PyTorchSim: A Comprehensive, Fast, and Accurate NPU Simulation Framework

Wonhyuk Yang*
POSTECH
Pohang, Republic of Korea
wonhyuk@postech.ac.kr

Yunseon Shin*
POSTECH
Pohang, Republic of Korea
ysshin@postech.ac.kr

Okkyun Woo*
POSTECH
Pohang, Republic of Korea
okkyun.w@postech.ac.kr

Geonwoo Park†
Samsung Electronics
Suwon, Republic of Korea
gbb.park@samsung.com

Hyungkyu Ham
POSTECH
Pohang, Republic of Korea
hhk971@postech.ac.kr

Jeehoon Kang
KAIST
Daejeon, Republic of Korea
FuriosaAI
Seoul, Republic of Korea
jeehoon.kang@furiosa.ai

Jongse Park
KAIST
Daejeon, Republic of Korea
jspark@casys.kaist.ac.kr

Gwangsun Kim‡
POSTECH
Pohang, Republic of Korea
g.kim@postech.ac.kr

Abstract

Deep Neural Networks (DNNs) have continuously increasing demands for the performance and efficiency of Neural Processing Units (NPUs). While analytical models enable rapid exploration of high-level aspects (e.g., tiling), later stages of NPU design require a cycle-accurate simulator that supports various scenarios. However, existing NPU simulators are limited in several aspects, including support for high-speed, multi-core, multi-model tenancy, generic ISA (with vector operations), compiler, data-dependent timing model, and enabling both inference and training.

To address these challenges, we propose PyTorchSim,[1] a novel NPU simulation framework integrated with PyTorch 2. PyTorchSim models NPUs with a custom RISC-V-based ISA extended to support various acceleration units (e.g., systolic array). Our custom backend for PyTorch 2 compiles a given DNN using this ISA through lowering passes with MLIR and LLVM. Then, our extended Gem5 and Spike simulators execute the machine code to accurately model the DNN's timing and functional aspects on the NPU. However, as such a conventional Instruction-Level Simulation (ILS) inevitably runs slowly, we propose Tile-Level Simulation (TLS) to improve speed without sacrificing accuracy. It uses tile-granularity operation latencies from offline ILS runs for high speed while still modeling DRAM and interconnect with cycle-accurate simulators. Furthermore, TLS can also be employed for sparse tensor operations using auxiliary per-tile latency obtained offline. As a result, PyTorchSim provides speedups of up to 139× compared to Accel-Sim, while achieving high simulation accuracy against Google TPUv3 with an MAE of 11.5%. Additionally, we also demonstrate the effectiveness of PyTorchSim over existing simulators for different scenarios, including heterogeneous dense-sparse NPU, multi-model tenancy, compiler optimization, chiplet-aware NPU scheduling, and impact of DNN training hyperparameter.

Keywords

NPU simulation, deep learning compiler.

ACM Reference Format:
Wonhyuk Yang, Yunseon Shin, Okkyun Woo, Geonwoo Park, Hyungkyu Ham, Jeehoon Kang, Jongse Park, and Gwangsun Kim. 2025. PyTorchSim: A Comprehensive, Fast, and Accurate NPU Simulation Framework. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3725843.3756045

1 Introduction

Modern DNNs pose rapidly growing demands for more computing power and memory capacity with higher costs. As a result, it has become increasingly important to design high-performance, cost-efficient NPUs. In computer architecture, new designs are typically evaluated with simulation before FPGA or ASIC hardware (HW) evaluation to avoid high costs. As a result, simulators that can evaluate diverse designs, workloads, and scenarios have contributed significantly to the area by facilitating quick architectural design evaluation [32, 33, 36, 57, 65, 80]. Thus, it is important to design comprehensive NPU simulators as well. As simulators have fidelity-speed trade-offs [30], different simulation methods are best suited for evaluating different aspects of NPUs (e.g., fast analytical model for exploring diverse tensor tiling strategies vs.

*These authors contributed equally to this work.
†This work was done while the author was at POSTECH.
‡Corresponding author.
[1]PyTorchSim is available at https://github.com/PSAL-POSTECH/PyTorchSim.

MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756045

cycle-accurate model for studying DRAM contention between concurrent DNN inferences). In this work, we focus on developing a high-fidelity NPU simulation framework that meets the requirements listed below, which existing simulators fail to fully satisfy.

- **High simulation speed** to maximize coverage of various NPU designs and execution scenarios within a given simulation infrastructure budget, while still providing cycle-accuracy.[2]
- **Multi-core and multi-tenant NPU support**, as modern server systems and even wearable devices have multi-core NPUs [17] and run multiple DNNs [68].
- **Cycle-accurate DRAM and interconnect models** for accurately modeling contention for shared resources [39, 57, 96].
- **Generic and extensible ISA** to support various DNNs, which continually evolve with new operators [52, 55].
- **General vector operations** for end-to-end evaluation of modern DNNs, beyond support for GEMM and convolution [55].
- **Compiler support** for analyzing compiler optimization techniques [4, 31].
- **Training support** for evaluating the DNN training phase, which has different characteristics from inference [62].
- **Data-dependent timing model** for accurately modeling the latencies of sparse matrix/vector operations.
- **Ease of use** for DNNs implemented with popular ML frameworks (e.g., PyTorch) without manual DNN format conversion.
- **Extensibility** for supporting various core designs.

However, there are several challenges and conflicts among the requirements that make it difficult to accomplish this goal while maintaining cycle accuracy. Notably, the demand for high-speed simulation directly contradicts the need for detailed, cycle-accurate modeling of the core, interconnect, and DRAM. In particular, as data-dependent timing models are typically implemented with Instruction-Level Simulation (ILS) [37, 65] approach, they do not achieve high speed. Furthermore, unlike CPU and GPU architectures, NPUs lack a widely adopted ISA, which has also led to the lack of open-source deep learning compilers for NPUs that are mature enough to support various DNNs for both inference and training. Consequently, NPU simulators often rely on custom DNN description formats rather than ML frameworks such as PyTorch [57, 89, 94].

To address these challenges, we propose *PyTorchSim*, a novel NPU simulation framework, to facilitate effective design space exploration for future NPUs. It consists of a high-level request scheduler, a compiler backend for PyTorch 2 [31], and functional/timing NPU models that can target various dataflows. Since PyTorch is used in the majority of recent deep learning projects [88], in most cases, we do not require the user to convert existing DNN implementations into a special format, unlike existing simulators [57, 85, 94].

We build PyTorchSim upon the compilation infrastructure of PyTorch 2, which captures a computational graph in FX graph format for forward/backward passes and lowers it into Aten IR in the frontend. Its Inductor backend then lowers it into loop-level IR for various device-specific backends. Our codegen backend lowers it to our NPU ISA through MLIR [73] and LLVM [72], or pre-defined template kernels, similar to CPU/GPU backends [46, 98]. Exploiting a generic compilation flow, PyTorchSim can model the impact of compiler optimizations, unlike existing simulators that lack compilers and use custom formats. Further, with PyTorch's automatic differentiation, we enable simulation of training as well as inference.

PyTorchSim can also flexibly model different NPU architectures. As a generic and extensible ISA targeted by our compiler backend, our NPU model adopts an extended RISC-V ISA. It supports custom instructions for DMA and dataflow unit (e.g., systolic array) operations while exploiting existing scalar/vector instructions to express various operations in modern DNNs. As a generic interface between the dataflow unit and the rest of the NPU core, we adopt the approach of SiFive Vector Coprocessor Interface (VCIX) [14]. Thus, while we currently support the systolic array (SA) dataflow unit, users can replace it with other units with different microarchitectures. It is even possible to replace the NPU core model entirely, as we demonstrate with a heterogeneous NPU case study (§5.1).

After compilation for this target NPU, the machine code can be executed one instruction at a time with ILS, by Gem5 [37, 76] (for NPU core's timing model) and Spike [1] (for NPU core's functional model) simulators that we have extended and integrated to accurately model NPU cores. The compiler-generated binary is also used for functional correctness and DNN model accuracy validation. However, such a detailed simulation approach inevitably runs slowly, conflicting with the goal of achieving high simulation speed.

To realize fast simulation without losing accuracy, we also propose an alternative approach called *Tile-Level Simulation (TLS)*. The TLS method is proposed based on the observation that the scalar, vector, and matrix instructions between load/store DMAs can be executed with *deterministic* latencies, because the operands are accessed from core-internal SRAM after DMA transfers from DRAM [54, 63]. Since the operations are performed in tensor tile-granularity [89], the deterministic latency of a tile-operation can be obtained *offline* and repeatedly used during different simulation runs to achieve high speed. In PyTorchSim, the tile-operation latencies are obtained through our extended Gem5 and Spike simulators. However, in contrast to the tile's compute latency, the latencies of the DMAs are *non-deterministic* due to contention. Thus, they are modeled *online* using cycle-accurate NoC and DRAM simulators [59, 78]. To express the tile compute operations and their dependencies with DMAs in a DNN, we propose a *Tensor Operation Graph (TOG)*. The TOG can be automatically generated by our PyTorch 2 compiler backend, and then be executed on our NPU timing simulator called *TOGSim* with high speed and high accuracy.

Furthermore, we provide a novel observation that, even if the tile operation is data-dependent (e.g., sparse tensors), its compute latency is deterministic for *each particular* tile, while it can vary *across* tiles. To enable TLS for this case, our extended Spike simulator is used with Gem5 to obtain the latencies for each tile and generate an auxiliary tile-latency file associated with the TOG. The TOGSim can then use them to achieve both high simulation speed and accuracy. It also extends to different core models (§3.3.2), even including sparse cores (§5.1). After the TOGs are obtained for different datasets, they can be reused over simulations with different NPU configurations as in typical exploration use cases. For multi-model

[2] In this work, we define *cycle-accurate* simulator as a simulator that can model a target system with accurate cycle counts, under given configuration parameters. However, it does not necessarily output the same cycle count as the target if there are unknown parameters/components (e.g., the sizes and existence of different buffers or frequencies of internal components, such as NoC and cache) or software (SW) implementation.

Table 1: Comparison of the features of different NPU simulators (ICNT: interconnect).[†]

	High speed	Multi-core	Multi-DNN tenancy	Cycle-accurate DRAM&ICNT	General vector ops	Compiler support	Training support	Base ISA	Data-dependent timing model	Model input format
Accel-Sim [65]	✗	✓	✗	Both	✓	✓	✓	PTX/SASS	✓	Instr. trace
mNPUsim [57]	✗	✓[‡]	✓[‡]	DRAM	✗	✗	✗	✗	✗	Custom
SCALE-Sim v3 [91]	✓	✓	✗	DRAM	✗	✗	✗	✗	Partial[**]	Custom
SMAUG [104]	✗	✓	✗	Both	✓	✗	✗	x86	✗	Custom
SST-STONNE [5, 85]	✗	✗	✗	Both	✗	✗	✗	Custom	✓	PyTorch[***]
MAESTRO [69]	✓	✗	✗	✗[*]	✗	✗	✗	✗	✓	Custom
Timeloop [89]	✓	✓	✗	✗[*]	✗	✗	✗	✗	✗	Custom
Sparseloop [103]	✓	✓	✗	✗[*]	✗	✗	✗	✗	Partial[**]	Custom
GeneSys [55]	✓	✗	✗	DRAM	✓	✓	✗	Custom	✗	ONNX
PyTorchSim	✓	✓	✓	**Both**	✓	✓	✓	**RISC-V**	✓	**PyTorch**

[†]While Accel-Sim is a GPGPU simulator, it is included because GPUs are widely used to accelerate DNNs. We exclude simulators, such as LLMServingSim [45] and vTrain [34], that do not have an NPU microarchitecture model and rely on an external NPU simulator or real GPU profiling results.
[‡]mNPUsim's core model is limited to supporting only a batch size of one.
[*]Their DRAM model does not model the microarchitectural details (e.g., row buffer hits/misses) while the network model ignores contention.
[**]Sparseloop estimates compute cycles based on tensor sparsity statistics, rather than actual data values. SCALE-Sim v3 lacks unstructured sparsity support.
[***]STONNE's PyTorch interface only supports GEMM and CONV, while other operations are ignored.

tenancy, multiple TOGs from various DNNs can also be executed simultaneously. With multiple NPU cores in PyTorchSim, different scheduling policies can be used to study their impact.

To summarize, we make the following contributions:

- To our knowledge, PyTorchSim is the first comprehensive NPU simulator that supports training, compilation, and high-speed simulation of both dense and sparse operations, while overcoming various limitations of existing simulators (Table 1).
- These features are enabled by our custom MLIR-based compiler backend, integrated with a widely-used ML framework (i.e., PyTorch). Thus, PyTorchSim can easily support existing DNNs without model format conversion, unlike existing simulators.
- Our Instruction-Level Simulation (ILS) mode combines our extended Gem5 and Spike simulators to accurately model both timing and functional aspects of the compiled DNN workload.
- To overcome the slow speed of ILS, we propose a Tile-Level Simulation (TLS) mode that uses our compiler-generated Tensor Operation Graph (TOG) DNN representation to achieve high speed while accurately modeling inter-core contention.
- We validate the accuracy of PyTorchSim with Google TPUv3, showing a mean absolute error of 11.5% while achieving a significant speedup of up to 139× compared to Accel-Sim, which is the closest to PyTorchSim in terms of the features supported.
- We also demonstrate the versatility of PyTorchSim with various case studies, including heterogeneous dense-sparse NPU, multi-model tenancy, compiler optimization, chiplet-aware NPU scheduling, and the impact of DNN training hyperparameter, which no other existing simulators support.

2 Motivation / Background

2.1 Limitations of Existing NPU Simulators

Analytical NPU models (e.g., Timeloop [89] and Sparseloop [103]) and NPU simulators that focus on modeling a single core [55, 85, 94] can achieve very high speed while capturing important characteristics at the core level, such as operation latency and memory bandwidth requirements. Thus, they can be useful for rapidly exploring a huge design space and mapping/tiling choices, especially in the early stages of NPU design. However, they do not model microarchitectural details and contentions with shared resources such as DRAM and interconnect, which can significantly impact the NPU performance in various scenarios, especially with multiple cores running different DNNs simultaneously [68, 105]. Thus, in later stages of NPU design, cycle-accurate multi-core NPU simulators with detailed shared resource models are necessary, despite their relatively lower speed. For fast simulation, prior work [57] exploited tile-compute-latency determinism, but it required manual latency estimation by the simulator developer, which is error-prone and not scalable across numerous kernels from various DNNs for inference and training. Also, it cannot model data-dependent compute latencies.

Large DNNs also pose efficiency challenges, which demand innovative DNN compression techniques (e.g., pruning), necessitating data-dependent timing models. However, execution-driven simulators [33, 85] with such capability run slowly because all arithmetic operations have to be executed within the simulator. Trace-driven simulators [65, 104] are relatively faster but still limited in speed due to modeling of instruction-level details (e.g., control signals like SIMD lane masks).

Another feature often overlooked in NPU simulators is compiler support, despite the importance of compiler optimization and training scenarios. While GeneSys [55] provides a DNN compiler, it lacks automatic differentiation, which is necessary for training. As a GPU simulator, Accel-Sim [65] can exploit the nvcc compiler, but the compiler is closed-source and new instructions cannot be added to the compiler even if they can be added in Accel-Sim.

2.2 PyTorch 2

Among different frameworks, PyTorch has been the most widely used framework among ML practitioners [88], with more implementations consistently available in PyTorch than in any other framework over the past five years [21]. PyTorch is also predominantly used in the reference implementations of MLPerf Inference and Training workloads [81, 82], although users may implement them with other frameworks. While PyTorch 1.x primarily operated

in eager mode, PyTorch 2 [31] introduced the graph-based compiled mode to provide higher performance. It has also addressed the usability and inflexibility limitations of earlier graph-based deep learning compilers through TorchDynamo, AOTAutograd, and TorchInductor. The compiler natively supports dynamic tensor shapes (e.g., varying batch sizes or sequence lengths).

In the compiled mode, for a given DNN model, TorchDynamo first automatically captures an FX graph [92] representing the PyTorch operations. For training, AOTAutograd also automatically generates another FX graph for backpropagation ahead-of-time. The captured FX graph uses the ATen/Prims intermediate representation (IR). Then, in the compiler backend, TorchInductor lowers them into loop-level IR or device-specific kernels. The loop-level IR is further lowered into machine code of different processors by device-specific backends. Currently, this IR mainly supports pointwise and reduction operations other than data movement and conditional execution. For matrix multiply, the Aten IR nodes (i.e., `aten.mm` or `aten.addmm`) are mapped to device-specific kernel implementations (e.g., using cuBLAS/CUTLASS, Triton [98], or OpenMP), rather than using the loop-level IR. Throughout compilation, different optimizations, including dead code elimination, constant folding, and operation fusion can be applied. Currently, TorchInductor uses Triton and OpenMP to generate code for the GPU and CPU, respectively, but it can be extended to support other HW such as NPUs. Although not well documented, TorchInductor supports parallel compilation using multiple threads via `AsyncCompile` [23] and enables the compilation of one operation to overlap asynchronously with the execution of another. We exploit this feature to hide compilation overhead during simulation.

2.3 MLIR and LLVM

LLVM [72] is a compiler infrastructure and toolchain project that supports various programming languages and architectures. It introduces the LLVM Intermediate Representation (IR), which serves as a common IR for frontends of different programming languages and facilitates translation to multiple target architectures. LLVM-based compilers can generate assembly code for vector processing units, such as the RISC-V vector extension.

Multi-Level Intermediate Representation (MLIR) [73] enables optimization across different levels of abstraction through the use of dialects. Moreover, MLIR programs can be lowered into the LLVM dialect and further translated into LLVM IR, allowing compilation into RISC-V binaries. MLIR is increasingly integrated into machine learning frameworks such as PyTorch 2.0, enabling systematic lowering and optimization of tensor computations for diverse HW backends.

2.4 Extending RISC-V for NPU Simulator

The choice of ISA for a processor should take into account the maturity of its SW ecosystem, as it significantly impacts the engineering effort required to develop the full system stack and to support a wide range of algorithms and applications. Additionally, to accommodate the evolving demands of modern workloads, the extensibility of the ISA is a critical factor. These considerations apply equally to NPUs, and NPU simulators must also be designed with them in mind.

In this regard, RISC-V is a strong candidate ISA for NPU simulators. It is an open, extensible, and modern ISA, unencumbered by legacy instructions, and supported by a sufficiently mature and growing SW ecosystem [93], including simulators [1, 36] and compiler infrastructure [13]. Furthermore, its openness eliminates intellectual property concerns, enabling contributions from both academia and industry – an important catalyst for advancing tool development [53, 77, 102]. Currently, no other open ISA has comparable momentum for ecosystem growth as RISC-V [50]. In addition, the RISC-V vector extension [2] is vector-length agnostic, offering flexibility for architectural exploration across varying vector widths, in contrast to fixed-width vector/SIMD instruction sets such as AVX-512 [58], PTX/SASS [18], or RDNA [24].

As a result, several NPUs have been built in the industry, using RISC-V in the datapath [50–52]. However, the architectural differences between the CPU and NPU need to be addressed by extending RISC-V to support necessary features, including SW-managed scratchpad memory, dataflow units to accelerate tensor/matrix operations, and support for efficient data movement to the dataflow units.

Fortunately, the Vector Coprocessor Interface Extension (VCIX) from SiFive provides a generalized interface between a RISC-V vector unit and a custom acceleration unit. The wide interface between the acceleration unit and the vector register file effectively avoids the communication overhead of the conventional integration approach using relatively narrow on-chip buses. Furthermore, since the VCIX dialect in MLIR can generate custom instructions for custom coprocessors, it can be used to control dataflow units (e.g., SA) in NPU architectures.

3 PyTorchSim Framework

3.1 Overview

PyTorchSim is a compiler-integrated, fast, and cycle-accurate NPU simulation framework that supports a comprehensive set of features (Table 1). While PyTorchSim can be extended to support multiple ML frameworks, including TensorFlow (§3.6.4), we prioritize PyTorch due to its dominance as the most widely used framework for implementing state-of-the-art ML models [21] to make them more accessible to NPU architects.

PyTorchSim consists of a request generator/scheduler, a compiler backend for PyTorch 2, and NPU functional/timing models, including Spike, Gem5, and TOGSim (Fig. 1). We employ the compiled mode of PyTorch 2 to enable NPU simulation without DNN model code modification, while properly modeling the impact of the compiler optimizations and supporting DNN training. The only changes required in the PyTorch code are specifying our simulated NPU as the target device, importing the device library, and choosing this device in the beginning. Among different compilation approaches for PyTorch, we adopt Inductor introduced with PyTorch 2, considering its broad adoption in deep learning, and we extend it for NPU simulation. Our backend for Inductor takes the Aten IR and loop-level IR, lowers them through MLIR and LLVM IR, and generates machine code for our target NPU architecture. Our target NPU architecture is based on a customized RISC-V ISA with vector extension, which includes a rich set of instructions

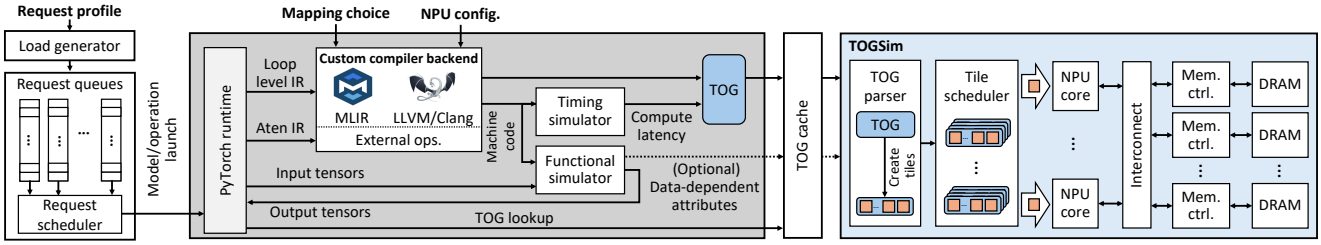

Figure 1: Overview of PyTorchSim framework.

Table 2: Simulator usages and corresponding components.

Usage	Simulator(s)
DNN accuracy validation	Spike
TOG (§3.7) generation (no data dependence)	Gem5
TOG (§3.7) generation (data-dependent)	Gem5 + Spike
Inference performance evaluation	TOGSim
Single-iteration training performance evaluation	TOGSim
Full training performance evaluation	TOGSim + Spike

to support diverse operations in modern DNNs [52, 55] as well as custom instructions to model NPU's dataflow (§3.4).

To overcome the challenge of meeting conflicting requirements for NPU simulators – including high simulation speed, cycle-level fidelity, and support for a functional model – we combine multiple simulators that address different aspects. These simulators can then be selectively enabled depending on the simulation task (Table 2). For example, by comparing the output from the Spike[3] simulation with that from a real CPU/GPU, the user can verify that the compilation for the NPU is done correctly.

In addition, to overcome the slow simulation speed of ILS, we introduce *TOGSim* (§3.7), which implements our proposed, high-speed TLS approach. For TLS, our extended Gem5 [75, 76] is used to model the NPU core and obtain the deterministic, data value-independent part of tile-level operations, by executing the instructions between DMA loads/stores. In case the latencies are data-dependent, Spike is used along with Gem5 to obtain the latencies offline for all tile operations for a given DNN and input data. These latencies are combined in the Tile-Operation Graph (TOG) for accurate and fast simulation using the TOGSim. It executes the TOG generated by our compiler backend along with NPU's compute latencies obtained offline from Gem5 while modeling DRAM and interconnect through cycle-accurate simulators for high accuracy. Furthermore, multiple TOGs can be used simultaneously to study multi-model tenancy scenarios and the impact of different scheduling policies.

Note that while training performance evaluation for a single iteration only requires the timing model (i.e., TOGSim), full training evaluation also requires the functional model (i.e., Spike) to obtain the loss values, which determine the total number of iterations.

[3]While Gem5 also has a functional model for RISC-V, its support for vector extensions is currently incomplete, making Spike a better choice for functional simulation of our RISC-V-based ISA for the NPU architecture.

3.2 Example Use Cases and Limitations

PyTorchSim is a versatile simulator that can support various NPU HW/SW evaluation scenarios beyond the cases supported by prior NPU simulators (§5). We describe some key example use cases below, while this is not an exhaustive list:

HW microarchitecture evaluation. Any HW components of PyTorchSim, including NPU core, scalar/vector units, DMA engines, on-/off-chip memory, and interconnect can be configured or modified to study different architectures. For instance, we present a case study on a heterogeneous NPU with dense-sparse matrix units and the impact of the memory system (§5.1).

SW optimization. SW optimizations can have a significant performance impact. We show studies on compute-DMA overlap and tensor layout modification (§5.3).

HW/SW co-design. New HW features often require SW support. For example, to support a multi-dimensional DMA feature [62], our compiler generates the necessary DMA commands (§3.6.3).

DNN training study. Through the compiler support, we enable studies on the training performance (§5.5).

Multi-model tenancy study. In contrast to other simulators, we support different scenarios with concurrent multi-DNN execution (e.g., performance interference between different DNNs (§5.2)).

While we believe PyTorchSim advances the state of the art in NPU simulation, as shown with various case studies, it currently lacks several features that are not fundamental and can be addressed with further development. For example, PyTorchSim does not yet support distributed inference or training; however, it can be extended as discussed in §3.9.3. Similar to other NPU simulators in Table 1, PyTorchSim does not support sub-16-bit quantization; however, we plan to add this capability in future versions. We further discuss the limitations of our current compiler implementation in §3.6.4. Finally, our proposed TLS cannot model latency accurately if the computation exhibits true nondeterminism even with the same input data (e.g., when using a random number generator). In such cases, TLS would need to be augmented with a statistical method.

3.3 Modeling NPU Architectures

3.3.1 Core Model. We model a generic NPU core that comprises scalar, vector, and dataflow-based matrix units for computation as well as a transpose-capable DMA engine (Fig. 2). However, users can implement diverse core models, including sparse cores (§3.3.2), and still take advantage of our fast TLS approach. For double-buffering [63], on each NPU core, two contexts with different tile operations can be executed concurrently. The tiling method we implement (§3.6) ensures that the scratchpad memory can hold two tile

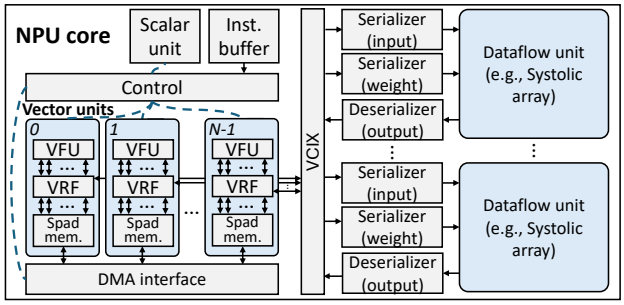

Figure 2: NPU core microarchitecture model.

sets. Alternatively, external tiling optimizers (e.g., Timeloop [89]) can also be used with the simulator.

Within the core, the dataflow unit communicates with the rest of the core through a wide VCIX-like interface to the vector unit's VRF (Vector Register File) to avoid an interface bottleneck. The interface is connected with each of N vector units through three FIFO queues – two serializer FIFO queues for providing weight and input tensor elements to the dataflow unit, and a deserializer FIFO queue from the dataflow unit to receive output tensor elements. When vector data from a vector unit's register is pushed to a serializer, all data elements in the vector register are pushed to the FIFO queue at once, and then, shifted into the SA row or column one element at a time. Similarly, the dataflow unit's output can be popped from a deserializer FIFO to a vector register. We present an example systolic array dataflow unit model in §3.5.

3.3.2 Alternative Core Models. To support a new dataflow unit, the compiler passes need to be implemented to apply optimization techniques and progressively lower a high-level IR (e.g., Aten IR) to the machine code. The NPU core timing model in Gem5 also needs to be implemented to calculate the latency of tile operations included in the TOG. In case the latency is data-dependent, Spike also needs to be modified to obtain the dataflow unit's latencies for each tile operation and combine them with the TOG. The scalar/vector units can be further customized with new instructions.

An alternative to using the VCIX interface to the VRF is to make the dataflow unit interface with a scratchpad memory or accumulator [54, 55]. While there are trade-offs, it can be beneficial to make the dataflow unit interface with VRF since it is common to fuse vector operations before and/or after a GEMM to reduce DRAM traffic [86]. For example, as a vector operation is fused with a preceding GEMM, the vector ALUs can immediately use the GEMM output in the VRF, rather than having another data movement step in between to move the GEMM output from the scratchpad memory to the VRF.

Furthermore, it is even possible to integrate PyTorchSim with NPU core models from other simulators, as we demonstrate by employing the Flexagon [84] core model from SST-STONNE [5] for a heterogeneous dense-sparse NPU architecture case study (§5.1).

3.3.3 Memory Hierarchy. The scalar and vector units have their own register files, and a scratchpad memory interfaces with the vector unit as well as the DMA engine [70]. The DMA engine

	7 bits	5 bits	3 bits	5 bits	5 bits	7 bits
(a) mvin/mvout	opcode	X	func3	mm addr reg	sram addr reg	funct7
(b) config	opcode	X	func3	stride info reg	shape info reg	funct7
(c) push	opcode	X	func3	X	src vreg	vfunct7
(d) pop	opcode	dest vreg	func3	X	X	vfunct7
(e) SFU Inst.	opcode	dest vreg	func3	X	src vreg	vfunct7

Figure 3: Custom instructions for our NPU model.

also supports an implicit transpose operation [79]. We also support last-level, shared SRAM managed by SW [62]. Our NPU cores currently do not use L1 data caches, as they tend to improve average latency rather than tail latency, which is more important for meeting SLOs [62, 95]. Recent NPUs also tend to use SW-managed scratchpad memory [27, 62, 90] to avoid unpredictable cache miss latency and maximize compute unit utilization. Even GPUs tend to use SW-managed data movement through (scratchpad-like) shared memory, rather than fetching tensors through L1 data cache [19]. However, it is still possible to model L1 caches by expressing cache accesses as nodes similar to DMAs and specifying the dependency between compute nodes and cache accesses in the TOG (§3.7). While DMA operations directly access global memory, the cache access nodes can first check cache hit before accessing global memory. We assume instructions are fetched from SW-managed per-core instruction memory [87]. Instruction fetch is not considered a significant concern for modern DNNs [106].

3.4 ISA Extension

Considering the openness, ecosystem maturity [50], and generality [52, 55] (§2.4), we adopt RISC-V and customize it for the NPU model. As described below, we introduce complex instructions that involve multiple vector units and/or the dataflow unit (e.g., triggering many PE operations in a systolic array with a single instruction) and multi-dimensional DMA operations, similar to Google TPU [63]. Thus, the resulting ISA is not purely RISC-style. Meanwhile, in contrast to the pure CISC-style ISA of TPUv1 [63], later TPUs have added RISC-style register-to-register vector ALU operations [62, 70], mixing both RISC and CISC instructions. Other NPUs also have RISC-style vector instructions as well as complex matrix multiply instructions [50, 100]. As needed, our NPU ISA can be further customized with new instructions, using the encoding space for RISC-V extensions that are unused in NPUs (e.g., Transactional Memory ("T") or Hypervisor Extension ("H")). While there may be opportunities to further optimize the encoding and thereby improve the efficiency of the NPU's control HW (e.g., the decoder) through the adoption of an entirely new ISA, such considerations are beyond the scope of this work. Instead, we focus on the datapath aspects that directly impact performance.

Scratchpad memory accesses and DMAs. We add support for the scratchpad memory by mapping it to a high virtual address region and making it accessible through load/store instructions. To model tensor DMA engines [62], we introduce `mvin`, `mvout`, and `config` instructions (Fig. 3a-b). There are four different `config` instructions that use parameters from the specified configuration registers – such as the `stride info` and `shape info` registers – to configure DMA parameters, including tensor shape, main memory stride, scratchpad stride, element size, and interleaving granularity

across the scratchpad memories of the vector units. Other parameters (i.e., main memory and scratchpad memory addresses), are provided in the `mm addr` and `sram addr` registers of the `mvin`/`mvout` instructions, which initiate the DMAs.

Dataflow unit operations. For the dataflow unit, we define `ivpush` and `wvpush` instructions to push input activation and weight tensors from vector registers to the dataflow unit, respectively, using the instruction formats defined by VCIX [14]. Execution of an `ivpush` (`wvpush`) instruction pushes a vector register's data from a vector unit i (where $0 \leq i < N$) to its input (weight) serializer FIFO for the dataflow unit (Fig. 2) from all vector units. Similarly, executing a `vpop` instruction pops v output elements from the deserializer FIFO connected to the dataflow unit and transfers them to a destination vector register across all vector units, where v denotes the number of elements in a vector register. Given the nature of dataflow units, their compute operations can be implicitly triggered by pushing input and weight tensors through the serializers rather than having an explicit compute instructions. Synchronization with the vector unit for the output can also be done implicitly by making the `vpop` instruction stall until the deserializer is ready.

Vector special operations. To support operations used in loss functions and activation functions, such as exponentials (`exp`) and hyperbolic tangents (`tanh`), which are not included in the RVV ISAs, we introduce the SFU instruction (Fig. 3(e)).

3.5 Dataflow Unit Example: Systolic Array

While the generic VCIX-based interface can support various dataflow units, by default, we employ the systolic array to accelerate matrix multiply. CONV operations are also implemented as GEMM with implicit im2col [107]. NPUs with systolic arrays (SAs) have been widely deployed at scale by hyperscale cloud providers, as exemplified by multiple generations of Google TPUs and Amazon AWS Inferentia/Trainium [7, 16, 61–63, 70, 87]. Further, the high-level microarchitecture of several Google TPUs are disclosed [62, 87], including details of the vector units [70], making it possible to model them accurately. We assume a weight-stationary SA where each PE (Processing Element) in the SA holds two weights for double buffering [20, 54, 87], overlapping compute and DMAs.

For correct dataflow execution, pushing data from the serializer to the dataflow unit or popping data from the deserializer to the vector register can require different delays across vector units. For the SA, the serializer of vector unit i delays shifting data to the SA by i cycles, ensuring that data are pushed in a skewed manner to achieve proper dataflow within the SA. Similarly, for the pop operation, vector unit i experiences an i-cycle delay. Note that these delays do not necessarily introduce pipeline bubbles between successive instructions, as they are overlapped.

To perform a GEMM, the SA's weights are loaded by pushing weights to the SA using `wvpush`, and the input activation is pushed to the SA using `ivpush`. As the PEs of the SA receive an input activation value, the MAC operations are triggered implicitly, collectively executing a GEMM operation. The weights, once loaded, can be reused for multiple GEMM operations. If `vpop` instructions are not issued properly and the deserializer FIFO becomes full, the SA will stall until `vpop` is done. In addition, `ivpush` instructions will also stall when the SA is stalled. However, since the NPU core microarchitecture is known at compile time, the instructions can be scheduled statically to prevent deadlocks. If needed, out-of-order scheduling can still be employed, as long as the order among `ivpush`, `wvpush`, and `vpop` are preserved; it can be done by marking them as having dependency by the instruction dispatch unit.

3.6 Compiler

3.6.1 Base Compiler. By adopting the compiled mode approach of PyTorch 2, we enable the simulation of existing DNNs implemented in PyTorch without requiring users to manually convert model descriptions. Moreover, by leveraging the automatic differentiation capabilities of PyTorch 2 alongside the RVV-based ISA, which supports a wide range of operations, our framework readily extends to training simulation.

PyTorch's Inductor backend further provides robust support for widely adopted optimization techniques, including graph optimization with operator fusion, autotuning, and loop/layout reordering, achieving competitive performance with other backends [31]. It also benefits from MLIR-based optimizations applied over multiple passes [73], similar to other MLIR-based compilers [25, 38]. Optimization methods that rely on hand-written kernels, such as FlashAttention [47], can be integrated as templates within Inductor.

Building on this foundation, we leverage the compiler frontend of PyTorch 2 and implement a custom Inductor backend targeting our NPU architecture (Fig. 4a), similar to how commercial GPUs, NPUs, and CPUs from NVIDIA, AMD, and Intel are supported. We discuss the extension to other ML frameworks in §3.9.

3.6.2 Lowering Computation Graphs. After an FX graph is captured from PyTorch code, it is first lowered into the Aten IR. The nodes in Aten IR can be further lowered using either custom templates or the loop-level IR of Inductor. Since the loop-level IR does not support GEMM/CONV, we lower them to MLIR kernel templates similar to OpenMP/Triton backends for CPUs/GPUs, while other operations are lowered into loop-level IR, and then MLIR. We use various MLIR dialects, including `affine`, `arith`, `func`, `linalg`, `llvm`, `math`, `memref`, and `vector`. The aforementioned GEMM and CONV templates additionally use the `vcix` MLIR dialect to generate code using our custom instructions (§3.4) to use the interface to the dataflow unit. The MLIR operations are further lowered into LLVM IR for code generation using the existing backend and our extension for the extended RVV. In addition to conventional optimization passes, we also implement additional passes for padding tensors and DMA optimization (§3.6.3).

By leveraging existing MLIR dialects with strong modularity and extensibility, our compiler design maintains a good separation of concerns between higher-level compiler optimizations and HW-specific details. In addition, our approach facilitates the integration of third-party optimizations expressed in MLIR, enhancing adaptability and reusability.

However, this design choice may also present challenges when integrating with other frameworks that do not align closely with the MLIR infrastructure. In this case, it is also possible to take the FX graph in Aten IR and lower it through an *external pass* (Fig. 4), which may or may not involve a compiler. Its output can then be

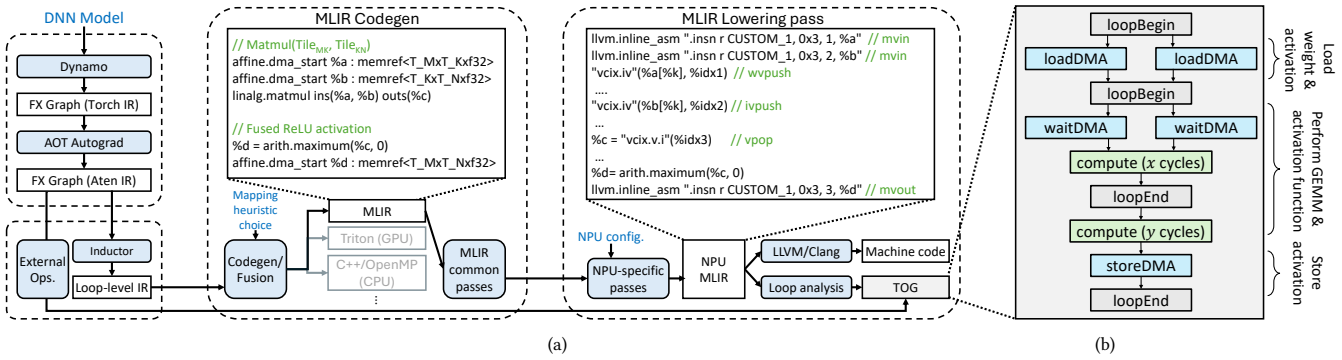

Figure 4: (a) Compilation workflow with simplified example IR pseudo-codes. (b) Example TOG.

converted into the *TOG* (§3.7), which is our internal input format for NPU timing model. We demonstrate this method to integrate STONNE NPU core timing model with diverse sparse cores designs for a case study on a heterogeneous dense-sparse NPU (§5.1).

On a separate note, in academia, deep learning compiler research has typically focused only on real CPUs and GPUs [108] due to limited access to NPU HW and its low-level architectural details. We believe that by using extensible frameworks such as Inductor and MLIR for compilation, PyTorchSim can facilitate future research on NPU compilers and HW/SW co-design.

3.6.3 Optimizations and Tensor Layout. We employ several HW/SW optimizations to properly model high-performance NPUs. First, to optimize mapping/tiling for GEMMs, we currently support a similar mapping heuristic as Gemmini [20], which maximizes the utilization of scratchpad memory, as well as user-specified mapping for different GEMM sizes. Additionally, third-party mapping heuristics (e.g., Timeloop [89]) can be integrated. We also support Inductor's autotuning [12] for choosing tile sizes and vector register group size (i.e., `LMUL` in RVV), and plan to extend it to GEMM/CONV.

Similar to TPUv4i [62], we implement a 4D DMA engine to reduce the overhead of generating addresses and issuing DMA requests with the scalar unit and 1D/2D DMA engines. To maximize the overlap of DMAs and compute, our backend also decomposes coarse-grained tensor DMAs into smaller DMAs, considering the size of the SA. By decomposing DMAs, the dependency of SA to DMAs is tracked in a finer granularity, allowing for compute to begin earlier to reduce idle time, as we show in §5.3.

In addition, our backend supports the fusion optimization of GEMM and element-wise operations (e.g., activation function). Furthermore, to maximize SA utilization for CONV, we also implement tensor layout optimizations. In DRAM, we use PyTorch's default NCHW (H: height, W: width, N: batch size, C: channel) layout, and use our transpose-capable DMA engine [79] to load tensor tile into scratchpad memory in HWNC layout for typical CONVs, using two innermost dimensions to create a single GEMM tile. However, as this layout underutilizes the SA when either N or C dimension is small, we implement optimizations to use different tiling methods in such cases: when N is 1, we use HWC layout and W×C input tile, and when C is small, we use HNWC layout and N×(K_wC) input tile, where K_w is the filter's width. The filter dimensions are determined accordingly. For Transformer models, simply using NSH (S: sequence length, H: hidden dimension) layout throughout is sufficient to achieve high SA utilization.

3.6.4 Compiler Limitations. Different NPUs have various dataflows with specialized components [26, 61, 67, 74, 90]. As a result, their compilers are typically tailored to a specific NPU architecture [63], making them incompatible with other NPUs. Similarly, the compiler of PyTorchSim may not be able to support significantly different NPU architectures without considerable modifications. Our compiler supports various operator fusions [64], including GEMM with epilogue, prologue with GEMM, CONV with epilogue, and GEMM with reduction. However, it does not yet support more complex fusions such as FlashAttention [47] or advanced autotuning as in TVM [43]. We also use hand-written (MLIR) kernels for GEMM and CONV as with OpenMP and Triton backends, rather than pure code generation [6]. Currently, dynamic tensor shape is also unsupported, requiring recompilation when a particular tensor shape is encountered for the first time. Afterward, the code will be cached for future execution. However, based on our evaluation, recompilation overhead is small (~ 4%) as it is done quickly on the real host CPU while NPU simulations run relatively slowly.

3.7 Tile Operation Graph (TOG)

For fast simulation in the TLS mode, we implement a lowering pass that takes our compiler backend's MLIR code and analyzes the loop structure and operations to generate a TOG (Fig. 4b) expressed in our custom ONNX format [8], in addition to binary code.

A TOG is a directed acyclic graph in which each node represents one of the following: the beginning of a loop (`loopBegin`), the end of a loop (`loopEnd`), a compute operation (`compute`), a load DMA (`loadDMA`), a store DMA (`storeDMA`), or a DMA-wait operation (`waitDMA`). Each node encapsulates the information required to model compute latency and/or dependencies. For example, the `loopBegin` node contains the initial and final values as well as the step of the loop index variable. The addresses for the DMA nodes can be calculated from the loop index variables, base address given as an argument for the TOG, and statically determined tile sizes and strides. Compute nodes simply hold the number of cycles needed for the tensor tile operation, which is obtained using Gem5 and Spike (§3.8). The edges represent dependencies between the nodes.

The `waitDMA` node is introduced to express dependencies from a compute node to a preceding `loadDMA` node. By providing separate

`loadDMA` and `waitDMA` types, DMAs for tiles needed across multiple loop iterations can all be initiated before entering computation loops, such that compute latency can be overlapped with DMA for other tiles needed in subsequent iterations. By using the information from a TOG, our timing model (i.e., TOGSim) can properly model the DMA traffic and compute the latency of NPU cores with high simulation speed. To achieve high accuracy, the compute latencies for different components can be separately captured as needed, along with dependency information in the TOG. In our example model of Google TPU, we capture the information for vector and matrix units separately.

Since the information in the TOG is generic (e.g., compute latency, DMA addresses/stride, and loop begin/end) and expressed in the common ONNX format with minor customization, TOGs can be generated from other deep-learning compilers. Then, it can be executed in our TOGSim to exploit its NPU HW model, without relying on our compiler implementation if needed.

3.8 Simulator Implementation

For ILS, we modify two simulators, Gem5 and Spike, to model our NPU core with a scratchpad memory, VCIX interface, dataflow unit, and our ISA. We modified a pipelined in-order core of Gem5 and use it as a cycle-accurate timing model of the NPU core to obtain the compute node's latency in the TOG. To obtain this information with minimal overhead, the Gem5 executes a single iteration of the loop using the machine code, ignoring DMAs, to model the deterministic part of the compute latency.

We extend Spike to implement our NPU ISA (§3.4) for modeling the functional aspects of the NPU, using input tensors exported from the PyTorch runtime. The output from Spike can also be fed back into PyTorch, to execute some operations on the CPU or other devices such as GPUs (e.g., for result verification or to support operations not implemented on the target NPU).

In addition, if the tile compute latency is data-dependent (e.g., NPUs for sparse DNNs), our extended Spike can execute part of the kernel to obtain the latency for the dataflow unit offline. We assume data-dependent conditional vector operations are implemented as masked vector operations, making their latencies not data-dependent. Thus, they can be properly modeled with our Gem5 timing model. The latency for each tensor tile can be obtained by combining the latencies from both Gem5 and Spike provided in the TOG. After the latencies are obtained once offline, they can be reused over multiple simulations across different scenarios and HW configurations, to achieve high simulation speed.

The TLS is implemented by our TOGSim, which uses the TOG to model compute latency, generate DMAs, and track their dependencies. DMAs are modeled using cycle-accurate Ramulator 2 [78] and Booksim [59] models for the DRAM and interconnect.

3.9 Extending PyTorchSim

3.9.1 Other ML Frameworks. Besides PyTorch, TensorFlow [25] and JAX [38] are also widely used. Their frontends lower their computations to a shared subset of MLIR dialects such as `linalg`, `tensor`, `memref`, `vector` [83]. Given that PyTorchSim shares the same core MLIR dialects, extending it to support TensorFlow and JAX frontends requires only the addition of appropriate conversion passes in MLIR, including the 4D DMA support from PyTorchSim.

3.9.2 GPU Architectures. There are significant similarities between modern NPUs and GPUs, making it possible to model GPU's deep learning performance in PyTorchSim with proper modifications. For example, the SIMT execution model and Tensor cores are similar to the vector and matrix units of NPUs at a high level. For DNNs, memory accesses in GPUs are also typically done through SW-managed shared memory rather than L1 data caches [19], and recent GPU's Tensor Memory Accelerator is similar to DMA engines [3]. Thus, tile computation latencies can be predicted similarly to NPUs, although the concurrency among threadblocks should be carefully considered. The existing MLIR passes in PyTorchSim and PyTorch's Triton backend can be extended to generate TOGs from GPU kernels for TOGSim. We leave such extensions for future work.

3.9.3 Multi-NPU and Multi-Node Systems. For distributed inference and training, recent simulators such as LLMServingSim [45] and vTrain [34] use layer runtime profiles from external NPU simulators or real devices. They can be easily combined with PyTorchSim to model multi-NPU and multi-node systems. However, they rely on the assumption that different devices have similar layer or kernel execution time, which may not hold for dynamic execution scenarios (e.g, MoE with dynamic request distribution). To address such cases, PyTorchSim can be extended to instantiate multiple NPU models connected through the interconnect to model large systems. Simulation of the NPUs can be parallelized across multiple host CPU cores. Considering that data/tensor/pipeline/expert parallelization requires coarse-grained communication (e.g., all-reduce) between a large chunk of computation, synchronization across host CPU cores can be done infrequently with low overhead. We also leave such extensions of PyTorchSim for future work.

3.10 Load Generator and Scheduler

Our load generator (Fig. 1) takes the user's DNN request profile describing DNN model, input data, and request arrival time distribution. Requests are then generated and pushed to one or multiple request queues for single-user and multi-model scenarios. The scheduler can choose requests from the queues and create a batch of requests that use the same DNN. Depending on the batch size, the tensor size and tiling method can change. Thus, the backend compiler will be triggered for new batch sizes, and the compiled code and the TOG will be kept in a TOG cache such that it can be reused for later requests with the same batch size and DNN. To allocate memory for each kernel, we use PyTorch's memory allocator provided for GPUs [11]. For multi-model tenancy, NPU can be partitioned to associate each subset of cores with a different request queue. The scheduler implements different policies, including temporal sharing and spatial sharing, while maximizing batching.

4 Evaluation

4.1 Methodology

We evaluated PyTorchSim's accuracy and simulation speed with an NPU model for Google TPUv3 [87].[4] The TPUv3 has two cores

[4]TPUv4 and TPUv4i have limited availability while, for more recent TPUs, architectural details are not disclosed.

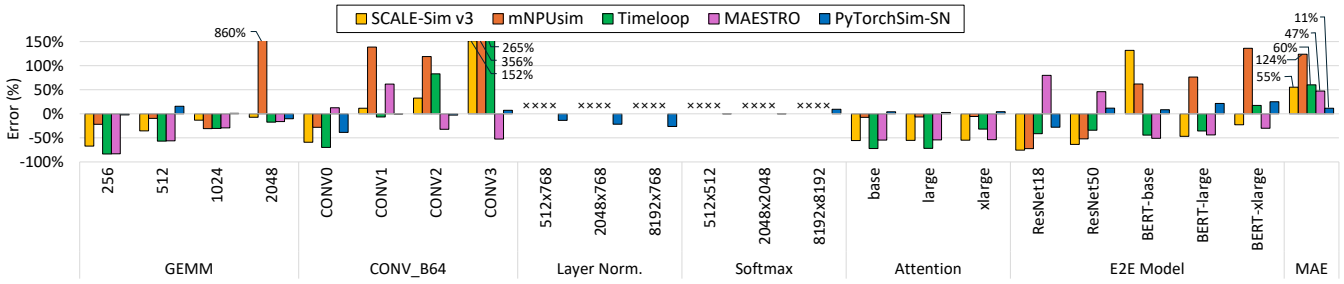

Figure 5: Simulation accuracy results for the TPUv3 configuration. Simulators other than PyTorchSim do not have LN and softmax results as they do not support these operations, and thus, these are excluded from MAE for fairness. The end-to-end results only include operations supported by each simulator (i.e., LN and softmax are included only for PyTorchSim).

(940 MHz) where each core has two 128x128 SAs, 128 vector units with 16 lanes per unit, and 16 MB of scratchpad memory. It has four HBM2 stacks with a total bandwidth of 960 GB/s. We assumed tCL, tRCD, tRAS, tWR, tRP timing parameters of 8, 8, 18, 8, and 8 ns, respectively. We assume a crossbar NoC with 256-bit flits.

For accuracy validation, we used only a single NPU core for the TPUv3 server NPU since TPUv3 cores are recognized as separate devices by the runtime, and scheduling workloads across them is not done transparently. Other simulations used all NPU cores unless otherwise stated. For TPU, the XLA compiler [4] was used to generate optimized code. For GEMM and CONV, we used TensorFlow and its profiler [10] to obtain execution traces with runtime. End-to-end models were profiled using PyTorch with `torch_xla` backend due to model code availability. We ran each kernel 100 times to obtain an average latency using the TensorBoard [15]. For other NPU simulators, we only considered GEMM, GEMV (from attention), and CONV operations as other operations were not supported. The functional correctness of PyTorchSim was validated by comparing its DNN output to that of a real CPU.

Simulators. We evaluated the simulation speed of PyTorchSim in comparison to two existing cycle-accurate simulators: Accel-Sim [65] and mNPUsim [57]. We excluded simulators that do not support multi-core NPU or contention among multiple cores to shared resources, as they are necessary for server-class NPUs. SMAUG was excluded as it did not support SAs larger than 8×8. While Accel-Sim is a GPU simulator, it was included for comparison because GPUs are widely used for deep learning, and it provides a more comprehensive set of features compared to others. Since GPUs do not have an SA, we adjusted their resources, such as the number of SMs, frequency, and cache size, to provide similar FLOPS and SRAM capacity as the target NPU. In the simulation speed comparison, analytical NPU models such as SCALE-Sim v3 [91], Timeloop [89], and MAESTRO [69] were not included, since they sacrifice accuracy (§4.2) while being faster than cycle-accurate simulators. The purpose of PyTorchSim is to complement such analytical models by providing high accuracy despite lower speed, rather than replacing them. PyTorchSim used the cycle-accurate Ramulator 2 [78] DRAM model and two different NoC models: a simple latency-bandwidth network model (`SN`) and the cycle-accurate Booksim network model [59] simulator (`CN`). PyTorchSim used TLS unless otherwise specified as `ILS`.

Workloads. We used multiple representative GEMM kernels denoted as GEMM(N), which operate on two square input matrices of size N×N. CONV0–3 represent commonly used convolution kernels with 3×3 filters, and 64, 128, 256, and 512 output channels, respectively. Each kernel processes an input feature map of size 56×56, 28×28, 14×14, and 7×7, with matching input and output channel dimensions. For full model evaluations, we used two variants of ResNet [56] and three different BERT [49, 71] models with 512-token input sequences. Simulation speed measurements were done on a single core of an AMD EPYC 7452 CPU, excluding the time required for input trace generation for Accel-Sim and compile time for PyTorchSim. For simulations taking 30 minutes or less, we report an average of five measurements. For all workloads, a batch size of one was used, unless otherwise stated.

4.2 Accuracy Validation

As shown in Fig. 5, PyTorchSim achieves significantly better accuracy than others by supporting various optimizations (§3.6.3), data transformations (e.g., transpose), and general vector operations. Overall, it achieved an 11.5% MAE (Mean Absolute Error) of runtime relative to the real TPUv3. Other simulators exhibited higher errors due to: (1) the lack of DRAM latency modeling, which is the primary source of errors in small GEMMs; (2) the absence of a vector unit, leading to insufficient support for layers such as pooling, skip connections, softmax, and batch/layer normalization; and (3) the failure to account for operation fusion. Consequently, these simulators significantly underestimated the end-to-end DNN runtime In addition, analytical models such as Timeloop and MAESTRO use simplified core models (e.g., compute latency calculated as the number of MAC operations divided by the number of PEs), ignoring other factors important for accurate modeling – such as structural hazards, core-tile dimension mismatch, or the degree of overlap among DMAs, inherent dataflow latency, and compute. Addressing the limitations requires a major redesign of the simulators.

For CONV0, PyTorchSim showed a slightly larger error because of TPU's unoptimized execution. XLA inserted an unnecessary data layout conversion before CONV0, incurring additional memory traffic, whereas this did not occur for other CONVs. PyTorchSim avoided this overhead by using the DMA unit to perform layout conversion on the fly. Because XLA is closed-source, we could not further analyze its behavior. In other words, the difference in

cycle counts arises from the real TPU's issue rather than PyTorchSim's. For CONV0 in ResNet18, PyTorchSim outperformed the TPU through our optimization described in §5.3. For other models, however, PyTorchSim exhibited higher runtime, as the TPU applied more aggressive operation fusion of vector and matrix operations. This limitation can be mitigated by adopting more fine-grained operation fusion.

Note that, beyond the basic DNN inference workloads shown here, there are other important scenarios—such as those presented in §5 with PyTorchSim—including DNN training and compiler optimization. For these scenarios, comparisons with existing simulators are not even possible due to the fundamental limitations of other simulators, such as the lack of integration with a compiler or an ML framework capable of automatic differentiation.

4.3 Simulation Speed

PyTorchSim achieved higher simulation speed across various workloads than other simulators. `PyTorchSim-SN` achieved up to 139.5× (47.9× on average) higher speed over Accel-Sim (Fig. 6). These improvements are achieved through the TLS by using offline tile operation latencies, overcoming the low speed of ILS; as the TLS achieved a speedup of 8.97× over ILS on average. Even `PyTorchSim-CN` achieved significant speedups of up to 18.2× (average 10.22×) compared to Accel-Sim, as cycle-by-cycle simulation is done only for shared resources, while the NPU cores are modeled at high speed with TLS. The detailed NoC simulation may not be necessary when the interconnect bandwidth is abundant, but it becomes important when the bandwidth is constrained (e.g., chiplet interconnect).

The end-to-end execution of ResNet-50, BERT-Base, and BERT-Large took 25-70 seconds for a single input data using PyTorchSim thanks to its high speed. Compared to the real TPUv3 machine, it is 60,000–95,000× slower, but it still significantly mitigates the practicality challenges of using other, even slower simulators. Thus, design space exploration for various NPU configurations can be done more easily, by running different workloads end-to-end.

Compared to mNPUsim, PyTorchSim achieved higher simulation speedups as the matrix size increased for GEMMs. The speed degradation for mNPUsim was caused by file-based intermediate data storage for memory access addresses in mNPUsim, which leads to frequent filesystem access with significant overhead. Moreover, mNPUsim only supports matrix multiplication and convolution operations, lacking support for tensor operations such as batch normalization and softmax, which are necessary for accurately modeling the end-to-end execution of DNNs on NPUs.

5 Case Studies

We demonstrate the versatility of PyTorchSim with various case studies. **Note that none of them are supported by any other simulators in Table 1, as we explain below.** While Accel-Sim can support case studies similar to those in §5.3, §5.4, and §5.5, it is slow (Fig. 6) and is a GPU simulator that depends on NVIDIA's closed-source compiler, thereby limiting its applicability for exploring diverse NPU HW and SW designs. We present validation results when possible (e.g., if the real Google TPU supports the scenarios).

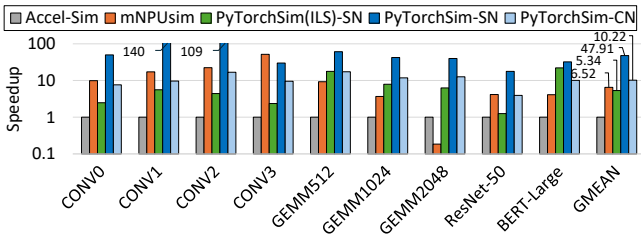

Figure 6: Simulation speedups, including PyTorchSim(ILS).

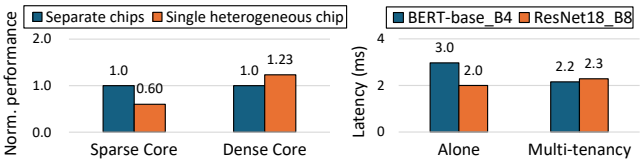

Figure 7: (a) Impact of integrating a dense and sparse matrix units in a single NPU over employing them in separate chips. (b) Impact of multi-model tenancy on the inference latency for BERT-base and ResNet-18.

5.1 Heterogeneous Dense-Sparse NPU

Here, we study the performance interference between a dense, SA core and a sparse, Flexagon core [84] integrated in a heterogeneous NPU. To support it, we implemented the lowering pass for PyTorch's sparse GEMM operator [22]. The baseline NPUs have either a dense core (running GEMM) or a sparse core (running SpMSpM), with 240 GB/s HBM, while the heterogeneous NPU has both cores with 480 GB/s HBM. In the heterogeneous NPU, the sparse core performance was significantly degraded by 40% while the dense core had a 23% speedup due to DRAM contention (Fig. 7a). The FR-FCFS memory scheduler we assumed favored the regular memory accesses from the SA, starving the sparse core with low memory access locality. Thus, the memory controller needs to be carefully designed to prevent such significant unfairness. For validation, we compared PyTorchSim extended with the Flexagon core model from SST-STONNE [5] against the original SST-STONNE. We evaluated SpMSpM256 and SpMSpM512 with 95% sparsity using outer product dataflow on both simulators with a simple 100 ns DRAM latency model. PyTorchSim achieved cycle errors of only 1.1-2.6% against the original SST-STONNE while achieving 16.5-27.4× speedups, thanks to the TLS.

Other NPU simulators' limitations: STONNE lacks multi-core support. Sparseloop lacks cycle accuracy. Others lack support for sparse core or only support structured sparsity.

5.2 DNN Inference with Multi-model Tenancy

This case study illustrates the impact of co-locating inference workloads of BERT-base (batch size 4) and ResNet-18 (batch size 8) on a single NPU. When executed independently, each model was allocated half of the total DRAM bandwidth, whereas in the co-located setting, both models shared the entire bandwidth. Co-location resulted in a 15% increase in inference latency for ResNet-18, while BERT-base experienced a 28% reduction in latency (Fig. 7b). When

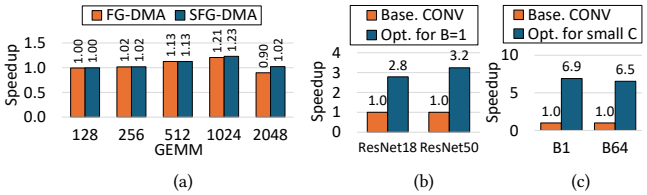

Figure 8: Speedups from optimizations: (a) improved DMA–compute overlap (using fine-grained DMAs) for GEMMs compared to coarse-grained DMA, (b) ResNets with a batch size of 1, and (c) CONVs with small input channels (for batch sizes of 1 and 64).

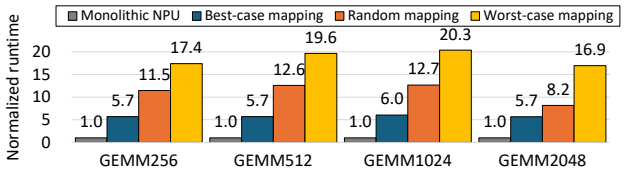

Figure 9: Normalized runtime for GEMM operations on a chiplet-based NPU under different weight tensor mapping choices.

running alone, ResNet-18 utilized only 39% of the allocated DRAM bandwidth, whereas BERT-base consumed up to 63%. This indicates that ResNet-18 is less bandwidth-intensive than BERT-base. The average memory bandwidth of ResNet18 decreased from 188 GB/s to 165 GB/s, while that of BERT-base increased from 303 GB/s to 424 GB/s. Since the later layers of ResNet have a lower memory footprint and intensity, it can unfairly suffer from the bandwidth contention. Under co-located execution, BERT-base effectively exploited the additional bandwidth that ResNet-18 did not fully utilize. As a result, BERT-base achieved lower latency by opportunistically utilizing DRAM bandwidth, whereas ResNet-18 experienced higher latency due to memory contention.

Other NPU simulators' limitations: mNPUsim supports multi-model tenancy but does not support batch sizes larger than 1, whereas the others do not support multi-model tenancy.

5.3 Compiler Optimization Impact

We show the impact of compiler optimizations (§3.6.3) to use fine-grained DMAs and CONV optimizations for a batch size of 1 or small number of input channels in Fig. 8. Fine-grained DMA (FG-DMA) achieved up to 23% higher throughput compared to coarse-grained DMA by improving compute-DMA overlap (Fig. 8a). The gains from overlap tend to increase with matrix size due to higher SA utilization. However, for GEMM(2048), FG-DMA underperformed because of its overhead: to overlap compute and DMA, inputs and weights must be loaded alternately, which reduces row buffer locality. Thus, with the Selective Fine-Grained DMA (SFG-DMA), fine-grained DMA was disabled for the large tensor, resulting in performance recovery for GEMM(2048). CONV optimizations shown in Fig. 8b-c address reduced SA utilization due to smaller tile sizes caused by the small batch size or input channel count: different tiling strategies were used to increase the tile size in the width dimension. As a result, significant speedups of 2.8-6.9× were achieved by the optimization.

Other NPU simulators' limitations: GeneSys supports a compiler but not the multi-core NPUs we assume. Others lack compilers.

5.4 Scheduling for Chiplet-based NPUs

We studied the impact of workload scheduling on a chiplet-based, multi-core NPU with NUMA (Non-Uniform Memory Access). The system consists of two chiplets, each integrating one core and one HBM stack, connected through an off-chip link with 64 GB/s bandwidth and 20 ns latency. Input and weight are partitioned horizontally into I_0, I_1 and vertically into W_0, W_1 and mapped to each chiplet, to compute: $[I_0, I_1]^T \times [W_0, W_1] = \begin{pmatrix} O_{00} & O_{01} \\ O_{10} & O_{11} \end{pmatrix}$. Depending on the mapping strategy, a core's memory accesses consist of varying proportions of local memory access (480 GB/s bandwidth) and remote memory access (limited by the 64 GB/s off-chip link).

Best-case mapping: each core computes outputs that primarily reuse its own local input and weight partitions, resulting in ~75% local and ~25% remote traffic.

Worst-case mapping: each core works mostly on partitions stored on the other chiplet, leading to ~25% local and ~75% remote traffic.

Random mapping: produces a balanced case of ~50% local and ~50% remote traffic.

Assuming the operation is memory-bound, runtime is inversely proportional to the *effective bandwidth*, which we estimate as the weighted harmonic mean of local and remote bandwidths:

$$BW_{\text{eff}} = \left(\frac{\alpha}{BW_{\text{local}}} + \frac{1-\alpha}{BW_{\text{remote}}} \right)^{-1},$$

where α is the fraction of local traffic. The monolithic NPU (all local) attains $BW_{\text{eff}} \approx 960$ GB/s (normalized runtime = 1.0). As the ratio of remote traffic increases, effective bandwidth decreases and runtime grows: the best-case mapping (75% local, 25% remote) yields $BW_{\text{eff}} \approx 183$ GB/s (normalized runtime $\approx$ 5.3), random mapping (50%/50%) gives 113 GB/s ($\approx$ 8.5), and the worst-case mapping (25%/75%) results in 82 GB/s ($\approx$ 11.8).

These estimates closely align with the trend shown in Fig. 9, showing up to 6.0× slowdown in the best case and 20.3× in the worst case relative to the monolithic design. The greater slowdown from the simulation result (e.g., 20.3× slowdown from simulation vs. ~ 11.8× slowdown from analysis for the worse-case mapping) can be attributed to the fact that the off-chip link bandwidth is only 32 GB/s in each direction whereas the local HBM bandwidth can be fully utilized for both reads and writes due to the differences in the interface. The degradation clearly results of the limited off-chip link bandwidth when remote accesses dominate. This analysis illustrates how workload-to-chiplet mapping directly impacts effective bandwidth and performance. Other scenarios with contention for shared resources can be similarly modeled, and mitigation techniques for them can be evaluated using the cycle-accurate models for the resources.

Other NPU simulators' limitations: SMAUG and STONNE do not support large SA and multi-core NPUs, respectively. Others do not support a cycle-accurate interconnect model. Even if mNPUsim is extended to support it, it has lower simulation speed and accuracy.

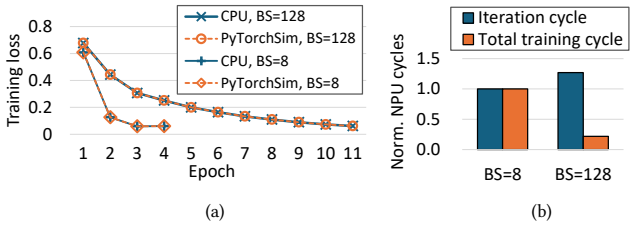

Figure 10: (a) Training loss curves from MLP training with different batch sizes (BS). (b) Normalized per-iteration runtime and total training time.

5.5 Impact of DNN Training Hyperparameter

As PyTorchSim supports simulation of DNN training on an NPU using tensor data from PyTorch runtime on the host (§3.8), we show a case study on the impact of the batch size hyperparameter on training loss convergence and NPU performance. For simplicity, we trained a multi-layer perceptron with an input size of 28×28 and a hidden dimension of 256 using the MNIST dataset [48].

Fig. 10 shows that the training loss curves from PyTorchSim are identical to those from a real CPU, validating the correctness of the compiled code. In this case, the small batch training converged with a smaller number of epochs. However, the larger batch size finished training 4.6× faster in cycles due to fewer iterations per epoch, even though per-iteration runtime increased by 21%. Meanwhile, the large batch training decreased the model accuracy by 7.8% compared to the small batch training. This result shows that when training with large batch sizes, other hyperparameters, such as learning rate, may need to be adjusted adaptively, as suggested by prior work [60]. As demonstrated here, PyTorchSim can be used to design performance-aware training methods. It can also support other training scenarios, such as studying the impact of modifying the model architecture, NPU scheduler, and/or its microarchitecture. While simulators are slower than real machines, the simulations do not have to start from scratch; by exploiting PyTorch's model checkpoint support, they can begin training with checkpoints from a real machine. It is also possible to parallelize the simulation of different training iterations by preparing their checkpoints beforehand and launching multiple simulations that begin at different checkpoints, using multiple cores of the CPU used to run the simulator.

We also validated the simulated NPU cycles against real TPUv3 and obtained an error rate of only -8% (3%) for a batch size of 8 (128), demonstrating the high accuracy of our timing model.

Other NPU simulators' limitations: No other NPU simulator supports DNN training.

6 Related Work

6.1 NPU Simulators

As discussed in §2.1, existing simulators do not comprehensively support features that are necessary in various NPU architecture evaluation scenarios. FPGA-based NPU implementations, such as Gemmini [54] and Tandem Processor [55], enable high-speed evaluation and high accuracy. However, they cannot model server-class NPUs that do not fit in a single FPGA chip due to constraints in logic capacity and on-chip resources. While enterprise-level FPGA prototyping systems [9] can validate large RTL designs, their accessibility outside large institutions is limited. In addition, they cannot be used for early-stage design space exploration when RTL implementation is not yet available.

LLMServingSim [45], vTrain [34], and Vidur [28] specifically focus on the simulation of LLMs in large-scale distributed systems and are limited in supporting various other DNNs. These simulators rely on a coarse NPU model or profiling results from real GPUs to obtain layer execution time, but they can be extended to use simulated NPU execution time from PyTorchSim.

6.2 Simulators for CPUs and GPUs

For high-speed simulation, some CPU simulators such as Sniper [40], ZSim [41], and McSimA+ [29] use the Pin dynamic binary instrumentation tool to use real CPUs to model the functional aspect of the architecture. However, such an approach is not applicable to NPUs, as they do not support dynamic instrumentation. In addition, while ZSim's separation of the compute latency calculation ("bound" phase) and contention modeling ("weave" phase) has similarities with our approach, we avoid its runtime overhead from ILS in the bound phase by obtaining the compute-memory dependence information offline. Even for data-dependent (sparse) operations, we obtain per-tile compute latency offline to amortize its overhead over multiple simulations. Chakravarty et al. [42] proposed a compiler-based back-annotation method at the basic block granularity for fast performance and power simulation, similar to our TLS. However, their technique does not allow basic blocks to have data-dependent execution latency while control flow is considered. Thus, in contrast to PyTorchSim, it cannot be used to model sparse tensor operation latency, where the same instruction that triggers dataflow unit operations can take varying latency depending on the sparsity of each input tile.

GPU simulators, such as Accel-Sim [65], MGPUSIM [97], and NaviSim [35], can be useful as alternatives to NPU simulators in evaluating systems for deep learning, as GPUs are currently widely used for DNN acceleration. GPU vendors also support SW frameworks for DNN acceleration (e.g., cuDNN [44] and MIOpen [66]). However, their instruction-level simulation approach results in significant simulation speed limitations. In addition, MGPUSim and NaviSim do not support instructions for the Matrix (or Tensor) Core, which accelerates GEMM and is important for modeling deep learning performance of modern GPUs. NVArchSim [101] is an industrial GPU simulator that achieves high speed by carefully selecting the fidelity of the components being modeled, but it is not publicly available. Vortex [99] is an open-source, FPGA-based GPU design, but its support for DNNs is currently limited. VTrain [34] enables simulation of a large GPU cluster for DNN training based on kernel execution profiles from real GPUs. Thus, it cannot be used to evaluate the impact of GPU's architectural changes, but it can be combined with PyTorchSim to model a large NPU cluster.

7 Conclusion

In this work, we propose PyTorchsim, a comprehensive, cycle-accurate NPU simulation framework with high speed. We model a generic and extensible NPU architecture based on RISC-V vector extension and provide an NPU compiler backend for PyTorch 2 to

enable simulation of existing DNNs implemented in PyTorch without DNN format conversion. A functional simulator is also included to verify the compiled code and support data-dependent timing simulation. The compiler also enables simulation of DNN training in addition to inference, in contrast to existing NPU simulators that only support inference. Furthermore, to support various scenarios, including multi-tenancy, we support multi-core NPUs with cycle-accurate DRAM and interconnect models. To achieve these features, we combine and extend different simulators, but this can lead to slow simulation speed due to instruction-level simulation. Thus, we propose tile-level simulation (TLS) to enable high-speed simulation by introducing a tile operation graph, which represents a DNN as a graph of tile-granularity DMA and compute nodes, with edges capturing their dependencies. TLS is based on the observation that NPUs perform computations on tensor tiles with *deterministic* latencies after they are made available through DMAs. Our proposed TOGSim thus executes the TOGs at high speed while the *non-deterministic* DMA latencies are modeled by cycle-accurate simulation online. We demonstrate that TLS even extends to data-dependent sparse NPU core models. Our evaluation results show that PyTorchSim can accurately model Google TPUv3 with an MAE of 11.5%, while achieving significant simulation speedups of up to 139× over Accel-Sim. Further, our sparse NPU core simulation achieved up to 27× speedup with 2.6% cycle count error against the existing target model. Our case studies also demonstrate that PyTorchSim enables various design space exploration studies, such as heterogeneous dense-sparse NPU design, multi-model tenancy, compiler optimization, chiplet-aware NPU scheduling, and impact of DNN training hyperparameters.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (No. RS-2024-00415602, RS-2023-00277080, RS-2025-00521761, RS-2024-00347786), and by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (No. RS-2019-II191906, Artificial Intelligence Graduate School Program (POSTECH)) funded by MSIT. It was also supported by the MSIT under the ITRC program (IITP_2024-RS-2024-00437866, 16.7%) supervised by IITP, and with Cloud TPUs from Google's TPU Research Cloud (TRC).

References

[1] 2019. Spike RISC-V ISA Simulator. GitHub repository. https://github.com/riscv-software-src/riscv-isa-sim.
[2] 2021. RISC-V "V" Vector Extension Version 1.0. RISC-V International. https://github.com/riscvarchive/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf
[3] 2022. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-data-center-overview/gtc22-whitepaper-hopper.
[4] 2023. OpenXLA Project. Google. https://openxla.org/xla
[5] 2023. SST-STONNE simulator. [Online]. Available: https://github.com/stonne-simulator/sst-elements-with-stonne.
[6] 2024. How the PolyBlocks AI Compiler Works. PolyMage Labs. [Online]. Available: https://docs.polymagelabs.com/articles/polyblocks-quantization.html.
[7] 2024. Inferentia Architecture. AWS Neuron Documentation. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-hardware/inferentia.html
[8] 2024. Introduction to ONNX. ONNX 0.19.0 documentation. [Online]. Available: https://onnx.ai/onnx/intro/.
[9] 2024. Meeting the challenge of concurrent RTL and workload verification and validation. SIEMENS whitepaper.
[10] 2024. Optimize TensorFlow performance using the Profiler. Google. [Online]. Available: https://www.tensorflow.org/guide/profiler.
[11] 2024. PyTorch Documentation (2.4) - CUDA semantics (Memory Management). https://pytorch.org/docs/stable/notes/cuda.html#cuda-memory-management
[12] 2024. PyTorch Documentation (2.4) - torch.compile. https://pytorch.org/docs/stable/generated/torch.compile.html
[13] 2024. RISC-V Compiler Infrastructure and Toolchain. https://github.com/riscv-software-src. [Accessed 21-06-2025].
[14] 2024. SiFive Vector Coprocessor Interface (VCIX) Software Specification Version 1.1. [Online]. Available: https://www.sifive.com/document-file/sifive-vector-coprocessor-interface-vcix-software.
[15] 2024. TensorBoard: TensorFlow's visualization toolkit. Google. [Online]. Available: https://www.tensorflow.org/tensorboard.
[16] 2024. Trainium Architecture. AWS Neuron Documentation. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-hardware/trainium.html
[17] 2025. Apple Watch Series 10. [Online]. Available: https://www.apple.com/apple-watch-series-10/specs/.
[18] 2025. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ Section on PTX and SASS.
[19] 2025. Efficient GEMM in CUDA. NVIDIA CUTLASS Documentation. https://docs.nvidia.com/cutlass/media/docs/cpp/efficient_gemm.html. [Accessed 20-06-2025].
[20] 2025. Gemmini. GitHub repository. https://github.com/ucb-bar/gemmini.
[21] 2025. Papers With Code : Trends. Meta AI. https://paperswithcode.com/trends. [Accessed 16 June 2025].
[22] 2025. PyTorch Documentation – torch.sparse.mm. The Linux Foundation. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.sparse.mm.html.
[23] 2025. PyTorch2 AsyncCompile. [Online]. Available: https://github.com/pytorch/pytorch/blob/main/torch/_inductor/async_compile.py.
[24] 2025. "RDNA4" Instruction Set Architecture Reference Guide. AMD. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna4-instruction-set-architecture.pdf.
[25] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.
[26] Dennis Abts, Garrin Kimmell, Andrew Ling, John Kim, Matt Boyd, Andrew Bitar, Sahil Parmar, Ibrahim Ahmed, Roberto DiCecco, David Han, John Thompson, Michael Bye, Jennifer Hwang, Jeremy Fowers, Peter Lillian, Ashwin Murthy, Elyas Mehtabuddin, Chetan Tekur, Thomas Sohmers, Kris Kang, Stephen Maresh, and Jonathan Ross. 2022. A software-defined tensor streaming multiprocessor for large-scale machine learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 567–580. https://doi.org/10.1145/3470496.3527405
[27] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think fast: a tensor streaming processor (TSP) for accelerating deep learning workloads. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 145–158. https://doi.org/10.1109/ISCA45697.2020.00023
[28] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. VIDUR: A LARGE-SCALE SIMULATION FRAMEWORK FOR LLM INFERENCE. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 351–366. https://proceedings.mlsys.org/paper_files/paper/2024/file/b74a8de47d2b3c928360e0a011f48351-Paper-Conference.pdf
[29] Jung Ho Ahn, Sheng Li, Seongil O, and Norman P. Jouppi. 2013. McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 74–85. https://doi.org/10.1109/ISPASS.2013.6557148
[30] Ayaz Akram and Lina Sawalha. 2016. x86 computer architecture simulators: A comparative study. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 638–645.
[31] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano,

Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. 929–947.

[32] Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer* 35, 2 (2002), 59–67.

[33] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[34] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. 2024. vTrain: A Simulation Framework for Evaluating Cost-effective and Compute-optimal Large Language Model Training. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*.

[35] Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L. Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. 2022. NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 333–345. https://doi.org/10.1145/3559009.3569666

[36] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[37] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[38] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

[39] Jingwei Cai, Zuotong Wu, Sen Peng, Yuchen Wei, Zhanhong Tan, Guiming Shi, Mingyu Gao, and Kaisheng Ma. 2024. Gemini: Mapping and Architecture Co-exploration for Large-scale DNN Chiplet Accelerators. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 156–171. https://doi.org/10.1109/HPCA57654.2024.00022

[40] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Trans. Archit. Code Optim.* 11, 3, Article 28 (Aug. 2014), 25 pages. https://doi.org/10.1145/2629677

[41] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Trans. Archit. Code Optim.* 11, 3, Article 28 (Aug. 2014), 25 pages. https://doi.org/10.1145/2629677

[42] Suhas Chakravarty, Zhuoran Zhao, and Andreas Gerstlauer. 2013. Automated, retargetable back-annotation for host compiled performance and power modeling. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–10. https://doi.org/10.1109/CODES-ISSS.2013.6659023

[43] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[44] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[45] Jaehong Cho, Minsu Kim, Hyunmin Choi, Guseul Heo, and Jongse Park. 2024. LLMServingSim: A HW/SW Co-Simulation Infrastructure for LLM Inference Serving at Scale. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*.

[46] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

[47] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[48] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[49] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[50] Dave Ditzel. 2024. Why Esperanto picked RISC-V for HPC Computing. International workshop on RISC-V for HPC (RISCV-HPC) at SC24. [Online]. Available: https://github.com/RISCVtestbed/riscvtestbed.github.io/blob/main/assets/files/sc24/Ditzel.pdf.

[51] Roger Espasa. 2024. SemiDynamics: Out-of-Order RISC-V Cores, Now with Vector! https://riscv-europe.org/summit/2024/media/proceedings/plenary/Tue-09-40-Roger-Espasa.pdf. Plenary Talk, RISC-V Summit Europe 2024.

[52] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, Adam Hutchin, Utku Diril, Krishnakumar Nair, Ehsan K. Aredestani, Martin Schatz, Yuchen Hao, Rakesh Komuravelli, Kunming Ho, Sameer Abu Asal, Joe Shajrawi, Kevin Quinn, Nagesh Sreedhara, Pankaj Kansal, Willie Wei, Dheepak Jayaraman, Linda Cheng, Pritam Chopda, Eric Wang, Ajay Bikumandla, Arun Karthik Sengottuvel, Krishna Thottempudi, Ashwin Narasimha, Brian Dodds, Cao Gao, Jiyuan Zhang, Mohammed Al-Sanabani, Ana Zehtabioskuie, Jordan Fix, Hangchen Yu, Richard Li, Kaustubh Gondkar, Jack Montgomery, Mike Tsai, Saritha Dwarakapuram, Sanjay Desai, Nili Avidan, Poorvaja Ramani, Karthik Narayanan, Ajit Mathews, Sethu Gopal, Maxim Naumov, Vijay Rao, Krishna Noru, Harikrishna Reddy, Prahlad Venkatapuram, and Alexis Bjorlin. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Article 80, 13 pages. https://doi.org/10.1145/3579371.3589348

[53] gem5 Project. 2021. gem5 21.2 Release. https://www.gem5.org/project/2021/12/28/gem5-21-2.html. [Accessed 21-06-2025].

[54] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE Press, 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[55] Soroush Ghodrati, Sean Kinzer, Hanyang Xu, Rohan Mahapatra, Yoonsung Kim, Byung Hoon Ahn, Dong Kai Wang, Lavanya Karthikeyan, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, and Hadi Esmaeilzadeh. 2024. Tandem Processor: Grappling with Emerging Operators in Neural Networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. 1165–1182. https://doi.org/10.1145/3620665.3640365

[56] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[57] Soojin Hwang, Sunho Lee, Jungwoo Kim, Hongbeen Kim, and Jaehyuk Huh. 2023. mNPUsim: Evaluating the Effect of Sharing Resources in Multi-core NPUs. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. 167–179. https://doi.org/10.1109/IISWC59245.2023.00018

[58] Intel Corporation. 2020. Intel® Advanced Vector Extensions 512 (Intel® AVX-512). https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html. [Accessed 21-06-2025].

[59] Nan Jiang, Daniel U. Becker, George Michelogiannakis, James Balfour, Brian Towles, D. E. Shaw, John Kim, and William J. Dally. 2013. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 86–96. https://doi.org/10.1109/ISPASS.2013.6557149

[60] Tyler Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. 2020. AdaScale SGD: A User-Friendly Algorithm for Distributed Training. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 4911–4920. https://proceedings.mlr.press/v119/johnson20a.html

[61] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Article 82, 14 pages. https://doi.org/10.1145/3579371.3589350

[62] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA52012.2021.00010

[63] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. 1–12. https://doi.org/10.1145/3079856.3080246

[64] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: a deep learning optimization framework for versatile GPU workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 190–205. https://doi.org/10.1145/3453483.3454038

[65] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[66] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasilii Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, and Mayank Daga. 2019. MIOpen: An Open Source Library For Deep Learning Primitives. *CoRR* abs/1910.00078 (2019). arXiv:1910.00078 http://arxiv.org/abs/1910.00078

[67] Hanjoon Kim, Younggeun Choi, Junyoung Park, Byeongwook Bae, Hyunmin Jeong, Sang Min Lee, Jeseung Yeon, Minho Kim, Changjae Park, Boncheol Gu, Changman Lee, Jaeick Bae, SungGyeong Bae, Yojung Cha, Wooyoung Choe, Jonguk Choi, Juho Ha, Hyuck Han, Namoh Hwang, Seokha Hwang, Kiseok Jang, Haechan Je, Hojin Jeon, Jaewoo Jeon, Hyunjun Jeong, Yeonsu Jung, Dongok Kang, Hyewon Kim, Minjae Kim, Muhwan Kim, Sewon Kim, Suhyung Kim, Won Kim, Yong Kim, Youngsik Kim, Younki Ku, Jeong Ki Lee, Juyun Lee, Kyungjae Lee, Seokho Lee, Minwoo Noh, Hyuntaek Oh, Gyunghee Park, Sanguk Park, Jimin Seo, Jungyoung Seong, June Paik, Nuno P. Lopes, and Sungjoo Yoo. 2024. TCP: A Tensor Contraction Processor for AI Workloads Industrial Product*. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 890–902. https://doi.org/10.1109/ISCA59077.2024.00069

[68] Seah Kim, Hyoukjun Kwon, Jinook Song, Jihyuck Jo, Yu-Hsin Chen, Liangzhen Lai, and Vikas Chandra. 2024. DREAM: A Dynamic Scheduler for Dynamic Real-time Multi-model ML Workloads. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 73–86. https://doi.org/10.1145/3623278.3624753

[69] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*. ACM, 754–768.

[70] William Lacy, Gregory Michael Thorson, Christopher Aaron Clark, Norman Paul Jouppi, Thomas Norrie, and Andrew Everett Phelps. 2022. Vector Processing Unit. U.S Patent 11520581.

[71] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. arXiv:1909.11942 [cs.CL] https://arxiv.org/abs/1909.11942

[72] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO 2004)*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[73] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[74] Sean Lie. 2023. Cerebras Architecture Deep Dive: First Look Inside the Hardware/Software Co-Design for Deep Learning. *IEEE Micro* 43, 3 (2023), 18–30. https://doi.org/10.1109/MM.2023.3256384

[75] Jason Lowe-Power. 2023. gem5 Version 23.1 Release. gem5 blog. https://www.gem5.org/project/2023/12/21/gem5-23-1.html

[76] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *CoRR* abs/2007.03152 (2020). arXiv:2007.03152 https://arxiv.org/abs/2007.03152

[77] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[78] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, and Onur Mutlu. 2023. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator. *IEEE Comput. Archit. Lett.* 23, 1 (Nov. 2023), 112–116. https://doi.org/10.1109/LCA.2023.3333759

[79] Sheng Ma, Yuanwu Lei, Libo Huang, and Zhiying Wang. 2019. MT-DMA: A DMA Controller Supporting Efficient Matrix Transposition for Digital Signal Processing. *IEEE Access* 7 (2019), 5808–5818. https://doi.org/10.1109/ACCESS.2018.2889558

[80] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News* 33, 4 (2005), 92–99.

[81] MLCommons. 2025. Reference implementations of MLPerf™ Inference benchmarks (v5.1). https://github.com/mlcommons/inference. [Accessed 16 June 2025].

[82] MLCommons. 2025. Reference implementations of MLPerf™ Training benchmarks (v5.0). https://github.com/mlcommons/training. [Accessed 16 June 2025].

[83] Rolf Morel. 2025. MLIR Tensor Compiler: Design Group and Charter Update. https://llvm.org/devmtg/2025-04/slides/quick_talk/morel_mlir_tensor.pdf. LLVM Developers' Meeting, April 2025. Quick Talk..

[84] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 252–265.

[85] Francisco Muñoz-Martínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2021. STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators. In *IEEE International Symposium on Workload Characterization (IISWC)*. 201–213. https://doi.org/10.1109/IISWC53511.2021.00028

[86] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. 883–898. https://doi.org/10.1145/3453483.3454083

[87] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63. https://doi.org/10.1109/MM.2021.3058217

[88] Ryan O'Connor. [n. d.]. PyTorch vs TensorFlow in 2023. AssemblyAI Blog. https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/

[89] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. https://doi.org/10.1109/ISPASS.2019.00042

[90] Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, Mingran Wang, Xiangyu Song, Kejie Zhang, Tianren Gao, Angela Wang, Karen Li, Yongning Sheng, Joshua Brot, Denis Sokolov, Apurv Vivek, Calvin Leung, Arjun Sabnis, Jiayu Bai, Tuowen Zhao, Mark Gottscho, David Jackson, Mark Luttrell, Manish K. Shah, Edison Chen, Kaizhao Liang, Swayambhoo Jain, Urmish Thakker, Dawei Huang, Sumti Jairath, Kevin J. Brown, and Kunle Olukotun. 2024. SambaNova SN40L: Scaling the AI Memory Wall with Dataflow and Composition of Experts. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*.

[91] Ritik Raj, Sarbartha Banerjee, Nikhil Chandra, Zishen Wan, Jianming Tong, Ananda Samajdhar, and Tushar Krishna. 2025. SCALE-Sim v3: A modular cycle-accurate systolic accelerator simulator for end-to-end system analysis. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 186–200.

[92] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. In *Proceedings of the Fifth Conference on Machine Learning and Systems, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*.

[93] RISC-V International. 2024. RISC-V Ecosystem Landscape. https://landscape.riscv.org/. [Accessed 21-06-2025].

[94] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 58–68. https://doi.org/10.1109/ISPASS48437.2020.00016

[95] Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. 2021. SLO-Aware Inference Scheduler for Heterogeneous Processors in Edge Platforms. *ACM Trans. Archit. Code Optim.* 18, 4, Article 43 (July 2021), 26 pages. https://doi.org/10.1145/3460352

[96] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3352460.3358302

[97] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 197–209.

[98] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[99] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 754–766. https://doi.org/10.1145/3466752.3480128

[100] Jasmina Vasiljevic and Davor Capalija. 2024. Blackhole & TT-Metalium: The Standalone AI Computer and its Programming Model. In *2024 IEEE Hot Chips 36 Symposium (HCS)*. 1–30. https://doi.org/10.1109/HCS61935.2024.10664810

[101] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 868–880. https://doi.org/10.1109/HPCA51647.2021.00077

[102] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[103] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2023. Sparseloop: An Analytical Approach to Sparse Tensor Accelerator Modeling. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois, USA) *(MICRO '22)*. IEEE Press, 1377–1395. https://doi.org/10.1109/MICRO56248.2022.00096

[104] Sam (Likun) Xi, Yuan Yao, Kshitij Bhardwaj, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2020. SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads. *ACM Trans. Archit. Code Optim.* 17, 4, Article 39 (nov 2020), 26 pages. https://doi.org/10.1145/3424669

[105] Yuqi Xue, Yiqi Liu, Lifeng Nai, and Jian Huang. 2023. V10: Hardware-Assisted NPU Multi-tenancy for Improved Resource Utilization and Fairness. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 24, 15 pages. https://doi.org/10.1145/3579371.3589059

[106] Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Huazuo Gao, Jiashi Li, Liyue Zhang, Panpan Huang, Shangyan Zhou, Shirong Ma, Wenfeng Liang, Ying He, Yuqing Wang, Yuxuan Liu, and Y. Wei. 2025. Insights into DeepSeek-V3: Scaling Challenges and Reflections on Hardware for AI Architectures. In *Proceedings of the ACM/IEEE 52nd Annual International Symposium on Computer Architecture*. https://arxiv.org/abs/2505.09343v1

[107] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators . In *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 214–225. https://doi.org/10.1109/IISWC53511.2021.00029

[108] Donglin Zhuang, Zhen Zheng, Haojun Xia, Xiafei Qiu, Junjie Bai, Wei Lin, and Shuaiwen Leon Song. 2024. MonoNN: Enabling a New Monolithic Optimization Space for Neural Network Inference Tasks on Modern GPU-Centric Architectures. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 989–1005. https://www.usenix.org/conference/osdi24/presentation/zhuang

A Artifact Appendix

A.1 Abstract

This artifact provides all necessary components to reproduce the key results of the paper, including software, workloads, and execution scripts. It contains the complete PyTorchSim framework, which consists of a custom PyTorch 2.0 backend for MLIR code generation, a set of MLIR transformation passes, and simulation tools such as TOGSim, Spike, and Gem5.

The artifact includes representative workloads, ranging from synthetic GEMM and convolution kernels to full models like BERT and ResNet. To reproduce the simulation results (i.e., Fig. 5 and Fig. 6), it also provides scripts, configuration files, and detailed instructions. The entire workflow is containerized for a Docker-based x86 environment, with a prebuilt Docker image and scripts to run experiments inside the container.

A.2 Artifact check-list (meta-information)

- **Program:** TOGSim implemented in C++, integrated with Ramulator 2 (DRAM simulator) and BookSim (interconnection network simulator); Custom PyTorch 2.0 backend for MLIR code generation implemented in Python; MLIR-based IR transformation (custom passes), customized Spike (functional simulator), and customized Gem5 (timing simulator)
- **Compilation:** CMake 3.26.4, G++ 11.4.0
- **Transformations:** MLIR (custom passes), custom PyTorch 2.0 backend for MLIR code generation
- **Binary:** Simulator (TOGSim), mlir-opt (with custom passes), spike, gem5-opt
- **Model:** Synthetic kernels of varying sizes, including GEMM, convolution, layernorm, and softmax; and end-to-end models such as BERT and ResNet
- **Data Set:** Synthetic data sets
- **Run-Time Environment:** Docker 20.10.7, Ubuntu 22.04.3, Python 3.10.3, PyTorch 2.2.0, CMake 3.26.4, risc-v64-unknown-elf-gcc 13.2.0, Conan 1.56.0, G++ 11.4.0
- **Hardware:** An x86 machine with 20 GB memory
- **Execution:** Shell scripts
- **Metrics:** Simulation speedup, simulation accuracy
- **Output:** Log files (e.g., TOGSim result), TOG files, and intermediate source codes
- **Experiments:** Workloads described in Fig. 5 and Fig. 6 were evaluated
- **How much disk space required (approximately)?:** 50 GB
- **How Much Time Is Needed to Prepare Workflow (Approximately)?:** Downloading the Docker image via Zenodo may take approximately 1–2 hours, depending on your network environment. After downloading, loading the image and creating a container usually takes about 10 minutes.
- **How Much Time Is Needed to Complete Experiments (Approximately)?:** 4 hours
- **Publicly Available?:** Yes
- **Code Licenses (If Publicly Available)?:** MIT license

- **Archived (provide DOI)?:** https://zenodo.org/records/16419768

A.3 Description

A.3.1 How to access. This artifact can be downloaded from https://zenodo.org/records/16419768. The source code and Docker image are also available on GitHub at https://github.com/PSAL-POSTECH/PyTorchSim.

A.3.2 Hardware dependencies. An x86-based machine with at least 20 GB of memory is required. Note that the simulation time may vary depending on the hardware configuration. All experiments in the paper were run on a system with two AMD EPYC 7773X 64-Core Processors (2.2 GHz) and 512 GB of DRAM.

A.3.3 Software dependencies. Docker must be installed in the user's environment, as all experiments are executed inside a Docker container. No additional software dependencies are required due to this Docker container environment.

A.4 Installation

PyTorchSim requires several external repositories, such as LLVM, Spike, and Gem5, which must be built. Setting up the environment and compiling these manually can be complex. To simplify this process, we provide a prebuilt Docker image that includes all required binaries and packages. The Docker image can be downloaded from Zenodo (https://zenodo.org/records/16419768).

Download from Zenodo: Since the PyTorchSim Docker image is large and takes a long time to download, it has been split into 10 smaller files: `torchsim_image.part_00-09`. After downloading all parts, the image can be loaded into Docker using the following command:

```
# Download split image files in parallel (00 ~ 09)
seq -w 0 9 | xargs -I{} -P 10 wget \
  https://zenodo.org/records/16419768/files/\torchsim_image.
       part_0{}

# Merge all parts and load the Docker image
cat torchsim_image.part_* | docker load
```

Note: If you are downloading via the web interface, instead of using Zenodo's `Download all` option, we recommend downloading the split files individually and in parallel (which was significantly faster in our experience).

After loading the image into the Docker environment, a new container can be launched by running the following command:

```
docker run -it --ipc=host --name torchsim \
  -w /workspace/PyTorchSim \
  ghcr.io/psal-postech/torchsim-ci:v1.0.0 bash
```

Then, the installation is complete, and all simulations can be executed inside the created container.

A.5 Experiment workflow

For each experiment (Fig. 5 and Fig. 6), two scripts are provided:

- `run_cycle.sh` — script for accuracy evaluations (Fig. 5).
- `run_speedup.sh` — script for speedup evaluations (Fig. 6).

The scripts must be executed in order: `run_cycle.sh` should be run first, as `run_speedup.sh` uses logs generated during the simulation accuracy experiment. This entire process takes approximately 4 hours to complete, depending on the system specifications. Final outputs will be printed to the terminal and saved in log files (`summary_cycle.log` and `summary_speedup.log`) in the same directory as their corresponding script.

A.6 Evaluation and expected results

To launch the experiments, use the commands below. Each script includes workloads and appropriate configurations.

```
# Run a cycle accuracy script
./experiments/artifact/cycle_validation/run_cycle.sh

# Run a speedup accuracy script
./experiments/artifact/speedup/run_speedup.sh
```

The results from `run_cycle.sh` should match Fig. 5, and the results from `run_speedup.sh` should follow the trends shown in Fig. 6.

Once the scripts have been executed, both the output logs and the generated plots will be saved in the same directory as the corresponding script, as shown below.

```
# 1) Cycle validation results
ls -1 ./experiments/artifact/cycle_validation
run_cycle.sh
cycle_validation.png            # Result Fig. 5
summary_cycle.out               # Result log file
summary_cycle.py

# 2) Speedup results
ls -1 ./experiments/artifact/speedup
results
run_speedup.sh
scripts
speedup.png                     # Result Fig. 6
summary_speedup.log             # Result log file
summary_speedup.py

# 3) Exit the container (when finished)
exit
```

To copy the results from the container to the host machine, you can use the `docker cp` command as shown below.

```
# Copy artifact results to host
docker cp \
  torchsim:/workspace/PyTorchSim/experiments/artifact ./
```