*

```
fattree_shape = "1,1:2,2"
```

- You can try to run a simulation with each topology

# Demo 2: Multi-node simulation with a hybrid OpenMP + MPI application

**2nd experiment: simulation scaling**

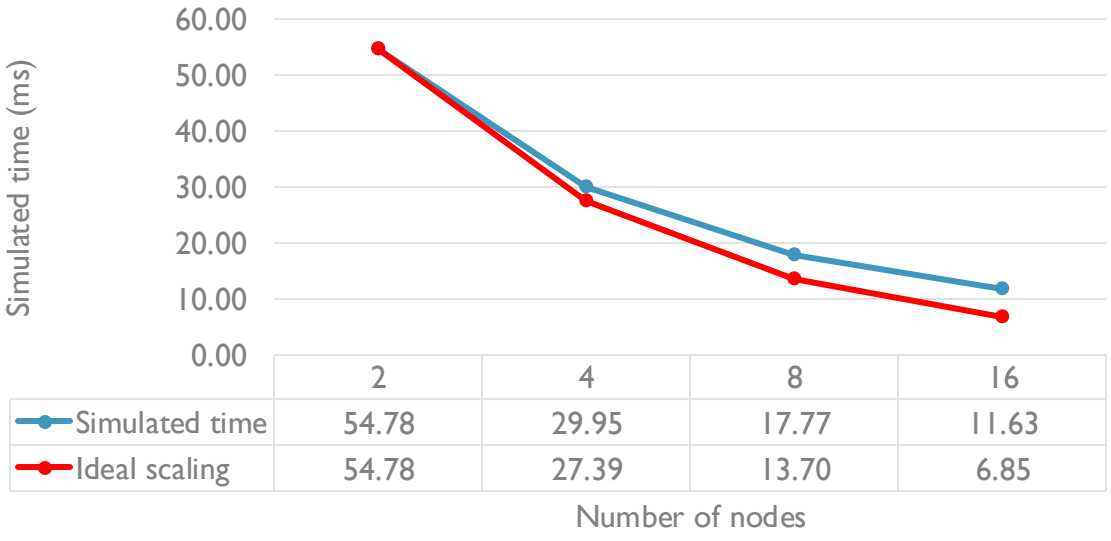
Let's explore the scalability of the **simulated system** and of the **simulation**:

- You can increase the compute power by increasing the number of nodes (e.g., 2, 4, 8). You can select any of the topologies

- You can parallelize the simulation by using multiple MPI ranks. **Make sure to not oversubscribe your computer** (i.e., no more process than number of core).
  - **mpirun --np 4 scale_out.py**

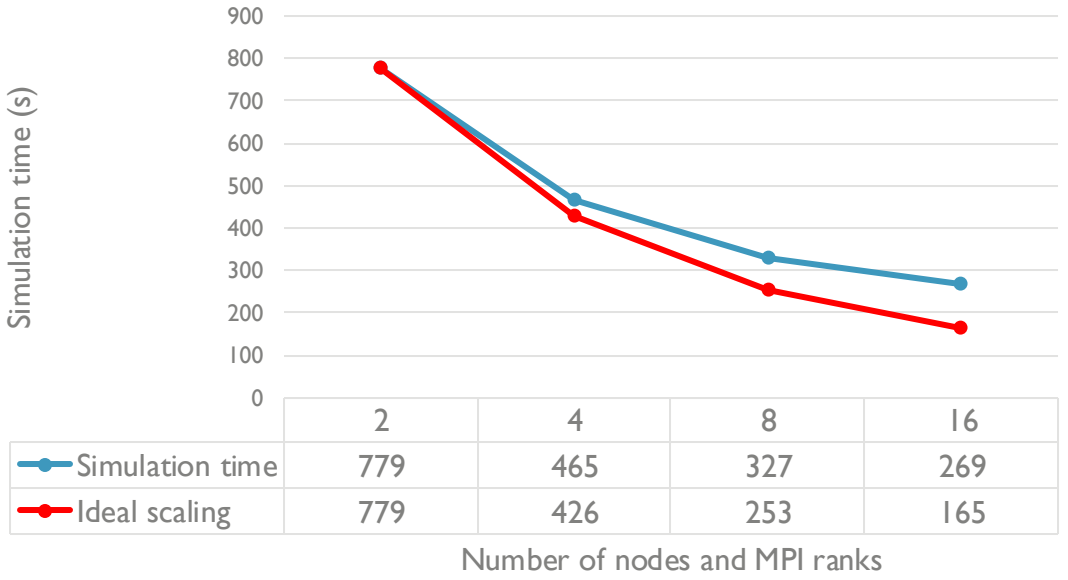
# Demo 2: Multi-node simulation with a hybrid OpenMP + MPI application

## 2nd experiment: simulation scaling




| | 2 | 4 | 8 | 16 |
|---|---|---|---|---|
| Simulated time | 54.78 | 29.95 | 17.77 | 11.63 |
| Ideal scaling | 54.78 | 27.39 | 13.70 | 6.85 |

Scaling of the simulated system.
Each node integrates 1 CPU with 1 thread.
Seqlen = 128, Dmodel,=128, heads = 16.




| | 2 | 4 | 8 | 16 |
|---|---|---|---|---|
| Simulation time | 779 | 465 | 327 | 269 |
| Ideal scaling | 779 | 426 | 253 | 165 |

Scaling of the simulation framework.
The number of MPI ranks used matches the number of simulated node.

Ideally the simulation time would decrease with the simulated time.

# Agenda

| Topic | Timeline |
|---|---|
| Introduction to System Simulation | 13:45-14:05 |
| Detailed Single Node simulation for ML workloads using gem5 x MLIR | 14:05-15:15 |
| Break | 15:15-15:25 |
| Scalable multi-node simulation using SST<br>▪ *Exploring multi-scale parallelism (MPI+OMP) with Instruction-level Simulation*<br>▪ *Exploring the scaling of LLM training with Packet-level Simulation* | 15:25-16:30 |

ML: Machine Learning MLIR: Multi-Level Intermediate Representation SST: Structural Simulation Toolkit LLM: Large Language Model

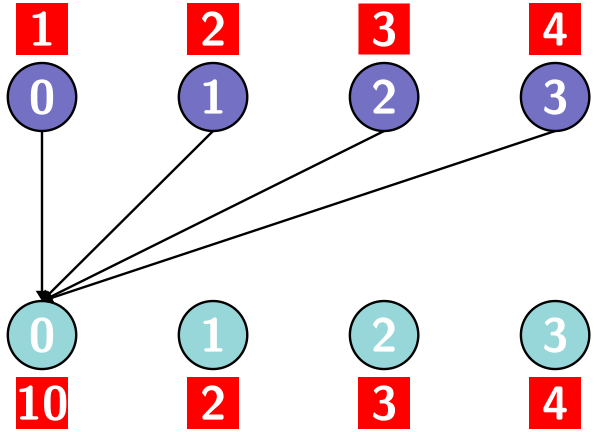
# Exploring the scaling of LLM training with Packet-level Simulation

- Packet-level simulation with SST/Ember
- Background on LLM training parallelism
- Exploring scaling of LLM training

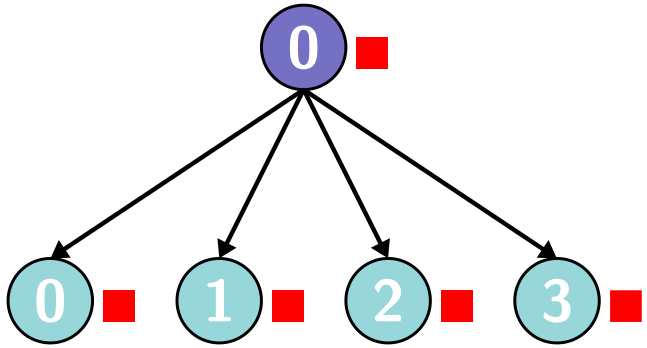
# Background on Parallel Programming on Distributed Memory System

## Message Passing Interface (MPI) collective communication

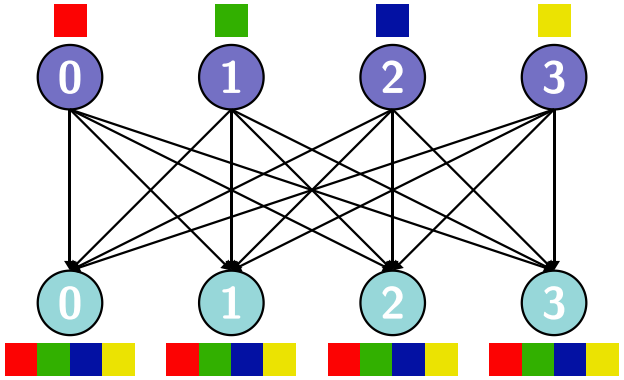

Send-Recv communication



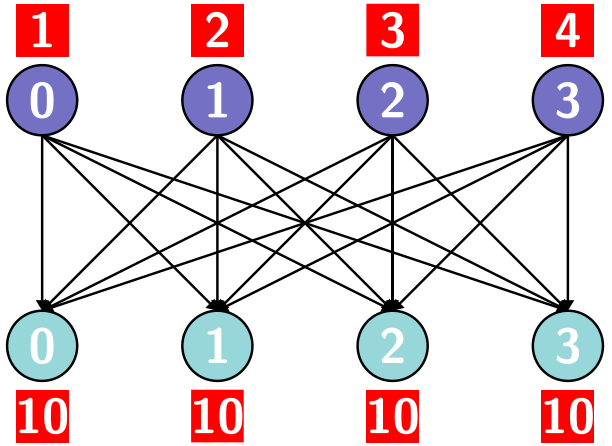

Reduce (sum) communication




Broadcast (bcast) communication

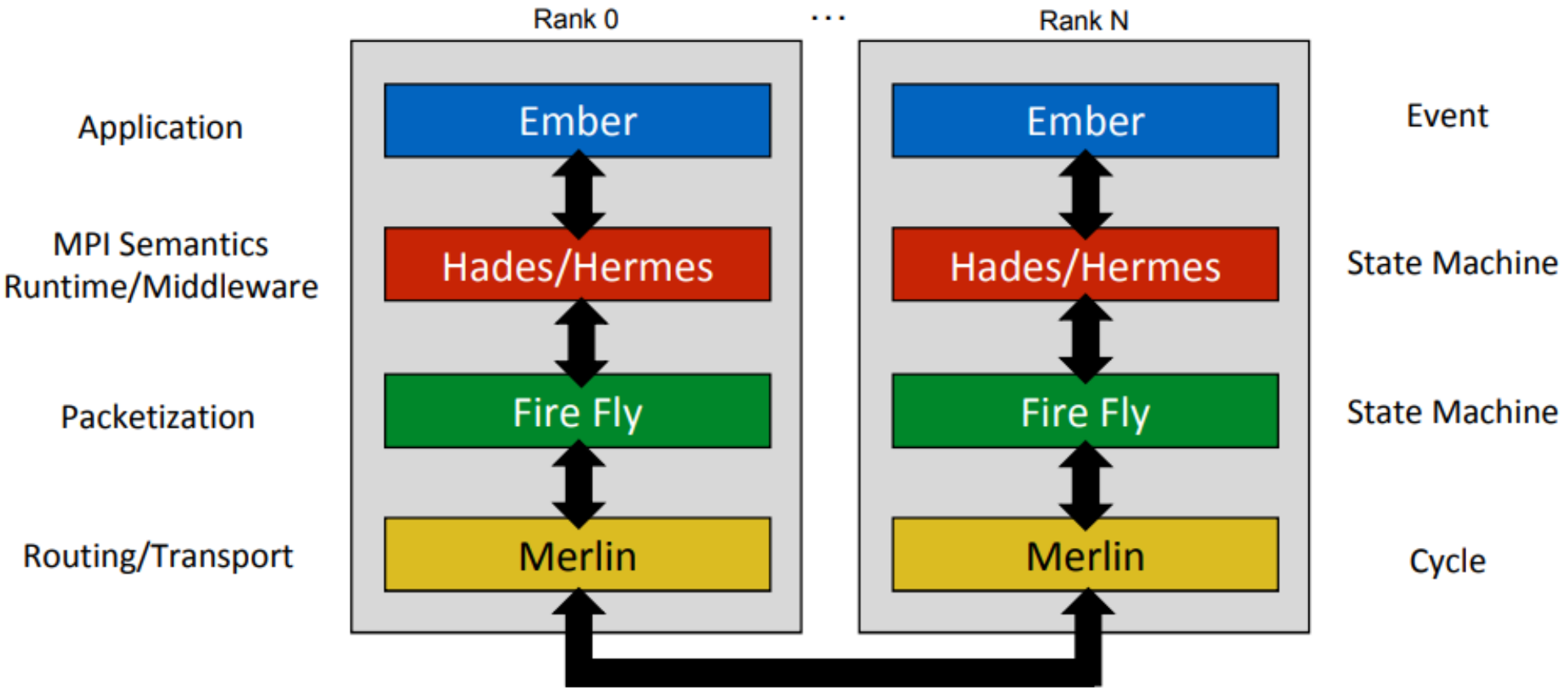



All gather communication




All reduce (sum) communication

# Packet-level simulation with SST/Ember




From:"Ember: Reference Communication Patterns for Exascale. S.D. Hammond et al." (osti.gov/servlets/purl/1307276)

# Packet-level simulation with SST/Ember

More complex motifs can be implemented by using a finite state machine

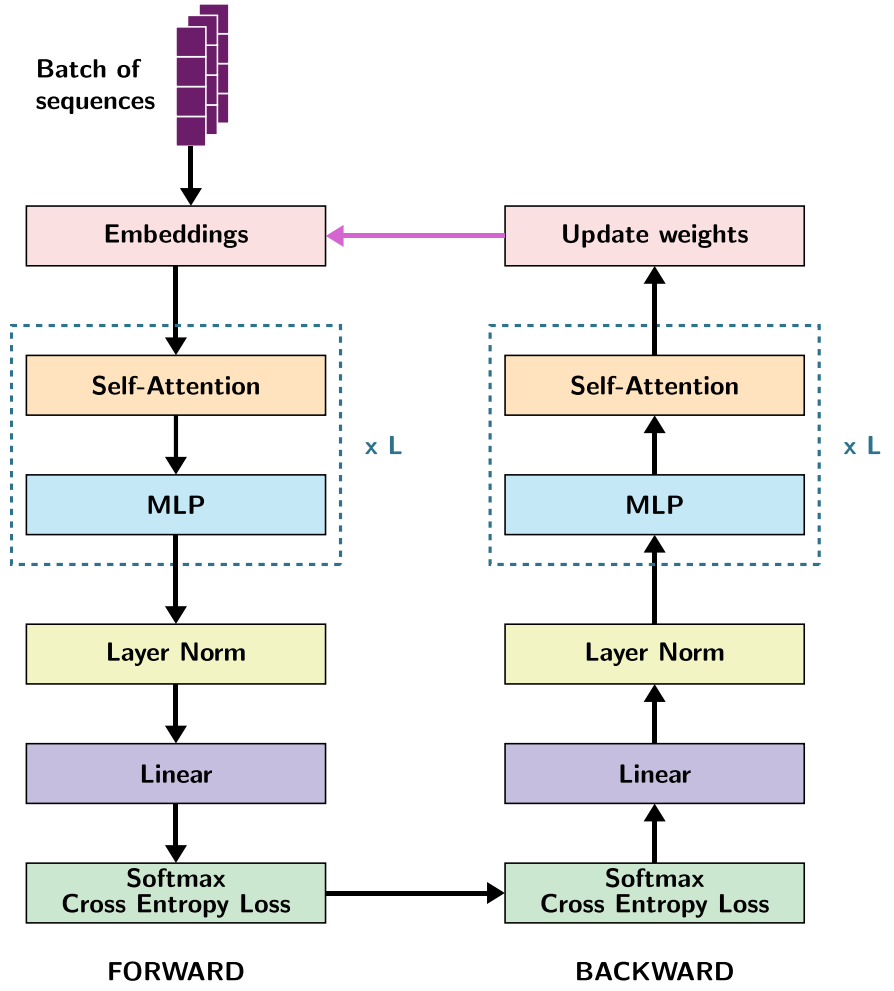
```
bool EmberDummyGenerator::generate(queue<EmberEvent*>& evQ) {
   bool finish = false;

   sendBuf = memAlloc(sizeof(double)*count);
   recvBuf = memAlloc(sizeof(double)*count);

   enQ_compute( evQ, 1e6 );
   enQ_allreduce(evQ, sendBuf, recvBuf, count, DOUBLE, Hermes::MP::SUM, GroupWorld);

   loop_index++;

   if(loop_index == num_iter)
      finish = true;

   return finish;
}
```

- Generates *num_iter* times the same motif ➔generation stops when the function return true.
- After 1ms (1e6 ns) of compute time ➔ *all reduce* communication is scheduled.
- When *all reduce* communication is completed ➔next compute phase is scheduled.

# Background on training parallelism

## Decoder-only transformers

- The input is a batch of token sequences ➔ converted into embeddings tensor by the embeddings layer
- Embeddings tensor crosses L hidden layers includes
  - Self-Attention block
  - Multilayer perceptron (MLP)
- Embeddings tensor is processed by a normalization function ➔ projected onto the vocabulary (linear).
- The loss is calculated with a SoftMax and a cross-entropy function.
- Finally, the loss is backpropagated to update the weights



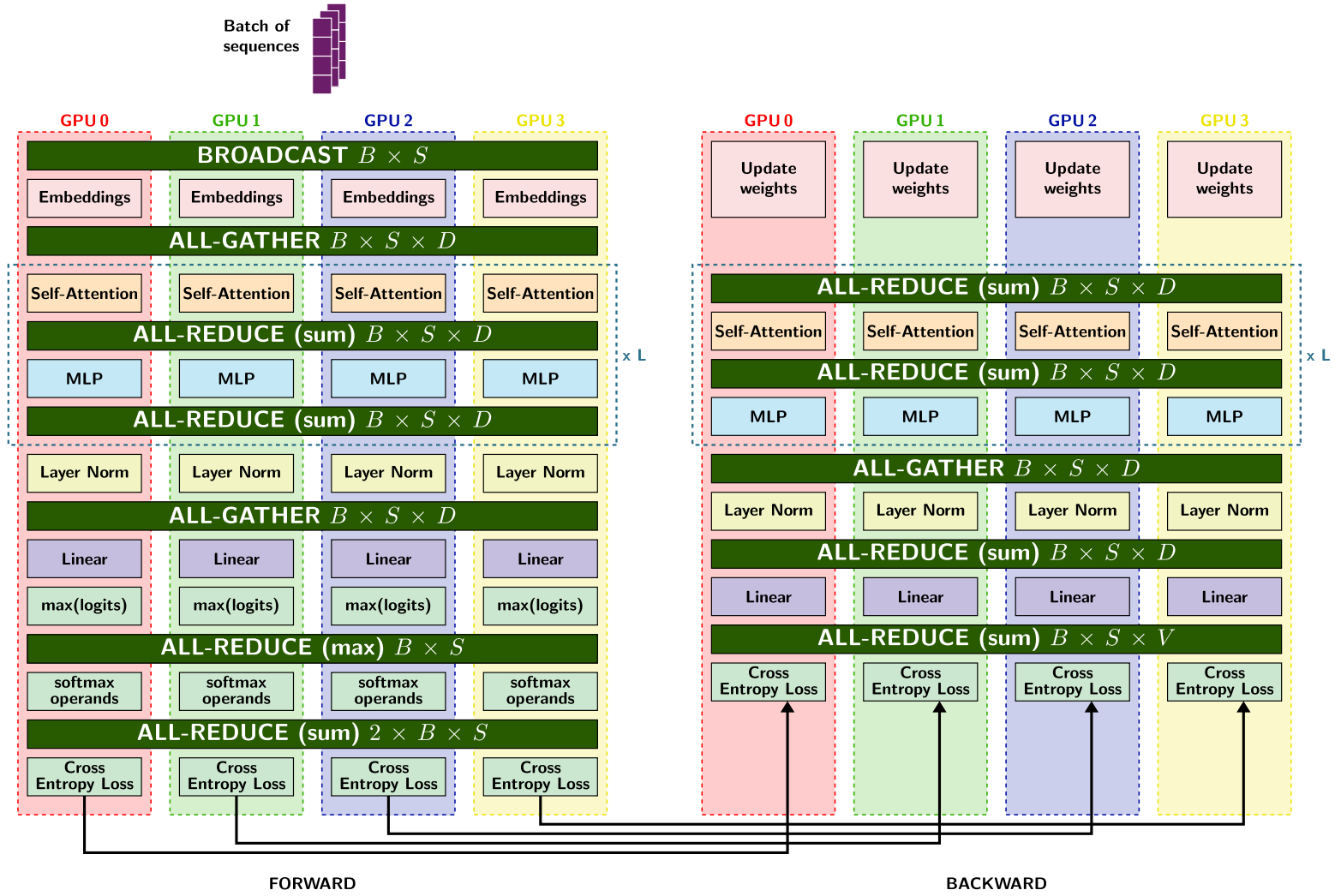

Architecture overview of decoder-only transformers

# Background on training parallelism

## Tensor parallelism

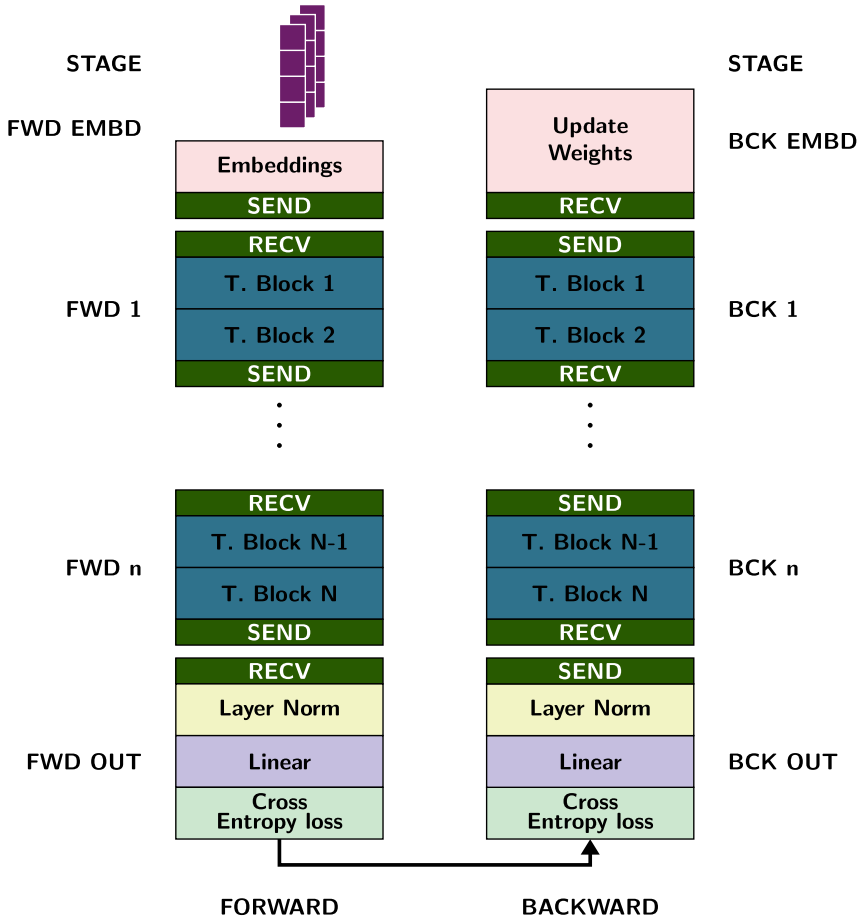
| Symbol | Name | Typical value |
|---|---|---|
| B | Batch size | 32 |
| S | Sequence length (# tokens per sequence) | 8192 |
| D | Model dimension (hidden size) | 8192 |
| V | Vocabulary size | 128256 |

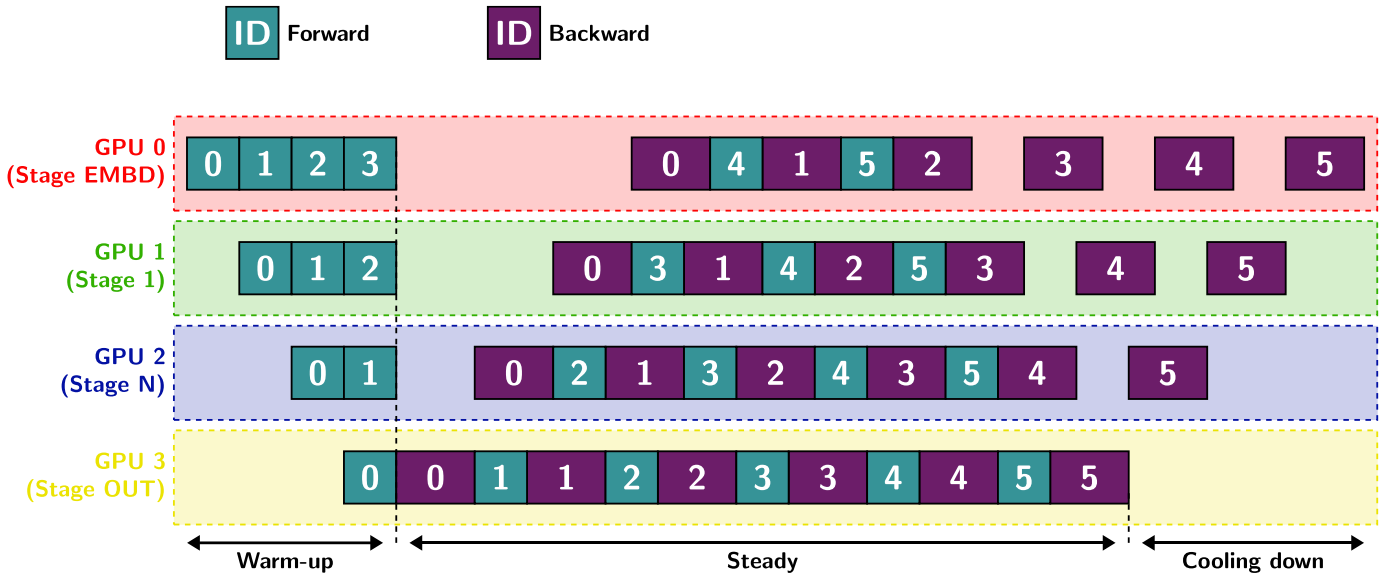
**Weight matrices are split and distributed across the GPUs**



# Background on training parallelism

## Pipeline parallelism




one-forward-one-backward (1F1B) pipeline parallelism

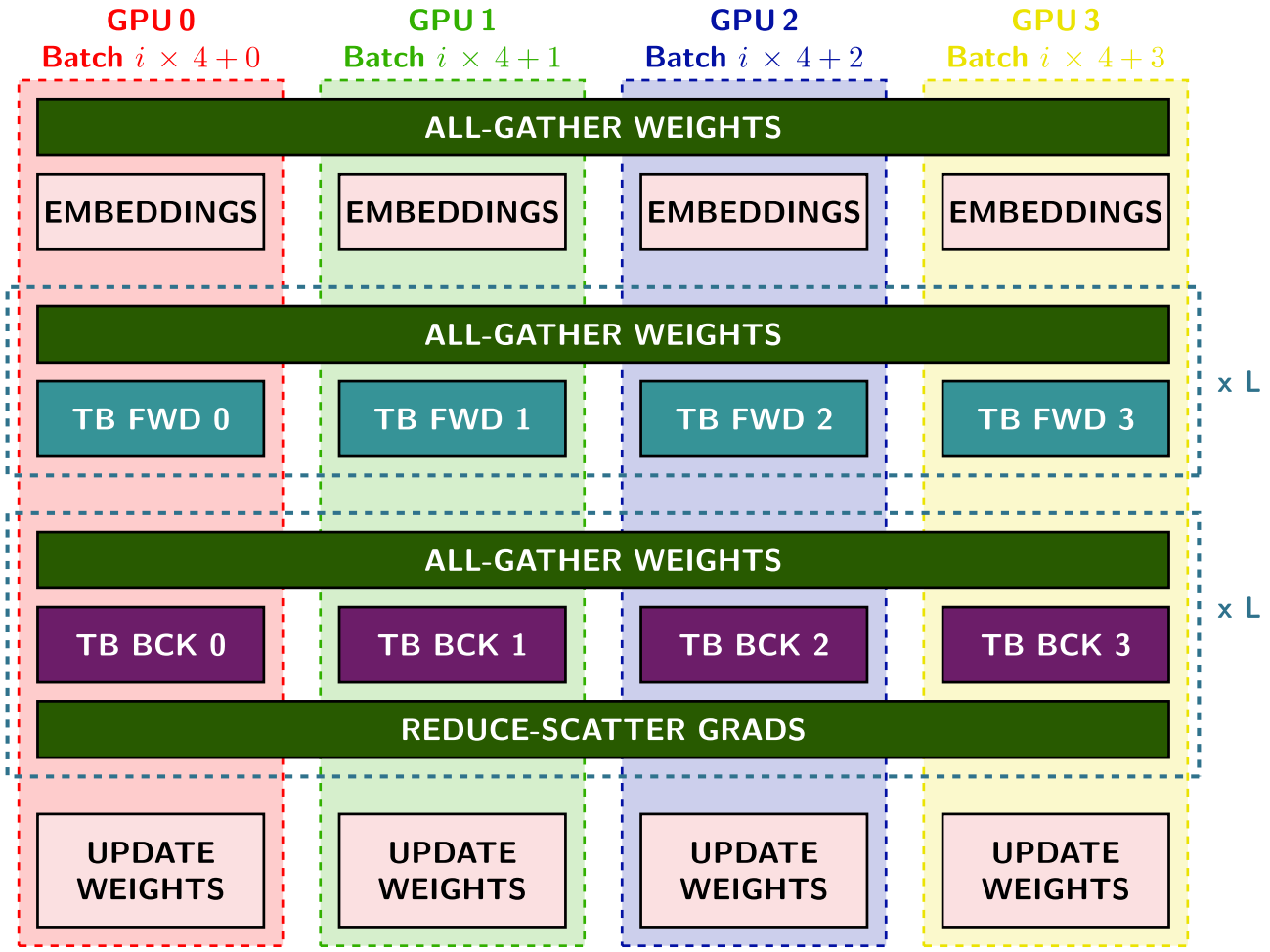
**The execution is split into stages which are mapped to a GPU:**

- The same mapping must be applied for the forward and backward path
- In this tutorial, we assume that 1 GPU is dedicated to embeddings stages and another one to the output stage
- The Transformer blocks are distributed across the remaining GPUs
- Requires at least 3 GPUs

# Background on training parallelism

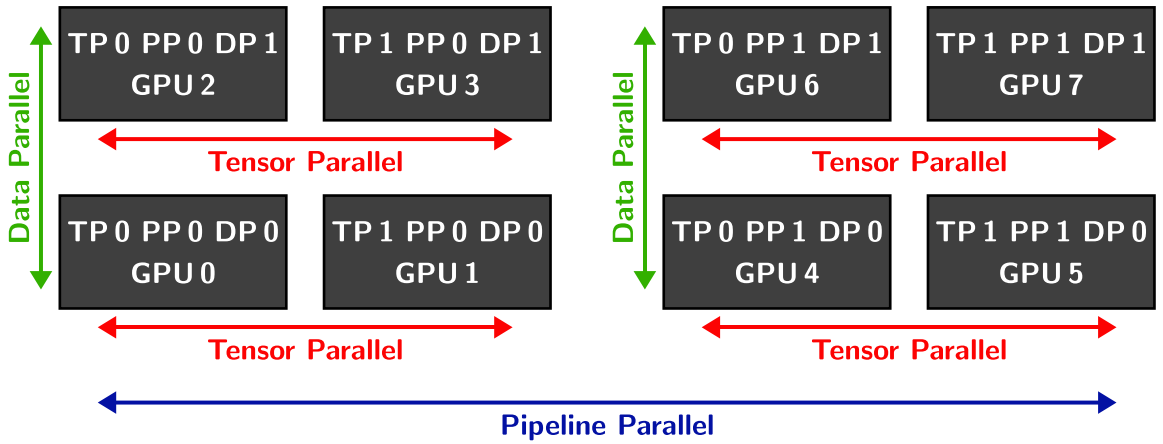
## Data parallelism

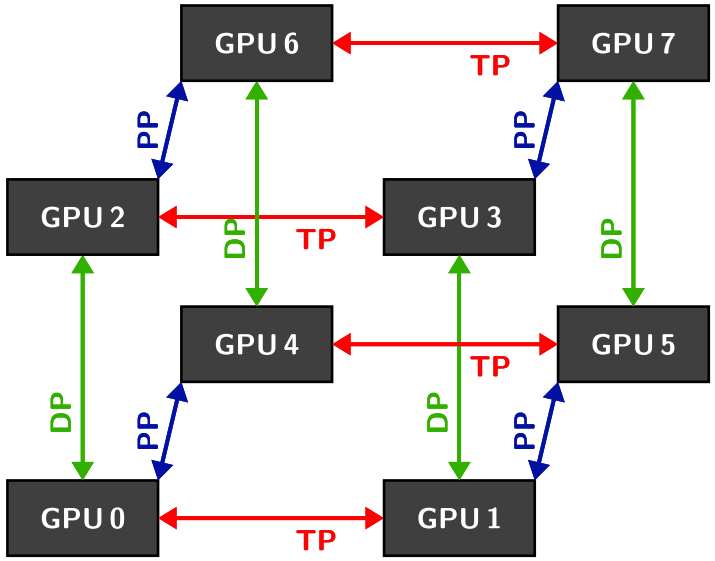
- **The Weights and gradients are distributed over the GPUs:**
- Each GPU processes a different batch
- Each GPU stores a subset of the weights
- Before executing a new transformer block, all GPU share their subset to acquire all the weights pertaining to the block
- In the backward path, after executing a block, each GPU performs a reduce to acquire a subset of the distributed gradients



# Background on training parallelism

## 3D parallelism




**3D parallelism combines Tensor Parallelism, Pipeline Parallelism and Data Parallelism:**

- GPU stores smaller subset of the weights:
  - GPU 0, 1, 2 & 3 store each only ¼ of the embedding weights and need to acquire only half of the weights
- For Data parallelism GPU 0 communicates only with GPU 2
- For Tensor parallelism GPU 0 communicates only with GPU 1
- For Pipeline parallelism GPU 0 communicates only with GPU 4

# Exploring scaling of LLM training

## Tensor parallelism

**Let's explore the scalability of tensor parallelism by sweeping the level of parallelism:**

Command to run:

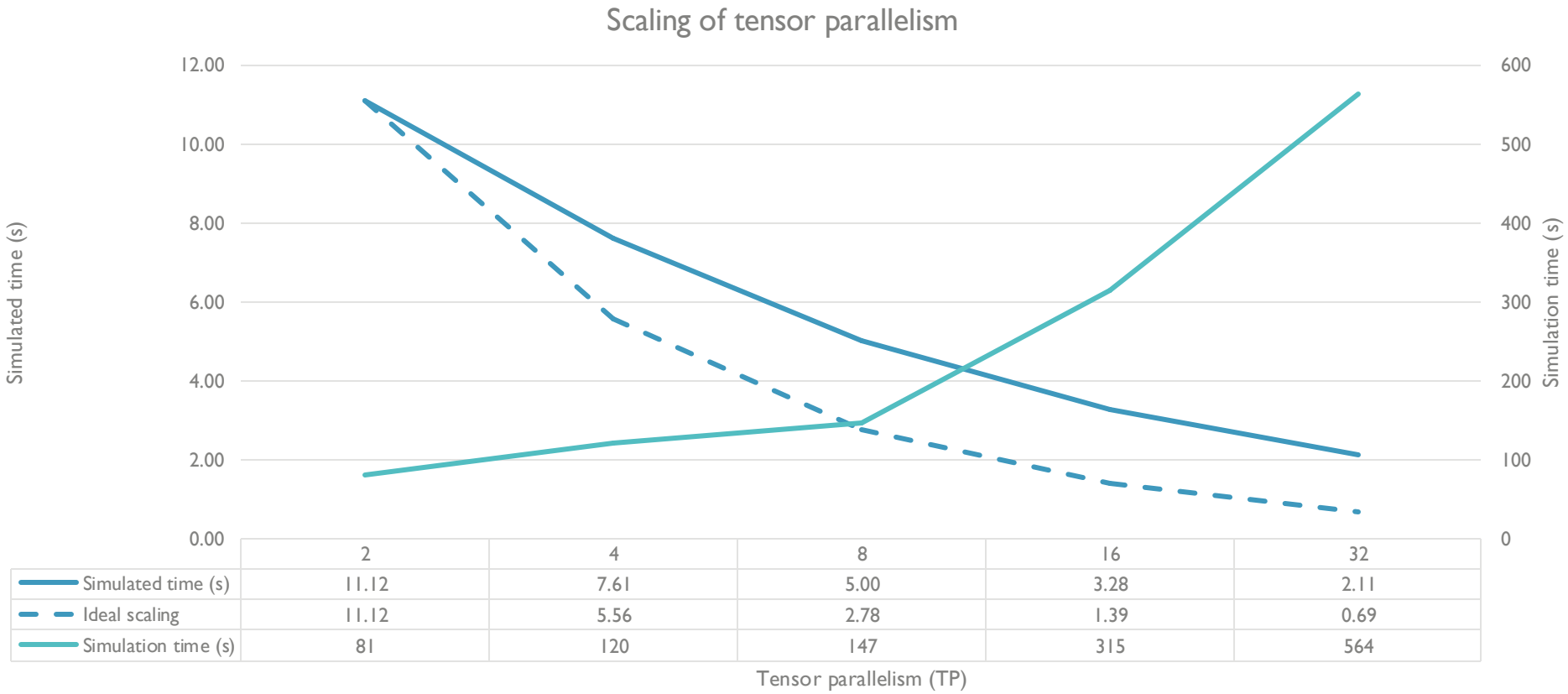
*sst --print-timing-info training_llm.py -- --tp **TP** --pp 1 --dp 1 --batch_size 16 --sequence_len 1024 --n_batch 8 --llm_config small_config.json --log logger --stats stats.csv --topology single --verbose 10*

You can fill the table below:

| Tensor Parallelism (TP) | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|
| Simulated time (s) | | | | | |
| Simulation time(s) | | | | | |

# Exploring scaling of LLM training

## Tensor parallelism




| | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|
| Simulated time (s) | 11.12 | 7.61 | 5.00 | 3.28 | 2.11 |
| Ideal scaling | 11.12 | 5.56 | 2.78 | 1.39 | 0.69 |
| Simulation time (s) | 81 | 120 | 147 | 315 | 564 |

# Exploring scaling of LLM training

## Pipeline parallelism

**Let's explore the scalability of pipeline parallelism by swiping the level of parallelism:**

Command to run:

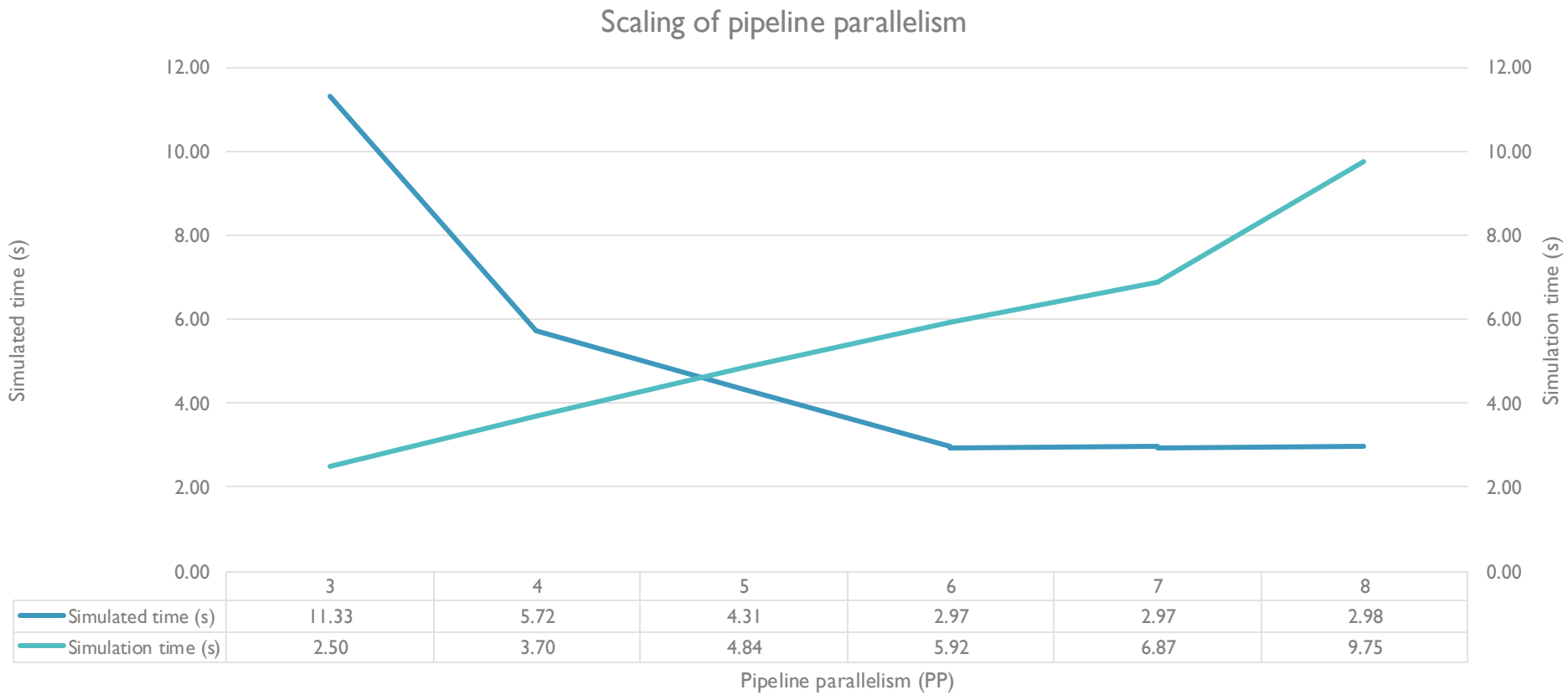
*sst --print-timing-info training_llm.py -- --tp 1 --pp **PP** --dp 1 --batch_size 16 --sequence_len 1024 --n_batch 8 --llm_config small_config.json --log logger --stats stats.csv --topology single --verbose 10*

You can fill the table below:

| Pipeline Parallelism (PP) | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|
| Simulated time (s) | | | | | | |
| Simulation time(s) | | | | | | |

# Exploring scaling of LLM training

## Pipeline parallelism




| | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|
| Simulated time (s) | 11.33 | 5.72 | 4.31 | 2.97 | 2.97 | 2.98 |
| Simulation time (s) | 2.50 | 3.70 | 4.84 | 5.92 | 6.87 | 9.75 |

# Exploring scaling of LLM training

## Data parallelism

**Let's explore the scalability of data parallelism by swiping the level of parallelism:**

Command to run:

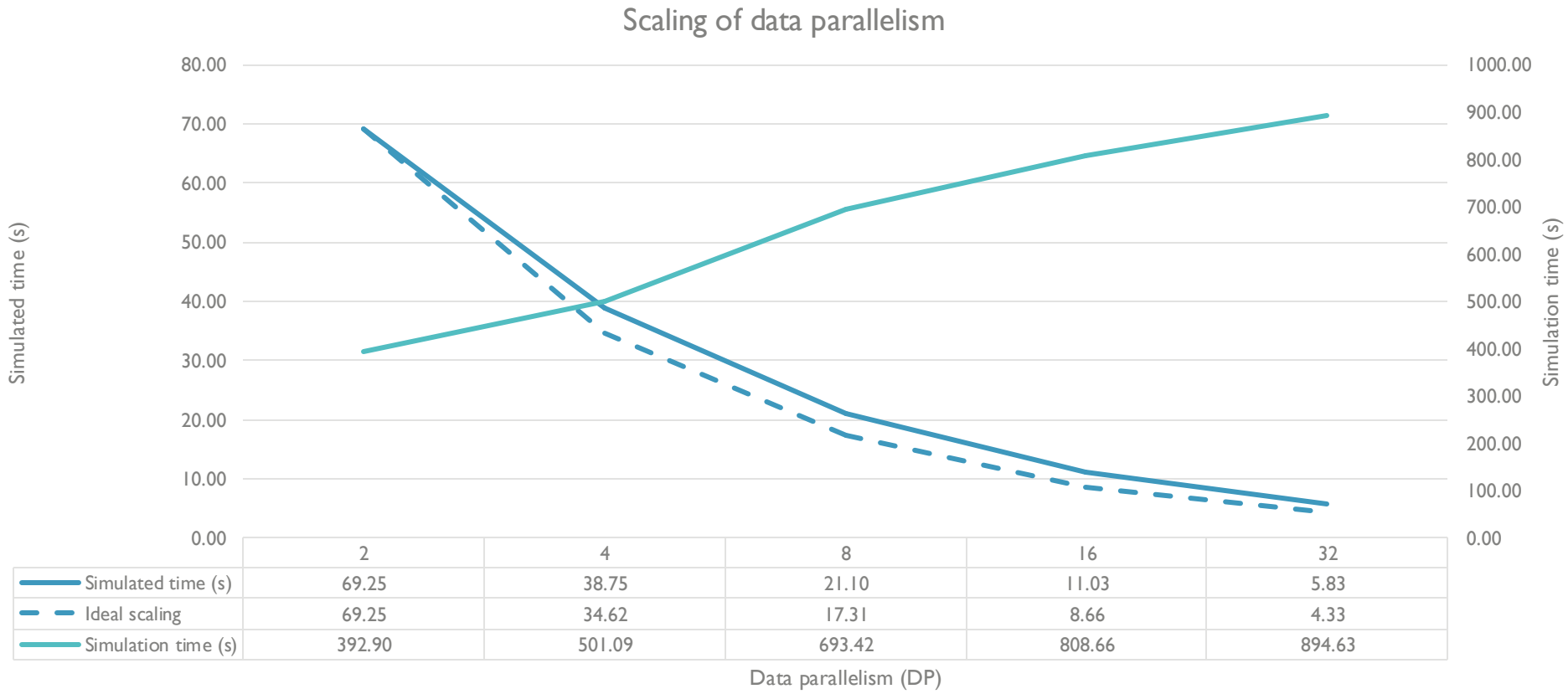
*sst --print-timing-info training_llm.py -- --tp 1 --pp 1 --dp **DP** --batch_size 16 --sequence_len 1024 --n_batch 64 --llm_config small_config.json --log logger --stats stats.csv --topology single --verbose 10*

You can fill the table below:

| Data Parallelism (DP) | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|
| Simulated time (s) | | | | | |
| Simulation time(s) | | | | | |

# Exploring scaling of LLM training

## Data parallelism




| | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|
| Simulated time (s) | 69.25 | 38.75 | 21.10 | 11.03 | 5.83 |
| Ideal scaling | 69.25 | 34.62 | 17.31 | 8.66 | 4.33 |
| Simulation time (s) | 392.90 | 501.09 | 693.42 | 808.66 | 894.63 |

# Exploring scaling of LLM training
## 3D parallelism

**Let's explore the impact of the topology on the performance:**

Command to run:

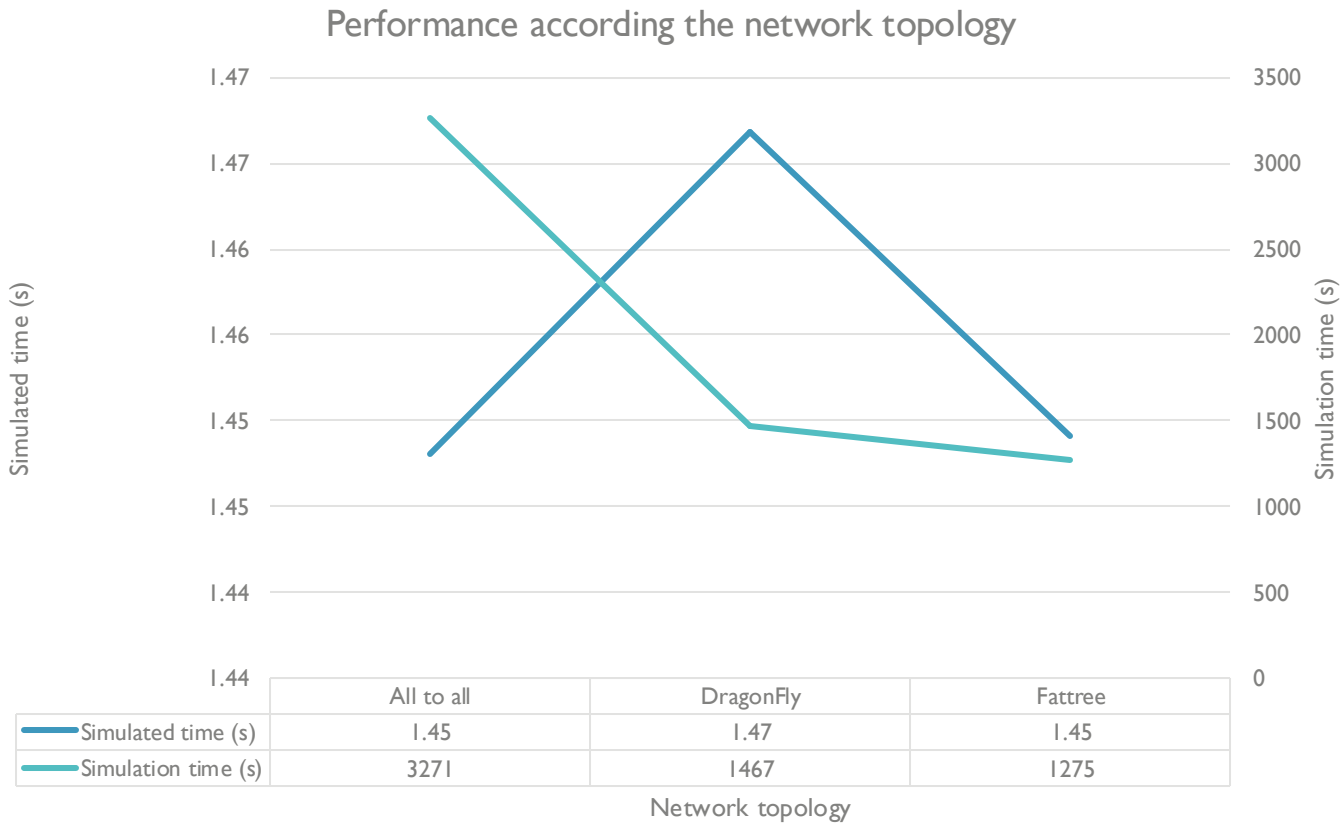
*sst --print-timing-info training_llm.py -- --tp 8 --pp 6 --dp 8 --batch_size 16 --sequence_len 1024 --n_batch 32 --llm_config small_config.json --log logger --stats stats.csv --topology **TOPOLOGY** --verbose 10*

You can fill the table below:

| Topology | single | dragonfly | fattree |
|---|---|---|---|
| Simulated time (s) | | | |
| Simulation time(s) | | | |

# Exploring scaling of LLM training

## 3D parallelism



# Wrap-up

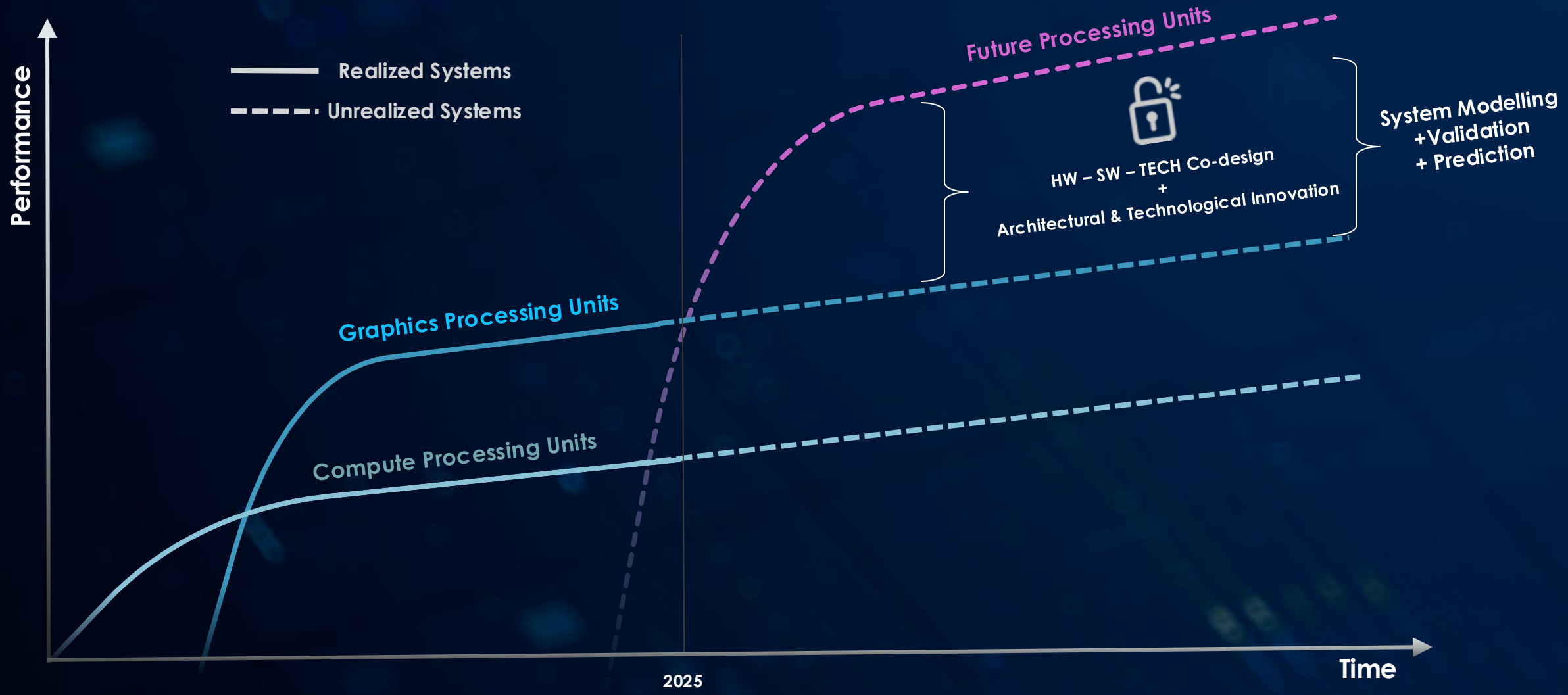
## SST provides an open-source framework for simulating large-scale HPC systems:

- Discrete event simulation parallelized with threads and MPI ranks
- Library of computer system elements
- Python library to build scale-out system with complex network topologies
- Provides a RISC-V CPU model with an emulated operating system:
  - Allows to run multi-threaded applications
  - Allows to control a co-processor:
    - e.g., Balar:Vanadis + GPGPUSIM
- Provides a MPI traffic generator:
  - Allows to simulate thousands of ranks in minutes

SST also allows connecting a gem5 compute core with its uncore infrastructure.

# Broader use in system architecture HW-SW codesign framework

# Enabling the next breakthrough in performance

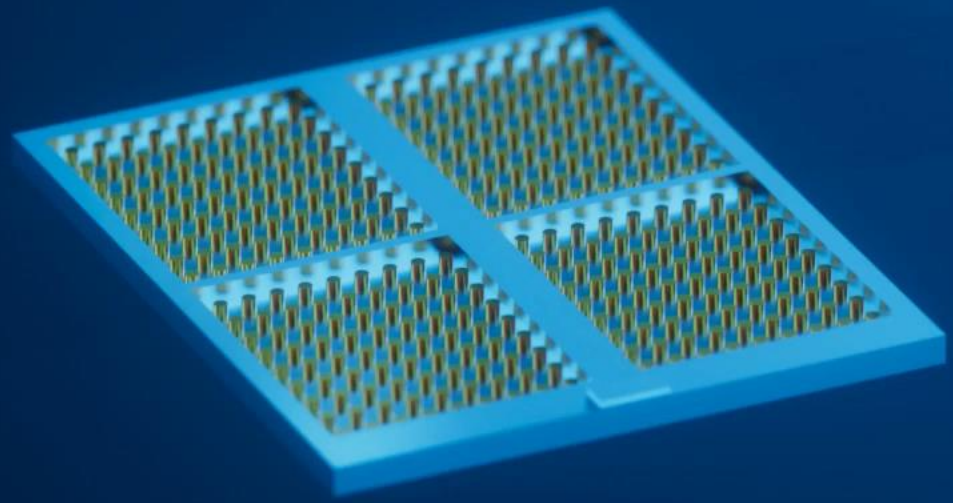


# Pathfinding Advanced Compute for HPC / AI

High throughput HPC / AI nodes

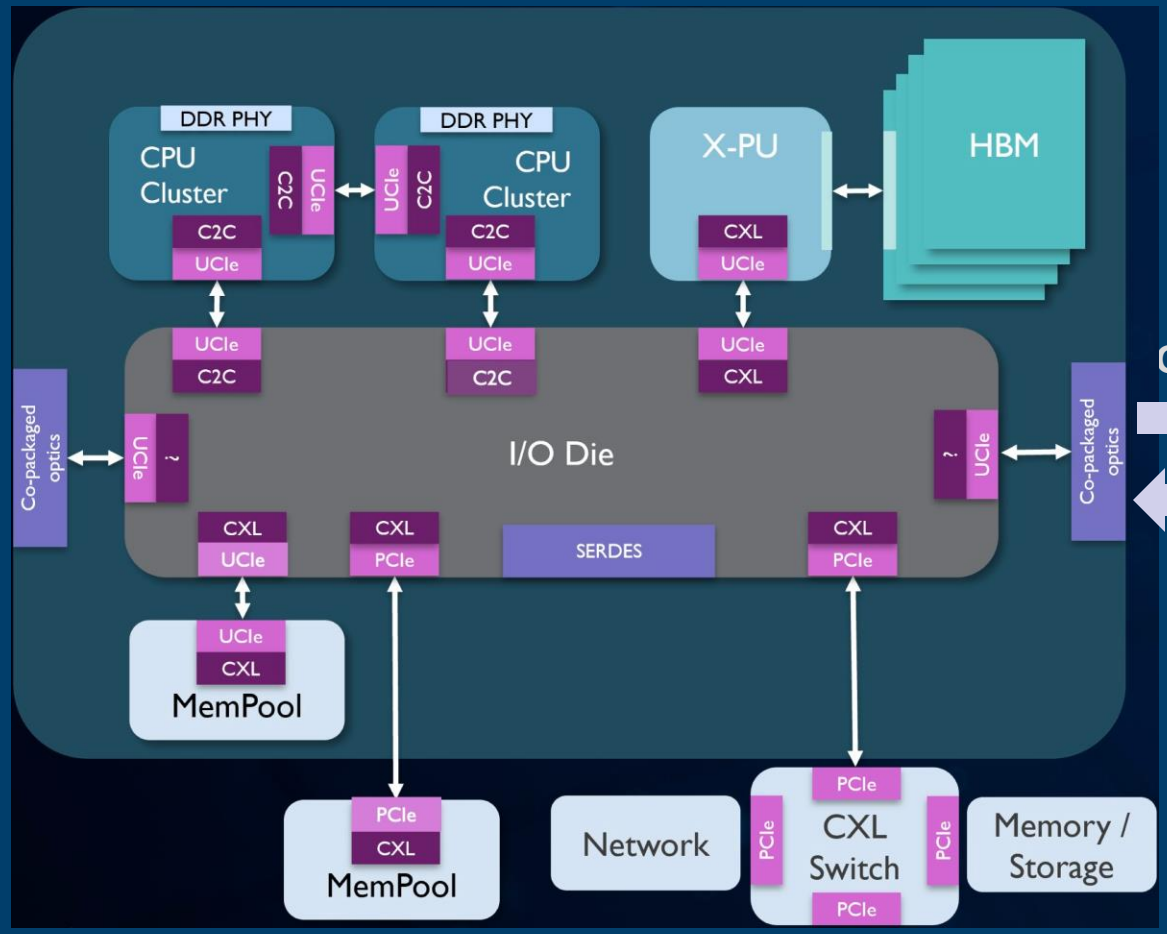
High-performance core arrays and 3D SRAM

Novel memory subsystem with Data Processing Unit and HBM chiplets.



# Chiplet Platform

## De-risking and Pathfinding via Virtual Platform Models




Questions

Insights

Representative workload model

Virtual Platform Model

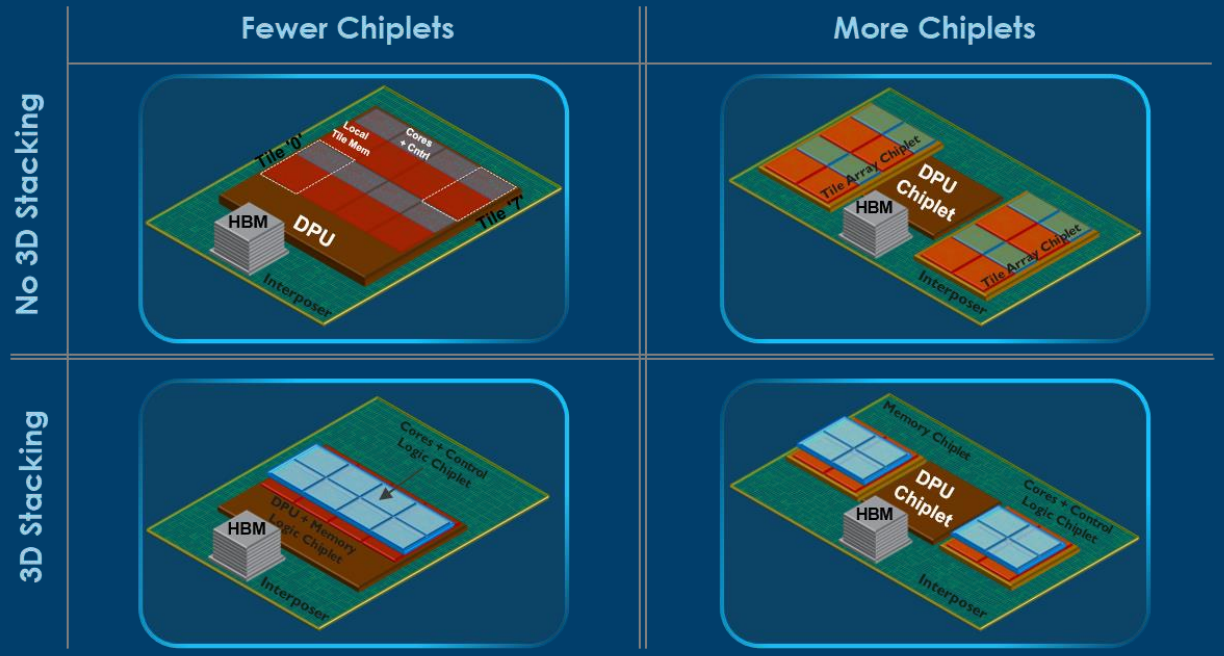
Performance
Power
Cost
Emissions

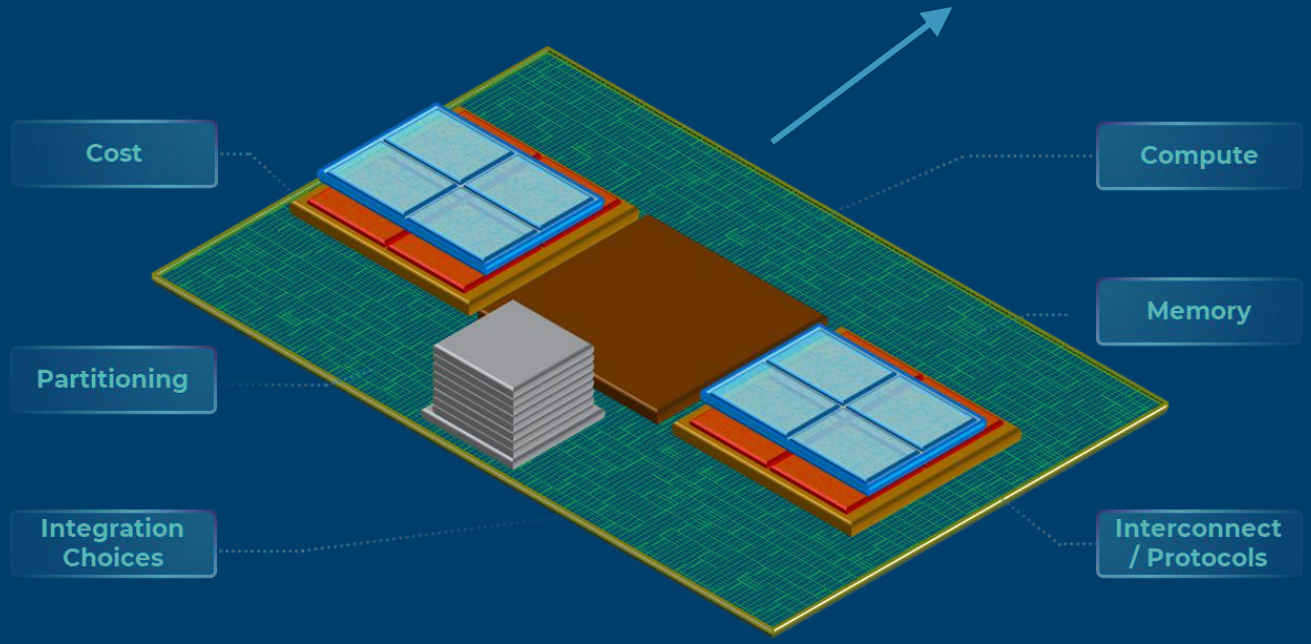
Physical / Technology Models

# Chiplet Platform

## Heterogenous System partitioning

Cost–Performance Co-Optimization across the different packaging and associated protocol layers



# We work **with you** to pathfind HPC / AI solutions of the future

Get access to our state-of-the-art infrastructure

Benefit from our large international network of expertise

Enhance competitiveness with our in-depth technological knowledge

Lower your research costs and risks through pre-competitive collaboration

Accelerate your product roadmap and reduce your time to market

**www.imec-int.com/csa**

# Learn more about our research

## A snapshot of some of our publications

- Graening et al. "Cost-Performance Co-optimization for the Chiplet Era", IEEE Electronics Packaging Technology Conference (EPTC), (2024)
- Pal et al., "System-Technology Co-Optimization for Advanced Integration: A Perspective in the Computing Context", Nature Reviews Electrical Engineering, (2024)
- Kundu et al. "Performance Modeling and Workload Analysis of Distributed Large Language Model Training and Inference", 2024 IEEE International Symposium on Workload Characterization (IISWC), (2024)
- Delestrac et al. "Multi-level Analysis of GPU Utilization in ML Training Workloads", 2024 Design, Automation & Test in Europe Conference (DATE), (2024)
- Chamazcoti et al. "Exploring Pareto-Optimal Hybrid Main Memory Configurations Using Different Emerging Memories", IEEE Transactions on Circuits and Systems I: Regular Papers, (2023)

Research@CSA!

https://www.imec-int.com/en/expertise/compute-system-architecture

# Work with us

## Imec jobs

- **Internships and Full time positions available in BE, NL, UK and US**

https://www.imec-int.com/en/work-at-imec/job-opportunities

# Acknowledgements


- Erwan Lenormand for the SST tutorial
- Contributors of the CSA Compute Epic for the groundwork for this tutorial
- Arindam Mallik for supporting and reviewing the workshop content
- Members of CSA


embracing a better life