# Hardware-Assisted Virtualization of Neural Processing Units for Cloud Platforms

Yuqi Xue
University of Illinois Urbana-Champaign
yuqixue2@illinois.edu

Yiqi Liu
University of Illinois Urbana-Champaign
yiqiliu2@illinois.edu

Lifeng Nai
Google
lnai@google.com

Jian Huang
University of Illinois Urbana-Champaign
jianh@illinois.edu

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00011

***Abstract*—Cloud platforms today have been deploying hardware accelerators like neural processing units (NPUs) for powering machine learning (ML) inference services. To maximize the resource utilization while ensuring reasonable quality of service, a natural approach is to virtualize NPUs for efficient resource sharing for multi-tenant ML services. However, virtualizing NPUs for modern cloud platforms is not easy. This is not only due to the lack of system abstraction support for NPU hardware, but also due to the lack of architectural and ISA support for enabling fine-grained dynamic operator scheduling for virtualized NPUs.**

**We present Neu10, a holistic NPU virtualization framework. We investigate virtualization techniques for NPUs across the entire software and hardware stack. Neu10 consists of (1) a flexible NPU abstraction called vNPU, which enables fine-grained virtualization of the heterogeneous compute units in a physical NPU (pNPU); (2) a vNPU resource allocator that enables pay-as-you-go computing model and flexible vNPU-to-pNPU mappings for improved resource utilization and cost-effectiveness; (3) an ISA extension of modern NPU architecture for facilitating fine-grained tensor operator scheduling for multiple vNPUs. We implement Neu10 based on a production-level NPU simulator. Our experiments show that Neu10 improves the throughput of ML inference services by up to 1.4× and reduces the tail latency by up to 4.6×, while improving the NPU utilization by 1.2× on average, compared to state-of-the-art NPU sharing approaches.**

***Index Terms*—virtualization, neural processing unit, machine learning accelerator.**

## I. Introduction

Machine learning (ML) is becoming the backbone for many popular ML services, such as online recommendation and natural language processing [4], [7], [44], [47]. To accelerate these ML services, cloud platforms have employed hardware accelerators like neural processing units (NPUs) as the mainstream compute engine [8], [15], [17], [20], [24], [25].

NPUs are highly specialized to accelerate the common operations in deep neural networks (DNNs), such as matrix multiplication and convolution. A typical NPU device is a peripheral board with multiple NPU chips, and each chip has multiple NPU cores. Each NPU core has matrix engines (MEs) that leverage systolic arrays to perform matrix multiplications and vector engines (VEs) for generic vector operations. A well-known example is the Google Cloud TPU [20].

A common approach to using NPUs in cloud platforms is to assign an entire NPU chip to a single ML application instance in a virtual machine (VM) or container via PCIe pass-through [47]. However, this disables resource sharing and causes severe resource underutilization of NPUs. For instance, prior studies [59] disclosed that a majority of the DNN inference workloads cannot fully utilize TPU cores, due to their imbalanced demands on MEs and VEs. Many DNN workloads have diverse demands on the number of MEs and VEs (see §II-B). As a result, the one-size-fits-all approach is much less attractive for cloud platforms.

To address the utilization challenge and ease the resource management for cloud platforms to accommodate diverse workload demands, it is desirable to virtualize hardware devices and enable resource sharing among multiple tenants. Unfortunately, modern cloud platforms have very limited virtualization support for NPUs across the software and hardware stack.

Lack of system abstraction support for NPUs. Unlike the system virtualization of multi-core processors [3], [10], NPUs have unique heterogeneous compute resources (i.e., MEs and VEs). To circumvent this complexity, cloud platforms today expose homogeneous NPU cores to the user VMs. However, the existing abstraction at the NPU core level is too coarse-grained, as user workloads may have diverse resource requirements. We need *a flexible system abstraction that allows users to specify the ME/VE resources* following the pay-as-you-go model [48]. Such an abstraction will simplify the NPU management for cloud platforms, including NPU resource (de)allocation, resource mapping, and scheduling. Prior studies investigated the system virtualization for FPGAs [6], [33], [34], [63], [64] and GPUs [26], [55]. However, they cannot be directly applied to NPUs, as they target different architectures.

Lack of architectural support for NPU virtualization. Prior studies enabled the time-sharing of an NPU device at the task level, and support the preemption for prioritized tasks [12], [13]. However, the coarse-grained time-sharing on the shared NPU board still suffers from severe resource underutilization, due to the lack of support of concurrent execution of multi-tenant workloads. Existing NPU sharing

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00011



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

approaches either sacrifice isolation or suffer from high preemption overhead [16]. V10 [59] enabled NPU sharing between multiple DNN workloads. However, it is still based on the time-sharing mechanism and suffers from operator interference between multi-tenant ML instances, resulting in poor performance isolation. As we move towards fine-grained NPU virtualization, we need *architectural support to achieve both improved performance isolation and NPU utilization*.

Lack of ISA support for virtualized NPUs. To simplify the hardware design, NPUs commonly employ VLIW-style ISAs, and the ML compiler explicitly exploits the parallelism of the compute units [5], [28], [32]. However, this requires the number of compute units to be explicitly specified at the compilation stage, and the number cannot be changed at runtime. In this case, the VLIW ISAs unnecessarily couple control flows of the compute units (i.e., MEs). Even though some compute units of a shared NPU become available, they cannot be utilized by the active workload (except recompiling the DNN program). This is caused by the fundamental tussle between dynamic scheduling and VLIW ISAs. As the collocated ML instances have various demands on compute units at runtime, this limitation inevitably causes either NPU underutilization or performance interference. We need to *rethink the NPU ISA design to facilitate dynamic resource scheduling for virtualized NPUs*.

Ideally, we wish to virtualize NPUs to enable flexible and fine-grained resource sharing and scheduling for improved NPU utilization and performance isolation. We present Neu10, a hardware-assisted system virtualization framework for NPUs.

Our contributions. We first develop a simple yet flexible *vNPU* abstraction. We use vNPU to create a virtualized NPU device for each ML instance. For each vNPU, the user can specify the number of different types of compute units (MEs/VEs) on-demand or follow the pay-as-you-go model in cloud computing. We propose a new resource allocation mechanism that can decide the optimized vNPU configuration for different ML workloads, based on the analysis using ML compilers. As different ML services have various ME/VE demands (see §II), such an abstraction enables fine-grained resource allocation, which benefits both end users and cloud platform operators[1].

Neu10 can map vNPUs to physical compute units of NPU cores in different manners, based on the service level objectives (SLOs) of ML services. To maximize the NPU utilization while ensuring performance isolation, Neu10 enables fine-grained spatial sharing with resource harvesting. It also enables the oversubscription of NPU cores by temporally sharing MEs/VEs among multiple vNPUs. Therefore, the idle compute units can be opportunistically utilized by collocated workloads.

To facilitate the dynamic scheduling for collocated vNPUs, Neu10 extends the VLIW-style ISA by reorganizing VLIW instructions into independent micro-Tensor operators ($\mu$TOps in §III). Neu10 introduces necessary architectural logic for fine-grained dynamic scheduling of $\mu$TOps on the shared physical NPU cores. It allows one vNPU to harvest available compute cycles of MEs/VEs from collocated vNPUs, without causing much interference. This is impossible with conventional VLIW-style ISAs, as they strictly couple the control flows of the (statically) allocated compute units. Our new architectural support enables Neu10 to offer the flexibility of NPU resource allocation and scheduling across the software (i.e., vNPU abstraction) and hardware (i.e., fine-grained $\mu$TOp scheduling) stack. Neu10 requires minimum modifications to NPU chips (0.04% die area cost) as well as ML compilers.

[1]The fine-grained resource allocation allows end users to allocate the NPU resources on demand, and enables cloud platforms to implement the pay-as-you-go model at a fine granularity as they have done for multi-core processors.

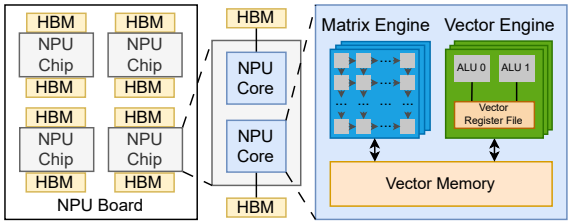



Fig. 1: System architecture of a typical NPU board.

We implement Neu10 with a production-level NPU simulator following the typical TPU architecture. We collect the traces of ML services as we run the MLPerf benchmarks [46] and the TPU reference models [22] on the real Google TPUs. Our experiments with multi-tenant ML instances show that Neu10 can improve the throughput of ML inference services by up to 1.4× and reduce the tail latency by up to 4.6×, while improving the NPU utilization by 1.2× on average, in comparison with state-of-the-art NPU sharing approaches. We summarize the contributions of Neu10 as follows:

- We conduct a thorough study of DNN inference workloads on real NPU hardware, and investigate the NPU virtualization challenges within both system and hardware stack (§II).
- We propose a new system abstraction named vNPU for enabling fine-grained virtualization of the heterogeneous compute units in NPU cores (§III-A).
- We present a new NPU resource allocation scheme and enable flexible vNPU-to-pNPU mappings (§III-B and §III-C).
- We extend the VLIW-style ISAs and NPU architecture for enabling fine-grained dynamic scheduling of virtualized NPUs for multi-tenant ML services (§III-D and §III-E).
- We evaluate the efficiency and flexibility of our NPU virtualization framework with real-world DNN traces (§V).

## II. Background and Motivation

### A. NPU System Architecture

As shown in Figure 1, an NPU board has multiple NPU chips, each chip has multiple NPU cores, each core is connected to an off-chip HBM. An NPU core has two types of compute units: matrix engines (MEs) that perform matrix multiplications with systolic arrays; and vector engines (VEs) that perform generic vector operations. Each NPU core employs an on-chip SRAM to hide HBM access latency. A typical example of NPU architecture in production is Google TPU [31].



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

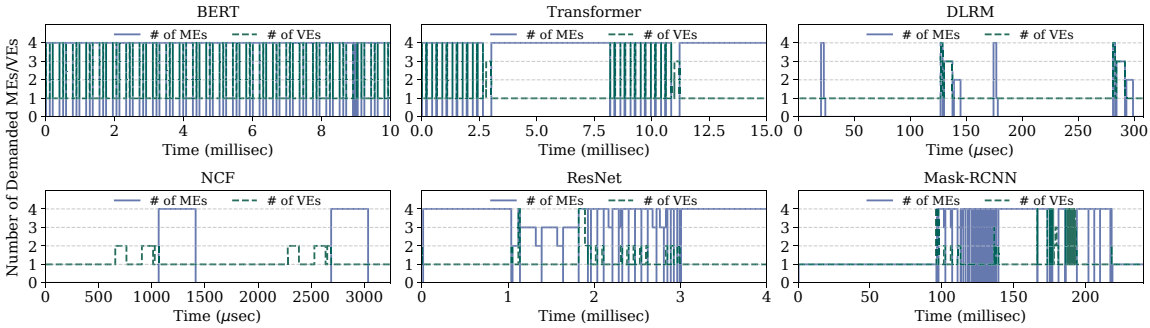



Fig. 2: The number of MEs and VEs demanded by DNN inference workloads over time (batch size = 8).

TABLE I: DNN models used as ML services in this paper.

| Category | Model Name | Abbrev. | HBM Footprint (batch size = 8) |
|---|---|---|---|
| Natural Language Processing | BERT | BERT | 1.27GB |
| | Transformer | TFMR | 1.54GB |
| Recommendation | DLRM | DLRM | 22.38GB |
| | NCF | NCF | 11.10GB |
| Object Detection & Segmentation | Mask-RCNN | MRCNN | 3.21GB |
| | RetinaNet | RtNt | 860.51MB |
| | ShapeMask | SMask | 6.04GB |
| Image Classification | MNIST | MNIST | 10.59MB |
| | ResNet | RsNt | 216.02MB |
| | ResNet-RS | RNRS | 458.17MB |
| | EfficientNet | ENet | 99.06MB |

To run a DNN program on NPUs, ML compilers [14], [21], [45] generate a sequence of tensor operators, which are then translated into device-specific machine instructions. An NPU core usually uses a VLIW-style ISA for simplifying the hardware. Each instruction contains multiple ME slots, VE slots, load/store slots for accessing the SRAM, and other slots (e.g., for DMA operations with HBM). The ML compilers can exploit the instruction-level parallelism with the knowledge of underlying compute resource, such as the numbers of MEs/VEs.

### *B. Characterization of ML Inference Services*

To motivate NPU virtualization, we conduct a study of resource demands of ML inference workloads and their impact on NPU utilization. We run various ML inference workloads from MLPerf benchmarks [46] and official TPU reference models [22] (see Table I), on a real Google TPUv4 board with 8 cores. Each core has four MEs and two VEs. We profile the number of MEs/VEs demanded by each workload with ML compiler techniques, and the resource utilization with performance counters on the TPU core. We vary the batch size (8 by default). The HBM footprint of benchmarks ranges from 10.59MB to 22.38GB, which does not fully occupy the HBM on modern NPU chips (e.g., 32GB/96GB on TPUv4/TPUv5p [52]). We report the resource utilization on one TPU core, as all cores perform identical computations with data parallelism.

**Diverse demands on MEs/VEs.** An ML inference workload can have diverse resource demands over time, as different operators in a DNN model have vastly different demands

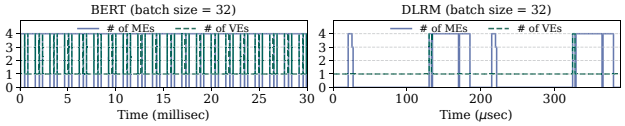



Fig. 3: The number of MEs and VEs demanded by DNN inference workloads with a larger batch size.

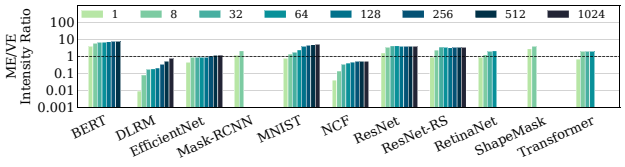



Fig. 4: Intensity ratio of ME vs. VE for different inference workloads (quantified by the execution time of ME/VE).

on MEs and VEs. For each workload, we analyze the DNN execution graph generated by the ML compiler. By default, the ML compiler picks the number of compute units for each operator to maximize the overall efficiency of the compute units based on the tensor shapes. We use this to quantify the ME/VE demands. Figure 2 shows that DNN inference workloads have various ME/VE demands over time. As we increase the batch size, we observe similar patterns (Figure 3). Due to space limitations, we only show the results of BERT and DLRM. The imbalanced demands are determined by the ML model architecture. For example, in Figure 4, ResNet is dominated by convolutions (ME-intensive operators), while DLRM contains many vector operators, which do not utilize the ME at all. For workloads that cannot run with large batch sizes due to insufficient memory, we do not show them in Figure 4.

**Low NPU resource utilization.** The diverse demands on MEs/VEs inevitably cause NPU underutilization. We quantify the percentage of idleness of the MEs/VEs in Figure 5. Although workloads like DLRM and NCF may appear to be VE-intensive, at least 20% of their execution time still involves heavy ME computation. For ME-intensive models such as ResNet, many operators are also VE-intensive. To balance the demands on ME and VE, the ML compiler can perform



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

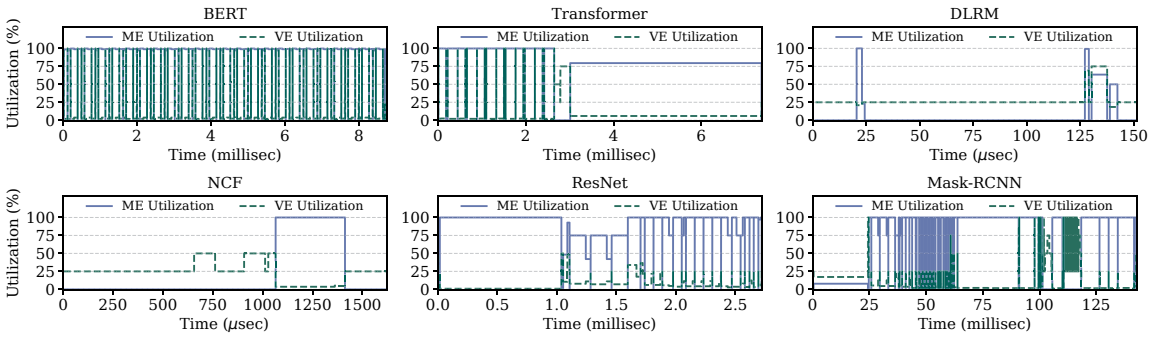



Fig. 5: The utilization of ME and VE of an inference request for representative DNN models.

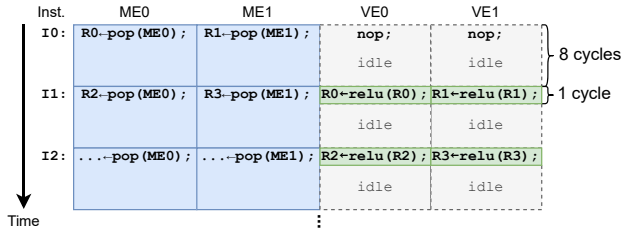



Fig. 6: Example of VE underutilization in an ME-intensive operator (fused matrix multiplication and ReLU activation).

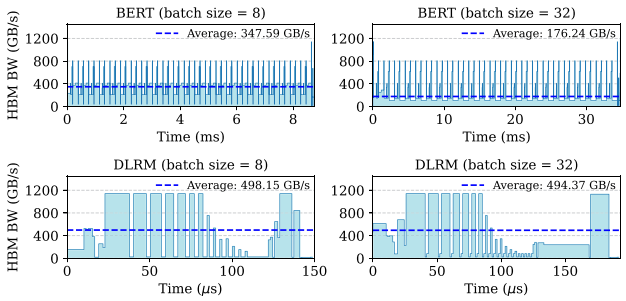



Fig. 7: The HBM bandwidth utilization for representative DNN models with different batch sizes.

operator fusion to pipeline the execution of ME and VE [14], [23], [59]. However, as such fusion opportunities are limited, most operators still have imbalanced ME/VE demands after fusion. Figure 6 shows an example of VE underutilization in an ME-intensive operator. Each `pop` operation takes 8 cycles to generate an $8 \times 128$ output vector from the ME, while each VE operation takes 1 cycle to post-process the output vector. As a result, the VE is idle for most of the time.

We also profile the HBM bandwidth utilization in Figure 7. While the peak bandwidth almost reaches the hardware limit (1.2TB/s on a TPUv4 chip), the average bandwidth is as low as 176–498GB/s. This is because different operators in a DNN model have varying bandwidth demands. For example, in DLRM, the embedding lookup consumes high bandwidth, while the multi-layer perceptron (MLP) has low bandwidth requirements. As we increase the batch sizes, the bandwidth consumption decreases for some workloads. For example, BERT is dominated by ME operators, which become more compute-intensive with larger batch sizes; DLRM is VE-intensive, and VE operators have low compute intensity regardless of batch sizes. As some DNN operators underutilize the HBM bandwidth while other operators underutilize the compute resources, collocating DNN workloads on the same NPU core helps cloud platforms utilize both resources.

### C. NPU Virtualization: Challenges and Opportunities

System virtualization offers the opportunity for supporting multi-tenancy and improving resource utilization. However, virtualizing NPUs suffers from unique challenges.

**New abstraction required for fine-grained virtualization.** As none of prior studies investigated NPU virtualization, it is unclear how the virtualized NPUs should be exposed to application instances. By virtualizing NPUs, we need to provide a simple yet effective abstraction, which can provide sufficient flexibility for users to specify the numbers of MEs and VEs based on the workload demand and target SLOs (see §III-B). For instance, we should allocate more MEs than VEs to an ME-intensive workload, and vice versa.

However, even if we can allocate the most appropriate numbers of MEs and VEs, the allocated resources still cannot be fully utilized, due to the diverse resource demands of different operators over time. A static allocation of MEs and VEs is insufficient. Instead, we need to enable dynamic resource scheduling. We should allow one workload to "*harvest*" the underutilized compute units allocated to other workloads for improving the overall utilization of the NPU core and the Quality-of-Service (QoS) of collocated ML inference services. Unfortunately, current NPU architectures do not support such fine-grained resource scheduling and harvesting.

**ISA limitations for enabling virtualized NPU scheduling.** The fundamental limitations of modern NPU architectures prevent dynamic resource scheduling. To simplify the hardware design of NPUs, developers usually employ VLIW-style ISAs, and utilize ML compilers to exploit the instruction-level parallelism. However, the statically scheduled ISAs cannot fully



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

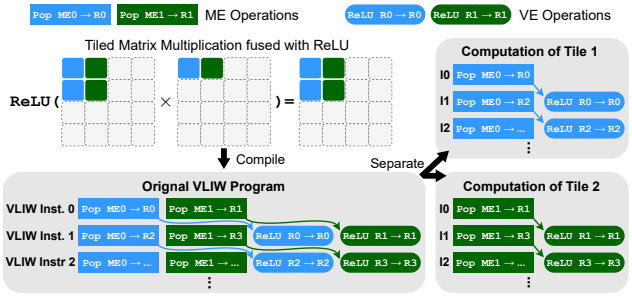



Fig. 8: Execution of MEs and VEs are separable. The arrows between instructions denote data dependencies.

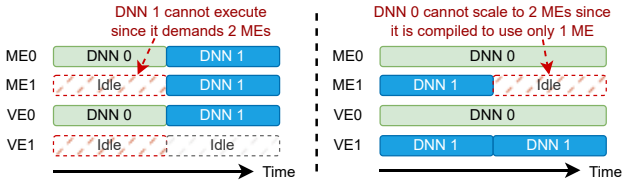



Fig. 9: The current VLIW-style ISA causes NPU underutilization, as it cannot exploit available MEs at runtime.

exploit the hardware resources at runtime. They unnecessarily couple the control flows of all MEs in a tensor operator, even though different MEs can execute independently. As shown in Figure 8, the original VLIW program must execute each VLIW instruction sequentially, creating false dependencies between operations on different MEs even though they do not have any true data dependencies. As the compiler explicitly specifies how many MEs are being used, the allocated MEs cannot be changed at runtime unless the DNN program is recompiled. For example, if the compiler generates `push`/`pop` operations for two MEs, these operations cannot be time-multiplexed on a single ME, since this will corrupt the intermediate states in the ME. Hence, if only one ME is available, this DNN program cannot run until at least two MEs are available (Figure 9 left). It also cannot utilize more than two MEs, even if more than two are available (Figure 9 right), because the `push`/`pop` operations for one ME share the intermediate data in this ME.

To address this problem and enable dynamic ME scheduling, one may consider switching from VLIW to another ISA (e.g., RISC) or employing superscalar out-of-order (OoO) execution (similar to a CPU core). However, they still lack the support for dynamic ME scheduling since the compiler still needs to specify which ME is the target of a `push`/`pop` instruction statically. To remove such a constraint, we need to offer the flexibility for the NPU program to determine the target ME at runtime. Therefore, we need to rethink the contract between the compiler and the NPU hardware by extending the ISA.

**Architectural support for parallelizing ME/VE operations.** Our key observation is that the execution of different MEs and VEs in a tensor operator is usually *separable*. Specifically, most DNN operators, such as matrix multiplication (MatMul) and

```
struct vNPU_Config {
   size_t num_chips;            size_t num_cores_per_chip;
   size_t num_MEs_per_core;     size_t num_VEs_per_core;
   size_t sram_size_per_core;   size_t mem_size_per_core;
}
```

Fig. 10: vNPU configuration.

convolution, are partitioned by DNN compilers [14], [66] into multiple tiles that can be computed independently. As shown in Figure 8, the original program computes a MatMul tile and directly applies a ReLU function to the results using 2 MEs and 2 VEs. However, the instructions executed on the first ME/VE (colored blue) have no dependencies with the instructions on the second ME/VE (colored green). The two instruction groups can be separated and independently executed.

## III. Design and Implementation

We design Neu10 to achieve the following objectives:

- **Allocation flexibility:** As DNN workloads have different resource and SLO requirements, we need to provide the flexibility for users to customize their NPU hardware.
- **NPU utilization:** Since an individual ML inference workload underutilizes NPU cores (§II-B), we need to enable fine-grained NPU virtualization for improved NPU utilization.
- **Performance isolation:** As we collocate DNN workloads on the same NPU core, we must provide performance isolation.

We first present a new vNPU abstraction for NPU virtualization (§III-A). Based on this, we enable flexible vNPU resource allocation (§III-B) and vNPU-to-pNPU mappings (§III-C). We extend VLIW-style ISA (§III-D) and NPU architecture (§III-E) for enabling fine-grained resource scheduling for vNPUs.

### A. vNPU: The New Abstraction for NPU Virtualization

We design the vNPU abstraction with the goals of (1) allocating NPU hardware resource to a vNPU instance on demand; (2) hiding the complexity from the ML programs with minimal changes to the guest software stack for compatibility.

**vNPU abstraction.** A vNPU instance reflects the hierarchy of a physical NPU board. Figure 10 shows the configurable parameters of a vNPU. Each vNPU is exposed to the VM as a PCIe device. The guest NPU driver can query the hierarchy of the vNPU, such as the number of chips, cores per chip, HBM size, and others. The maximum vNPU size is capped by the physical NPU size. If a guest VM requires more resources than is available on a physical NPU board, Neu10 can allocate multiple vNPU instances to it. The guest ML framework can handle the data distribution across multiple vNPU cores in the same way as that on physical NPUs. Take Google TPU for example, TensorFlow already handles data parallelism across physical NPUs. It can work in the same way with vNPUs.

**vNPU lifecycle.** To create a vNPU instance, a user can specify the vNPU configuration following the pay-as-you-go model [48]. Cloud providers can define various default configurations (e.g., small/medium/large vNPU cores as having



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

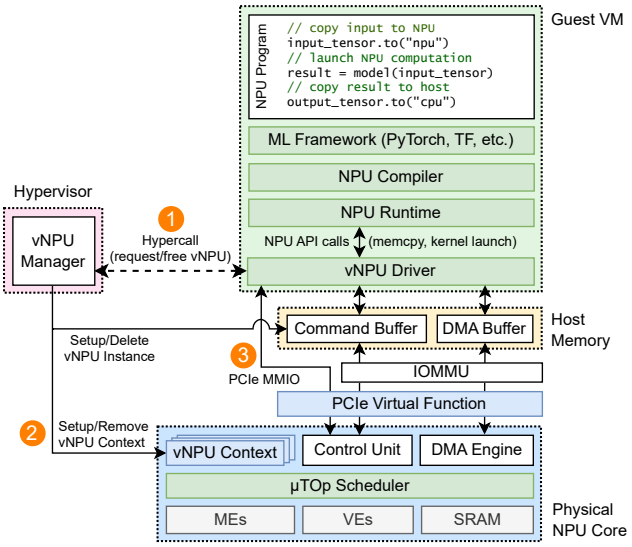



Fig. 11: System architecture of Neu10.

1/4/8 MEs/VEs). Neu10 can also learn an optimized vNPU configuration for a DNN workload with ML compilers (§III-B). As shown in Figure 11, upon vNPU initialization, the guest driver sends a request to the hypervisor through a para-virtualized interface (§III-F) (1). The vNPU manager maps the vNPU instance to NPU hardware resources (§III-C). Then, it initializes the vNPU context in the physical NPU device and creates the MMIO mappings for the guest VM to access the vNPU (2). During execution, the application issues commands such as memcpy and compute offloading through the command buffer. The NPU hardware directly fetches the commands from the host memory without the hypervisor intervention. It also has DMA access to the DMA buffer in the guest memory space via the IOMMU. The DNN program on the NPU executes asynchronously from the CPU program, and the NPU hardware schedules vNPUs (§III-E) independently of existing OS/hypervisor schedulers. The guest VM waits for the completion interrupt or actively polls the memory-mapped control registers for the current status of the vNPU (3). After execution, the user can free the vNPU.

### B. vNPU Allocation and Deallocation

Following the popular pay-as-you-go model [48], cloud platforms allow users to specify the vNPU configuration on demand. However, as ML inference workloads have diverse ME/VE demands (see §II-B), specifying the number of MEs/VEs can be challenging for users who are not NPU experts. Thus, we allow them to specify the total number of execution units (EUs), which is directly related to the cost of running the vNPU instance. Neu10 provides the vNPU allocator, a compile-time tool to improve the performance per cost of vNPUs by identifying an optimized ME/VE ratio for the user workload.

**ME/VE allocation.** The ME/VE demands of a ML workload can be reflected by how it runs on one ME and one VE. We denote the $\frac{\text{ME active runtime}}{\text{NPU total runtime}}$ as $\boldsymbol{m}$, and that of VE as $\boldsymbol{v}$. These numbers can be obtained via profiling at the compilation stage. Based on our study in §II-B, for most DNN models, at least one of ME/VE is active during the execution of an NPU core. Thus, the time portion where only ME is active is $1-v$, that of only VE is $1-m$, and that of concurrent ME/VE execution is $m+v-1$. With Amdahl's Law, the normalized execution time on $\boldsymbol{n_m}$ MEs and $\boldsymbol{n_v}$ VEs is

$$\boldsymbol{T} = \frac{1-v}{n_m} + \frac{1-m}{n_v} + \frac{m+v-1}{\min(n_m, n_v)}. \quad (1)$$

where the concurrent part is bottlenecked by the minority type of EU. Let $n_m + n_v$ be the hypothetical speedup regardless of EU types, which means an EU can execute both ME and VE operators. Compared to real cases where each EU must respect data dependencies and operator types, the hypothetical speedup assumes all $n_m + n_v$ EUs are 100% utilized. Thus, the hypothetical execution time on $\boldsymbol{n_m}$ MEs and $\boldsymbol{n_v}$ VEs is $\boldsymbol{T_h} = \frac{m+v}{n_m+n_v}$, and the total EU utilization can be quantified as the ratio between hypothetical and estimated execution times:

$$\boldsymbol{U} = \frac{\boldsymbol{T_h}}{\boldsymbol{T}} = \frac{m+v}{(n_m+n_v)(\frac{1-v}{n_m} + \frac{1-m}{n_v} + \frac{m+v-1}{\min(n_m,n_v)})}. \quad (2)$$

To isolate the impact of total ME and VE quantity, we simplify the function by letting $\boldsymbol{k} = n_m/n_v$ be the ratio between the numbers of MEs and VEs. Without loss of generality, we assume $n_v \geq n_m$, which means $k \leq 1$. Then, we can simplify Equation (2) with mathematical tools [56]:

$$\boldsymbol{U} = \frac{(m+v)k}{(1-m)k^2 + k + m} \quad (k \leq 1). \quad (3)$$

To find the value of $k$ that maximizes $U$, we compute the value of $k$ where $\frac{\mathrm{d}U}{\mathrm{d}k} = 0$. This gives $k = \sqrt{m/(1-m)}$ for $m < 0.5$. If $m \geq 0.5$, $U$ will be monotonic, so $k = 1$ maximizes $U$. Similarly, for the case when $n_m \geq n_v$, we derive $k = \sqrt{(1-v)/v}$ for $v < 0.5$ and $k = 1$ for $v \geq 0.5$. Consequently, we have

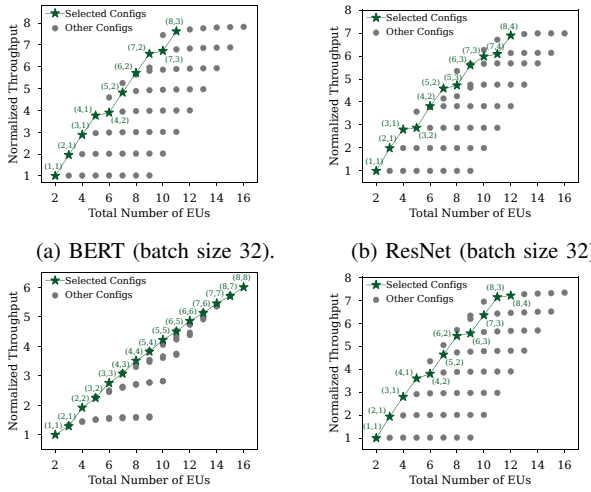
$$k = \frac{n_m}{n_v} = \begin{cases} \sqrt{m/(1-m)}, & m < 0.5, \\ \sqrt{(1-v)/v}, & v < 0.5, \\ 1, & m \geq 0.5 \text{ and } v \geq 0.5. \end{cases} \quad (4)$$

The case when both $m < 0.5$ and $n < 0.5$ does not exist since at least one of ME/VE will be active ($m+n \geq 1$). When $m < 0.5$, for workloads with ME active time ratio $m$, we allocate $\sqrt{m/(1-m)}$ times more MEs than VEs. When $v < 0.5$, for workloads with VE active time ratio $v$, we approximate the allocated ME/VE quantity ratio to $\sqrt{(1-v)/v}$. If $m > 0.5$ and $v > 0.5$, we allocate the same number of MEs and VEs. Note that each vNPU will have at least one ME and one VE.

**Memory allocation.** Users can use the compiler to estimate the total HBM capacity needed by a DNN workload. By default, the SRAM capacity is allocated proportionally to the number of allocated MEs, as more MEs usually indicate larger tile sizes. Based on our study in §II-B, for many common ML inference services, the HBM bandwidth is less of a concern. Thus, Neu10 allows fair sharing of HBM bandwidth by default. For



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.




(a) BERT (batch size 32). (b) ResNet (batch size 32).

(c) EfficientNet (batch size 32). (d) ShapeMask (batch size 8).

Fig. 12: vNPU allocation results for representative DNN models as we scale up the available EUs on an NPU core from 1 ME and 1 VE to 8 MEs and 8 VEs. Each data point is a vNPU configuration. The label ($m$, $v$) means $m$ MEs and $v$ VEs.

large models that demand large HBM capacity and bandwidth, the vNPU abstraction offers the flexibility for end users to allocate the demanded resources. The user may also leverage existing tensor swapping techniques to support large DNN workloads with limited memory capacity [27], [65]. After vNPU allocation, ML compilers will compile the DNN program with the allocated resources. The compiler ensures the DNN program does not exceed the allocated SRAM and HBM. We will discuss how Neu10 handles compilation for different numbers of MEs/VEs in §III-D.

**Cost-effectiveness analysis.** We evaluate our allocation algorithm in Figure 12. For vNPUs with no more than 4 MEs and 2 VEs, we use a real TPUv4 to test the throughput. For others, we use a production-level NPU simulator (see §III-G). In most cases, our algorithm selects a configuration with better performance than others for the same number of EUs. Though a sub-optimal configuration may be selected, it still achieves similar performance as the optimal one. The ME/VE harvesting (§III-E) also tolerates some allocation inaccuracies by opportunistically utilizing more EUs.

**vNPU deallocation.** Upon vNPU deallocation, the vNPU manager will send a command associated with the vNPU ID to the corresponding NPU board to clean up the vNPU context, as well as remove the DMA setup for this vNPU.

### C. vNPU Mapping

The vNPU manager attempts to balance the number of allocated EUs and the size of allocated memory. This minimizes the chance that all EUs on one core are allocated but a large portion of its memory is not allocated, or vice versa. Thus, vNPUs with many EUs and small memory will be collocated

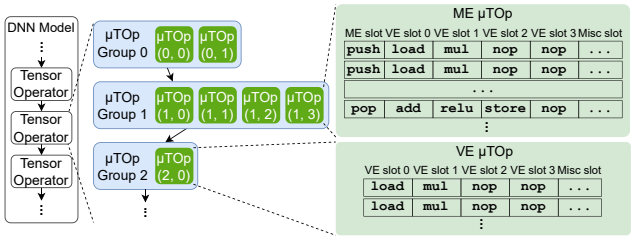



Fig. 13: NeuISA programming model.

with vNPUs with few EUs and large memory. Neu10 uses a greedy algorithm for this by default.

**vNPU mapping schemes.** Neu10 provides the flexibility for cloud platforms to enable both *hardware-isolated* (spatial-isolated) mapping and *software-isolated* (temporal-sharing) mapping. With hardware-isolated mapping, a vNPU is mapped to dedicated EUs and SRAM, and the allocated hardware is not shared with other vNPUs. With software-isolated mapping, multiple vNPUs can temporally share the same EUs. Neu10 uses priority-based scheduling for fair sharing and performs context switches between vNPUs (see §III-E).

**vNPU mapping policies.** Neu10 decides which vNPUs can be mapped onto the same physical NPU (pNPU) as follows. With hardware-isolated mapping, Neu10 collocates a set of vNPUs as long as the total resource requirement (e.g., number of MEs/VEs, HBM capacity) does not exceed the pNPU. With software-isolated mapping, Neu10 aims to load-balance the pNPUs while allowing oversubscription. Neu10 tracks the total resource requirement of assigned vNPUs on each pNPU, and assigns a new vNPU to the pNPU that suffers the least resource requirement. Neu10 can support other collocation policies [12], [39], [59] as well. At scale, Neu10 can be integrated with a cluster-wise VM/container orchestration framework such as KubeVirt/Kubernetes [50] to decide which VM should be placed on what machine. Developing advanced vNPU/VM collocation policies is orthogonal to our work.

**vNPU security isolation.** Neu10 enforces memory address space isolation among collocated vNPUs with the conventional memory segmentation scheme [2], [59] for both HBM and SRAM. Neu10 divides the SRAM and HBM into fixed-sized segments and maps each segment to the virtual address space of a vNPU. For the NPU core in Table II, an SRAM/HBM segment is 2MB/1GB. There is no external fragmentation since the segment size is fixed. The address translation is performed by adding the segment offset to the starting address of the physical segment, which incurs negligible overhead. A page fault will be triggered when an invalid access happens. This is sufficient since ML frameworks like TensorFlow typically request a contiguous chunk of memory for the entire lifetime of an ML inference service and have their own memory management mechanism. To isolate the vNPU instances as they communicate with the host, Neu10 uses IOMMU to enforce DMA remapping (§III-F). We leave side-channel mitigation to future work.



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

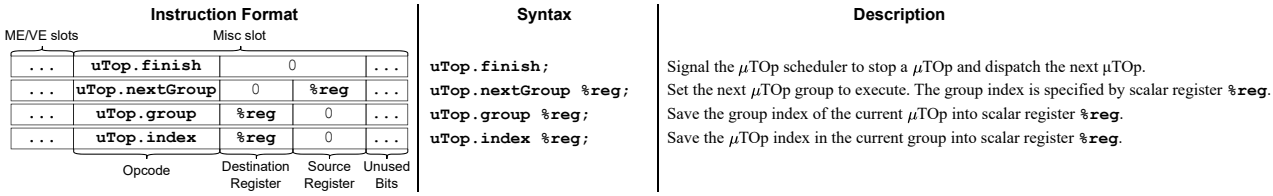



Fig. 14: Definitions of $\mu$TOp control instructions. Scalar register zero (`%r0`) is read-only and always has a value of 0.

### D. ISA Extension for NPU Virtualization

To support dynamic ME/VE scheduling (§II-C), we develop NeuISA, in which when the ML compiler maps a tensor operator onto MEs, it generates "sub-tasks" for each ME, so the hardware can decide which "sub-task" can be executed at runtime based on the availability of MEs. NeuISA is still expressive for compilers to exploit the instruction-level parallelism between MEs and VEs, and preserve the flexibility of supporting fused operators and complex control-flow structures like branches and nested loops in VLIW-style ISAs.

**Separating ME control flow with $\mu$TOps.** NeuISA decouples the execution of independent MEs in a tensor operator by separating the control flow of each ME and VE into independent instruction sequences (see Figure 8), called *micro-Tensor Operators ($\mu$TOps)*. To minimize changes to the existing VLIW compiler and hardware, the instruction format inside a $\mu$TOp resembles the original VLIW ISA: an instruction contains multiple slots, and each slot encodes an operation (such as a `push`/`pop` operation in an ME slot and an ALU operation in a VE slot). However, the number of ME slots in a NeuISA instruction differs from that of a traditional NPU ISA.

**$\mu$TOp types.** As shown in Figure 13, for a physical NPU core with $n_x$ MEs and $n_y$ VEs, NeuISA defines two types of $\mu$TOps: (1) An *ME $\mu$TOp* contains instructions with one ME slot and $n_y$ VE slots. An ME $\mu$TOp will only use one ME during execution, which enforces that each ME $\mu$TOp only contains the control flow of one ME. To execute an operator on multiple MEs, the compiler generates multiple ME $\mu$TOps. At runtime, the hardware dynamically adjusts the number of MEs assigned to this operator by deciding how many ME $\mu$TOps are being executed. The VE slots in an ME $\mu$TOp enable instruction-level parallelism between MEs and VEs. VE slots are necessary because the VE needs to aggregate the outputs of the systolic array. They also enable operator fusions such as MatMul+ReLU (see Figure 8). (2) A *VE $\mu$TOp* contains instructions with no ME slot and $n_y$ VE slots, which performs vector operations that do not involve ME computation. The $n_y$ VE slots allow a VE $\mu$TOp to utilize all the VEs. Having multiple VE slots in an instruction does not increase the hardware complexity since the original VLIW NPU architecture already supports this.

**Supporting fused operators with $\mu$TOp groups.** The $\mu$TOps can efficiently support basic tensor operators, such as tiled matrix multiplication with each $\mu$TOp computing a different tile. However, ML compilers may generate fused operators that cannot be handled by $\mu$TOps alone, e.g., a matrix multiplication

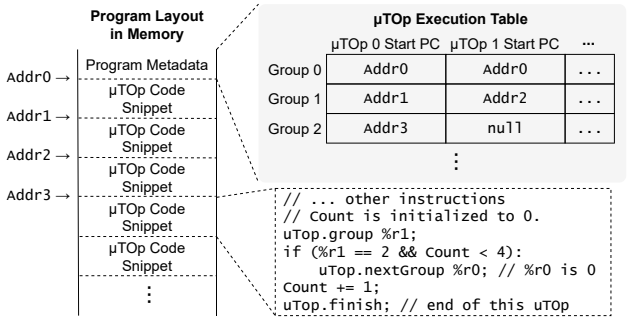



Fig. 15: NeuISA program structure.

may be executed with $n_x$ ME $\mu$TOps, while the succeeding fused normalization operator only needs a VE $\mu$TOp.

To support a fused operator, NeuISA organizes the $\mu$TOps into a sequence of *$\mu$TOp groups* to express the dependencies between $\mu$TOps, as shown in Figure 13. Each group contains up to $n_x$ ME $\mu$TOps, allowing the operator to utilize all the allocated MEs, and up to one VE $\mu$TOp, as one VE $\mu$TOp already contains $n_y$ VE slots to utilize all the VEs. All $\mu$TOps in one $\mu$TOp group may execute concurrently, but each group must execute sequentially to preserve data dependency. As an example, a fused operator may contain one $\mu$TOp group doing a MatMul+ReLU with multiple ME $\mu$TOps, followed by a $\mu$TOp group doing normalization with a single VE $\mu$TOp.

**NeuISA control flow.** As NeuISA inherits the VLIW semantic inside each $\mu$TOp, it intrinsically supports conditional branches and loops inside a $\mu$TOp. It is also desirable to have branches across $\mu$TOp groups. For example, an operator contains a nested loop in which the inner-most loop is a matrix multiplication that can be mapped to a $\mu$TOp group. In this case, we need to support loops across multiple $\mu$TOp groups.

NeuISA defines special control instructions that can be invoked in each $\mu$TOp (see Figure 14). The `uTop.nextGroup` instruction can be used to specify the target $\mu$TOp group that should be executed next. It may be executed by more than one $\mu$TOps in the same group as long as they specify the same target group index. Otherwise, an exception will be raised. Figure 15 shows a loop structure example. The loop counter `Count` is stored in the on-chip SRAM. The loop body contains $\mu$TOp group 0–2. In group 2, `Count` is incremented and examined at the end of a $\mu$TOp. If this is not the last loop iteration, `uTop.nextGroup` is executed to loop back to group 0.

**NeuISA program structure.** A NeuISA binary contains *$\mu$TOp*



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

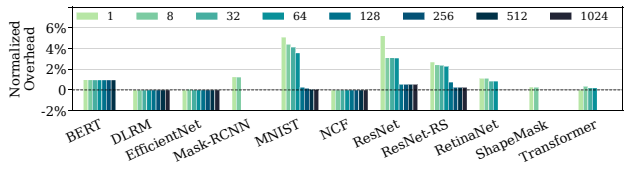



Fig. 16: Performance overhead of NeuISA over the traditional VLIW-style ISA for various DNN workloads.

*code snippets*, as shown in Figure 15, which are the assemblies for $\mu$TOps. The $\mu$TOp groups are encoded by a *$\mu$TOp execution table*. Each row defines a $\mu$TOp group. Each cell is the start address of a $\mu$TOp code snippet. NeuISA provides control instructions to retrieve the group index and $\mu$TOp index of the current $\mu$TOp (see Figure 14). The size of each row in the $\mu$TOp execution table depends on the number of MEs/VEs on the physical NPU core. For a physical core with $n_x$ MEs, each row has $n_x$ ME $\mu$TOp entries and one VE $\mu$TOp entry. An entry will be `null` if the $\mu$TOp does not exist in the group.

A DNN program is executed by the NPU core following the $\mu$TOp execution table. By default, $\mu$TOp group $i+1$ will be executed after group $i$ (starting from group 0), unless `uTop.nextGroup` specifies another group index. The $\mu$TOps in the same group can execute in any order. Each $\mu$TOp executes a snippet of VLIW instructions.

**Compiler support for NeuISA.** NeuISA allows a DNN program to utilize different numbers of MEs/VEs at runtime without recompilation, regardless of the allocated vNPU size at compilation time. This is supported with minimal compiler changes. For a physical NPU core with $n_x$ MEs and $n_y$ VEs, we first employ existing compiler techniques [66] to partition each operator into up to $n_x$ $\mu$TOps, which allows the DNN program to utilize all MEs on the NPU core. Next, we employ the existing compiler backend such as XLA [23] to compile each $\mu$TOp independently assuming a fictional NPU with one ME and $n_y$ VEs. Finally, we extract the dependencies between $\mu$TOps from the DNN execution graph, and append NeuISA control flow instructions at the end of $\mu$TOps when necessary.

**NeuISA Overhead.** NeuISA incurs negligible performance overhead (less than 1% on average) for most DNN workloads (see Figure 16). The major overhead occurs when a matrix multiplication is partitioned on the reduction dimension to utilize all MEs. In this case, NeuISA prevents instruction-level pipelining between ME computation and summing the ME outputs on the VEs, as the summation must be done in a separate VE $\mu$TOp after the ME $\mu$TOps. The overhead is smaller for larger batch sizes, as the compiler will partition other dimensions (e.g., the batch dimension) if they are large enough. While NeuISA may inflate the code size by having multiple multiple VE slots in a $\mu$TOp, this is less of a concern in practice since NeuISA minimizes code inflation by sharing the same code snippet among $\mu$TOps. The on-chip instruction memory is large enough to avoid stalling the pipeline.

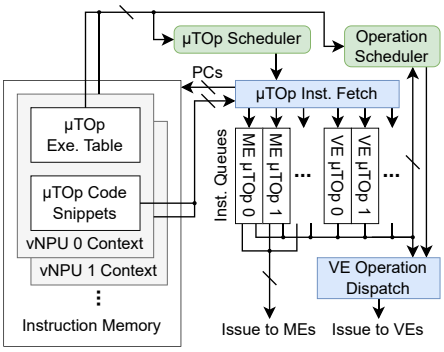



Fig. 17: NPU core pipeline frontend for NeuISA.

### *E. Architectural Support for NeuISA*

The $\mu$TOp design enables dynamic operator scheduling. It allows a vNPU to harvest unused ME/VEs from other collocated vNPUs in the same physical NPU core at runtime.

**Hardware scheduler for NeuISA.** Figure 17 shows the pipeline design for fetching and scheduling $\mu$TOps. The NPU core maintains the contexts of multiple vNPUs, including the PC pointers to the program and the vNPU configurations. Each time a new $\mu$TOp is ready or an existing $\mu$TOp finishes, the *$\mu$TOp scheduler* selects the $\mu$TOps to be executed next. For each vNPU, the $\mu$TOp scheduler retrieves the number of allocated MEs and the number of ready ME $\mu$TOps from the vNPU context. It selects a set of ready $\mu$TOps, and fetch their instructions to the instruction queues.

Next, the *operation scheduler* selects which operations from the instruction queues will be executed at every cycle. The ME operations from the ME $\mu$TOp instruction queues are directly issued to the corresponding MEs. For the VE operations, the scheduler selects which operations to issue from all VE $\mu$TOp instruction queues. To reclaim a harvested ME, Neu10 performs a context switch to preempt the harvesting $\mu$TOp. Upon a context switch, the register file and the intermediate data in the MEs are saved to SRAM, which incurs negligible overhead compared to the length of an operator. The number of instruction queues should be large enough to support simultaneous execution of all MEs/VEs. For an NPU core with $n_x$ MEs and $n_y$ VEs, there are $n_x$ ME $\mu$TOp instruction queues and $n_y$ VE $\mu$TOp instruction queues.

**$\mu$TOp scheduling policy.** The $\mu$TOp scheduler can be configured in either spatial-isolated or temporal-sharing vNPU scheduling mode, as discussed in §III-C.

With spatial-isolated mode, the scheduler aims to ensure performance isolation. First, if a vNPU has $n_x$ MEs and at least $n_x$ ready ME $\mu$TOps, the scheduler will execute $n_x$ ME $\mu$TOps to fully utilize all the allocated MEs for this vNPU. In this case, no MEs will be harvested from this vNPU. If the allocated MEs are already being harvested by $\mu$TOps from other vNPUs, these $\mu$TOps will be preempted to reclaim the harvested MEs. Second, to improve utilization, if the vNPU has more than $n_x$ ready ME $\mu$TOps, and if another vNPU



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

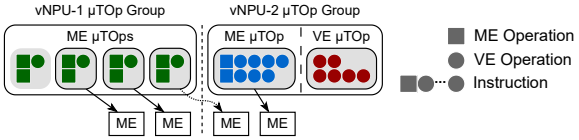
does not have enough ME $\mu$TOps to utilize all its MEs, the scheduler allows the unused MEs to be harvested. A ready VE $\mu$TOp is always executed, as it does not occupy any MEs.

With temporal-sharing mode, as the NPU is oversubscribed, the scheduler maintains fair sharing with the best effort. It uses a priority-based preemptive policy similar to that in previous works [16], [59]. It uses a performance counter to track the active cycles of each vNPU and balances the execution times of vNPUs based on their relative priorities.

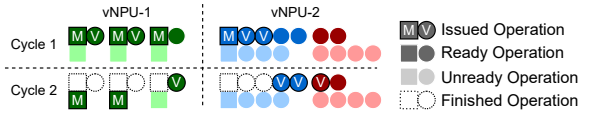
Figure 18(a) shows an example of two vNPUs collocated on an NPU with 4 MEs and 4 VEs with spatial-isolated mapping. Each vNPU has 2 MEs and 2 VEs. Since vNPU-2 only has one ME $\mu$TOp, vNPU-1 can harvest an ME from vNPU-2.

**Operation scheduling policy.** The operation scheduler schedules VE operations using a similar policy as ME $\mu$TOp scheduling. First, the scheduler determines the number of VEs assigned to each vNPU. Then, among all the VE operations in each vNPU, the scheduler prioritizes those from ME $\mu$TOps, which allows the occupied MEs to be freed as soon as possible.

Figure 18(b) shows an example of VE scheduling. In cycle 1, vNPU-1 has 3 ready VE operations and vNPU-2 has 6 ready ones. Each vNPU has 2 VEs, and all VEs are given to operations from ME $\mu$TOps. In cycle 2, vNPU-1 has one ready VE operation, so one of its VEs is harvested by vNPU-2. Since vNPU-2 gets 3 VEs and its ME $\mu$TOp cannot utilize all of them, the remaining VE is given to the VE $\mu$TOp.




(a) NeuISA µTOp scheduling for ME harvesting. Each row of squares and circles inside a µTOp represents an instruction consisting of ME/VE operations. Since vNPU-2 does not have enough ME µTOps, vNPU-1 can harvest an ME from vNPU-2 (shown by the dashed arrow).




(b) NeuISA operation scheduling for VE harvesting. The µTOps in (a) are being executed. "M"/"V" means the operation is issued to an ME/VE.

Fig. 18: ME/VE harvesting in NeuISA. For simplicity, we assume all operations finish in one cycle. In practice, an ME operation takes longer time than a VE operation (see §II-A).

### *F. System Support for NPU Virtualization*

**OS hypervisor.** Neu10 can work with OS hypervisors to provide system support for virtualizing NPUs. Take the KVM hypervisor as a case study, Neu10 leverages `vfio-mdev` to expose vNPUs to VMs as mediated PCIe devices [29]. The hypervisor only mediates the resource management functions that are not on the critical path, including the following hypercalls: (1) create a new vNPU, (2) change the configuration of an existing vNPU, and (3) deallocate a vNPU. The hypercalls are routed to the vNPU manager, which is implemented as a host kernel module. The vNPU manager tracks the allocated and free resources (e.g., MEs/VEs, SRAM, HBM) of all physical NPUs on the host machine and implements the vNPU mapping policies (§III-C). Once a vNPU is set up, the VM can bypass the hypervisor and directly talk to the NPU device. Neu10 uses SR-IOV [54] to expose each vNPU as a PCIe virtual function to the VM via PCIe-passthrough. The IOMMU performs DMA and interrupt remapping for the vNPUs.

**Guest VM software.** Neu10 requires minimal changes to the guest VM software stack. First, the user source code remains unchanged. Typically, user codes are programmed with ML frameworks like PyTorch or TensorFlow [21], [45]. Second, for ML frameworks, only the backend NPU compiler needs to be revised to support NeuISA (§III-D). The ML framework has two parts: (1) The frontend converts the user code into a device-agnostic DNN dataflow graph and optionally partitions the graph onto multiple NPU cores. As our vNPU abstraction reflects the hierarchy of a physical NPU device, the frontend requires no changes. (2) The backend compiles the DNN graph into NPU binary using NeuISA. Third, the NPU vendor will provide a para-virtualized vNPU driver, which is a common practice for virtualizing PCIe devices. The vNPU driver provides user APIs for vNPU management and issues hypercalls to realize these APIs. With PCIe passthrough, the vNPU driver can directly interact with the NPU device [29].

### *G. Neu10 Implementation*

We implement Neu10 with a production-level event-driven NPU simulator. We obtain the operator execution traces for each DNN workload on real Google Cloud TPUs. For each operator, the trace contains the ME/VE time, HBM time, tensor shapes, and the tile sizes and tiling dimensions selected by the compiler. We use the tiling information to generate $\mu$TOps and replay the generated $\mu$TOp traces in our simulator. We modify the frontend of the NPU simulator to implement the $\mu$TOp scheduling and harvesting policy (see §III-E). The scheduler picks $\mu$TOps from multiple traces (each trace represents the DNN workload of a vNPU) and issues them to the backend, which simulates the execution of each ME/VE, on-chip SRAM accesses, and DMA operations to the off-chip HBM at cycle level. To model the penalty of $\mu$TOp preemption (i.e., the context switch overhead of MEs), we set the ME preemption latency to 256 cycles based on the systolic array dimension (i.e., 128×128), including 128 cycles to pop the partial sums and 128 cycles to pop the weights of the preempted $\mu$TOp.

We also prototype the hardware scheduler for NeuISA in Verilog and synthesize it using the FreePDK-15nm cell library [1]. Since the DNN workloads have deterministic dataflow graphs, they do not require complex dependency tracking or speculation in hardware. The hardware area overhead of Neu10 is only 0.04% on a TPUv4 chip. The power overhead of this small extra area is negligible compared to that of the entire chip.



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

TABLE II: NPU simulator configuration.

| | |
|---|---|
| # of MEs/VEs | 4 MEs & 4 VEs |
| ME dimension | 128 × 128 systolic array |
| VE ALU dimension | 128 × 8 FP32 operations/cycle |
| Frequency | 1050 MHz |
| On-chip SRAM | 128 MB |
| HBM Capacity & Bandwidth | 64 GB, 1200 GB/s |

## IV. Discussion

**Support for multi-chip inferences.** Currently, Neu10 supports multi-chip inference with data parallelism by using multiple vNPU chips. As the first step of NPU virtualization, we focus on enabling fine-grained resource sharing on individual NPU chips. In future work, we will extend Neu10 by investigating how to virtualize inter-chip interconnects to support more complicated scenarios (e.g., model parallelism).

**Engineering efforts in developing Neu10.** While Neu10 is a full-stack NPU virtualization design, each component is developed and tested in a modular way, and we minimize changes to the existing system at each level to reduce the debugging and verification efforts. The compiler and hardware changes are minimized as NeuISA reuses the VLIW instruction format in each $\mu$TOp. The guest vNPU driver greatly resembles a native NPU driver thanks to PCIe pass-through, and the major change is the new hypercalls for vNPU management. The KVM hypervisor already provides extensibility for new PCIe devices, and we leverage this feature to integrate the vNPU manager.

**Inter-generational compatibility with NeuISA.** NeuISA enables a DNN program to run on different numbers of MEs/VEs without recompilation. This greatly eases the effort to provide compatibility across generations of NPU hardware. NeuISA could ease the future development efforts of ML frameworks to support new NPU hardware and enable more flexible and transparent ways to manage NPU resources. Neu10 provides a general vNPU abstraction, which allows a vNPU to be mapped to different generations of NPU hardware.

## V. Evaluation

Our evaluation shows that: (1) Neu10 provides performance isolation with up to 4.6× reduction in tail latency, while improving the ML service throughput by 1.4× over state-of-the-art NPU sharing approaches (§V-B); (2) It improves the NPU utilization by 1.2× (§V-C); (3) It scales as we change the number of MEs/VEs (§V-E); (4) It benefits multi-tenant ML services with various HBM bandwidths (§V-F).

### A. Experimental Setup

We evaluate DNN workloads (see Table I) from MLPerf v2.1 [46] and the official TPU reference models [22]. To test Neu10 under different workload combinations, we select workload pairs with low ME/VE contention (`DLRM+SMask`, `DLRM+RtNt`, `NCF+RsNt`), medium contention (`ENet+SMask`, `BERT+ENet`, `ENet+MRCN`), and high contention (`ENet+TFMR`, `MNIST+RtNt`, `RNRS+RtNt`). The batch size is 32 except for `MRCN` and `SMask` (batch size is 8 for them). Each workload runs on a vNPU with 2 MEs and 2 VEs. We map two vNPUs to a physical NPU core with 4 MEs and 4 VEs as listed in Table II. The SRAM and HBM capacity is evenly partitioned between the vNPUs. To obtain steady-state performance, we run inference requests continuously for each workload until all collocated workloads have completed a certain number of requests. We compare the following designs:

- **PMT** [16]: temporal-sharing of the entire NPU core among multiple vNPUs. A preemptive fair scheduling mechanism is employed for performance isolation.
- **V10** [59]: temporal-sharing of all MEs and VEs among the vNPUs, with a priority-based preemptive policy. The workload is compiled with the traditional VLIW-style ISA. If an ME operator from one vNPU is running, only VE-only operators from collocated vNPUs can execute simultaneously.
- **Neu10-NoHarvest (Neu10-NH)**: spatial-isolated vNPUs with dedicated MEs/VEs without dynamic scheduling. This resembles existing static partitioning techniques such as NVIDIA Multi-instance GPU (MIG) [41].
- **Neu10**: spatial-isolated vNPUs with dynamic resource scheduling and harvesting enabled by NeuISA.

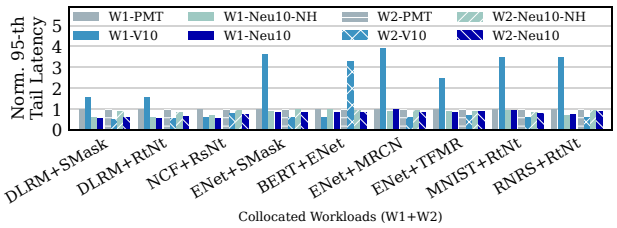



Fig. 19: 95% Percentile latency of Neu10 (normalized to PMT).

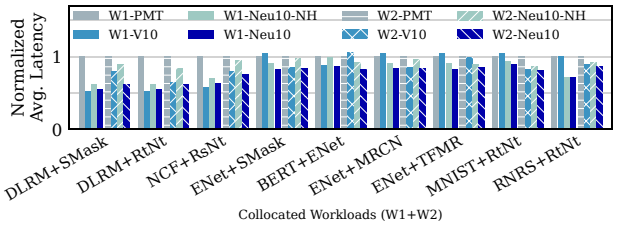



Fig. 20: Average request latency of Neu10 (normalized to PMT).

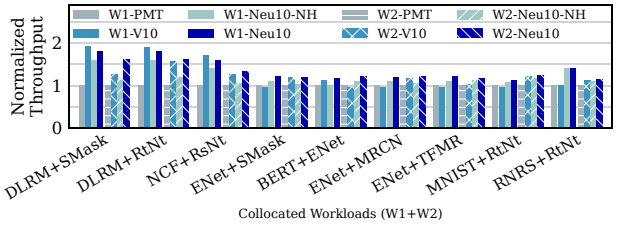
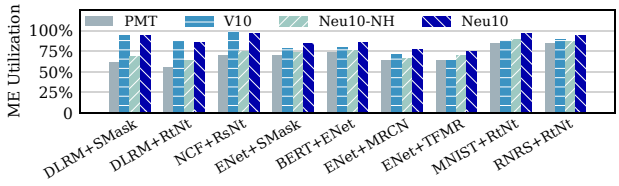


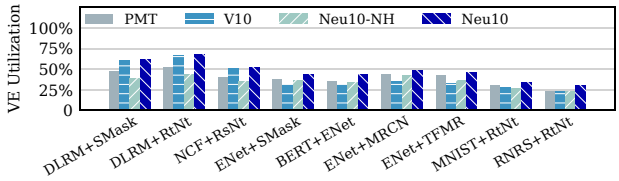

Fig. 21: Throughput of Neu10 (normalized to PMT).



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.




(a) Total ME utilization of the NPU core.




(b) Total VE utilization of the NPU core.

Fig. 22: Total utilization of MEs and VEs.

### B. Performance of Neu10

**Tail Latency.** Figure 19 shows that Neu10 improves the 95% tail latency over V10 by up to 4.6× (1.56× on average). V10 primarily focuses on maximizing the utilization. Thus, even with an operator preemption mechanism, it still fails to enforce performance isolation between vNPUs, due to complex inter-operator dependencies and imbalanced operator lengths.

In contrast, Neu10 ensures performance isolation between vNPUs while opportunistically improving their performance by harvesting. As Neu10 only harvests the underutilized compute units, the performance interference between vNPUs is minimized. Hence, a harvested workload in Neu10 experiences negligible tail latency overhead compared to that in Neu10-NH. In a few cases (e.g., `ENet+MRCN` and `RNRS+RtNt`), harvesting increases the burden on memory bandwidth, which may slightly impact the tail latency. However, Neu10 still achieves much better tail latency than PMT and V10.

**Average Latency.** Neu10 improves the average latency of inference requests by 1.33× over PMT and 1.12× over V10 on average (Figure 20). While both V10 and Neu10 perform dynamic scheduling to utilize the NPU hardware, Neu10 greatly reduces ME contentions with $\mu$TOp-level scheduling. V10 treats all MEs on a physical NPU core as a whole unit, due to the VLIW ISA limitation. This causes false contentions on the MEs when an operator cannot fully exploit the MEs but still fully occupies them. In contrast, Neu10 eliminates such contention by assigning MEs to different operators.

**Throughput.** Figure 21 shows the throughput of the collocated workloads. When the ME/VE contention is low, both V10 and Neu10 improve the throughput significantly over PMT (by 1.58× and 1.62× on average), as the major benefit comes from overlapping the execution of ME-intensive operators and VE-intensive operators. When the ME/VE contention is high, Neu10 improves the throughput of DNN workloads over V10 by up to 1.41×, since Neu10 offers more flexibility for dynamic ME/VE scheduling with $\mu$TOps, as discussed above.

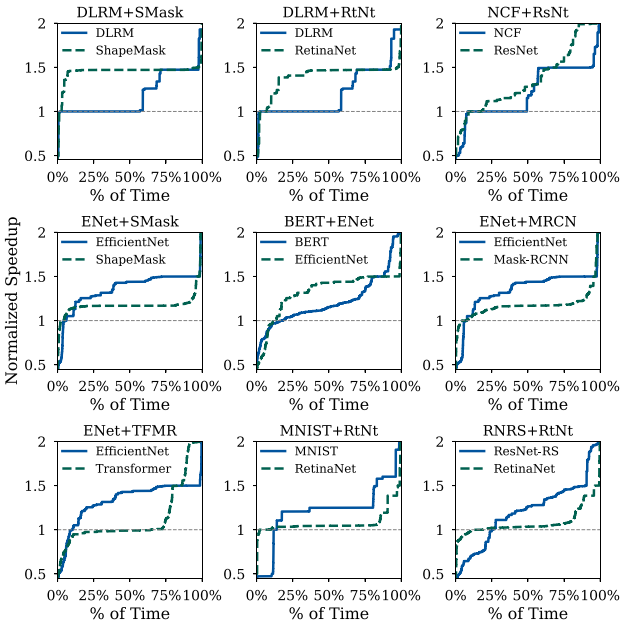



Fig. 23: Benefit breakdown of ME/VE harvesting. Y-axis is the speedup of Neu10 over Neu10-NH. X-axis is the percentage of time when executing on Neu10-NH. The curves below $Y = 1$ indicate the slowdown of operators due to interference. The curves above $Y = 1$ indicate the speedup due to harvesting.

TABLE III: The harvesting overhead in each workload, quantified by how much time a workload is blocked due to being harvested over the end-to-end execution time of the workload. "<0.01%" means the overhead is smaller than 0.01%, which rounds to 0 when we only preserve two decimals. For all workloads, the overhead of being harvested is completely outweighed by the benefit of harvesting.

| Collocated Workloads (W1+W2) | Overhead | |
|---|---|---|
| | W1 | W2 |
| DLRM+SMask | 2.47% | 0.01% |
| DLRM+RtNt | 2.54% | <0.01% |
| NCF+RsNt | 6.16% | <0.01% |
| ENet+SMask | 5.31% | 1.12% |
| BERT+ENet | <0.01% | 5.54% |
| ENet+MRCN | 5.17% | 1.00% |
| ENet+TFMR | 5.61% | 0.15% |
| MNIST+RtNt | 10.63% | 1.74% |
| RNRS+RtNt | 7.33% | 2.21% |

### C. Resource Utilization Improvement

We show the utilization of the MEs and VEs on the NPU core in Figure 22. With dynamic operator scheduling, Neu10 improves the ME and VE utilization by 1.26× and 1.2× over PMT on average. For some workload pairs, Neu10 achieves slightly better utilization than V10, since Neu10 has less preemption overhead with $\mu$TOp scheduling. Specifically, V10 needs to preempt the entire operator from all MEs, while Neu10 only preempts the $\mu$TOps on the harvested MEs, such that the remaining $\mu$TOps can continue execution.



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

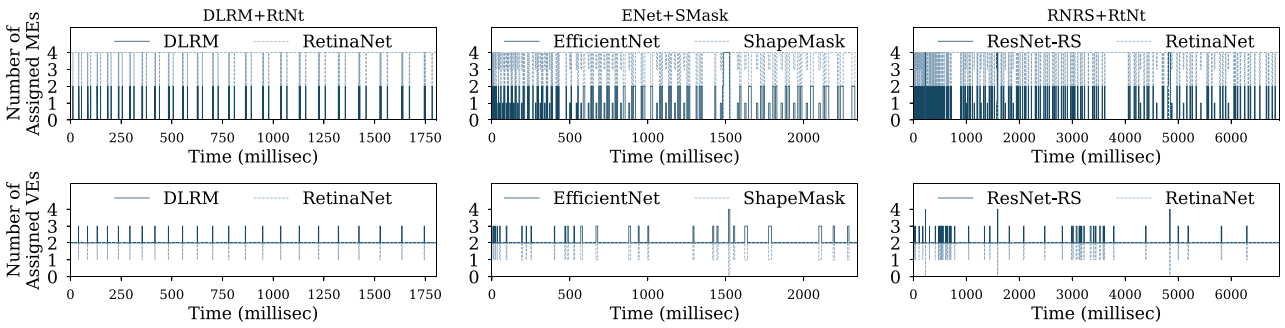



Fig. 24: Breakdown of the number of assigned MEs/VEs over time for different DNN workload combinations.

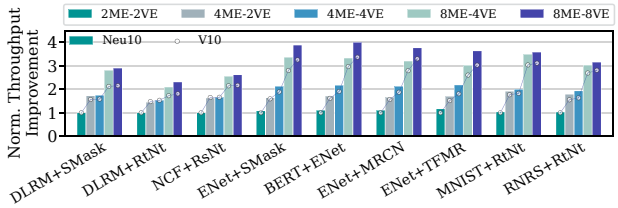



Fig. 25: Throughput improvement of Neu10 with varying numbers of MEs and VEs over V10 with 2 MEs and 2 VEs.

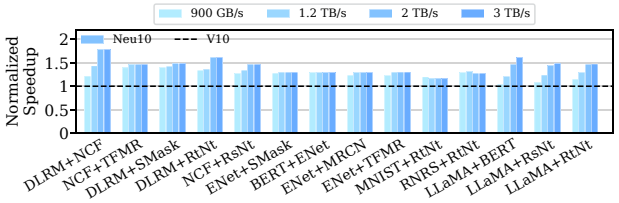



Fig. 26: Throughput improvement of Neu10 with varying HBM bandwidth (normalized to V10).

### D. Benefit Breakdown of ME/VE Harvesting

To better understand the benefit and overhead of harvesting, we trace the speedup of each operator in Neu10 over Neu10-NH, and show the impact in Figure 23.

For workload pairs with low ME/VE contention (the first row in Figure 23), most operators achieve at least 1.5× speedup by harvesting unused compute units from the collocated vNPU with negligible performance interference. For workload pairs with high ME/VE contention (the last row in Figure 23), harvesting causes performance degradation for some operators. Harvesting may incur extra power and performance overhead (3.12% on average) when an operator is blocked due to being harvested. We summarize the harvesting overhead in Table III. Although harvesting causes slowdowns for some operators, the overall speedup of the workload still outweighs the slowdowns.

To visualize the behavior of Neu10's dynamic ME/VE scheduling, we trace the number of MEs and VEs assigned to each collocated workload at runtime in Figure 24. As the ME/VE demands of the workloads vary across time, the ME-intensive workload (e.g., `RetinaNet` and `ShapeMask`) attempts to harvest the unused MEs from the collocated workload. The VEs are harvested similarly.

### E. Impact of Varying MEs and VEs

To show Neu10's benefits on different hardware configurations, we vary the numbers of MEs and VEs on the physical NPU core and evenly partition the core between the two collocated vNPUs. We compare Neu10 with V10, as V10 has fine-grained preemption, which serves as the most competitive baseline. We show the throughput in Figure 25. With more MEs/VEs, Neu10 brings more benefits, since there is more flexibility for dynamic ME/VE scheduling. With more MEs/VEs, it is more likely that an operator cannot fully exploit all ME/VEs. Therefore, the benefit of $\mu$TOp-level scheduling and harvesting becomes more obvious.

### F. Impact of Varying Memory Bandwidth

We show Neu10's performance under different HBM bandwidth configurations in Figure 26. In most cases, Neu10 achieves similar throughput benefits. This is because many ML inference workloads suffer from the ME/VE contention rather than HBM bandwidth contention (see Figure 7). To understand the impact of memory bandwidth contention, we collocate two memory-intensive workloads (i.e., `DLRM+NCF` and `NCF+TFMR`). Even with low available memory bandwidth (e.g., 900 GB/s), Neu10 still outperforms the time sharing-based scheme V10. With more available bandwidth, Neu10 brings more benefits for memory-intensive workloads, since higher bandwidth helps alleviate memory contention.

For memory-intensive workloads, Neu10 enables them to be collocated with compute-intensive workloads following existing workload collocation approaches [12], [37], [59], which helps cloud platforms better utilize both compute and memory resources. As a case study, we collocate a memory bandwidth-intensive LLM inference workload, LLaMA2-13B [51] (`LLaMA`), with compute-intensive workloads (i.e., `BERT`, `RsNt`, and `RtNt`). As shown in Figure 27, with V10, when LLaMA temporarily occupies all MEs/VEs, it underutilizes the MEs/VEs since the execution is bounded by



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

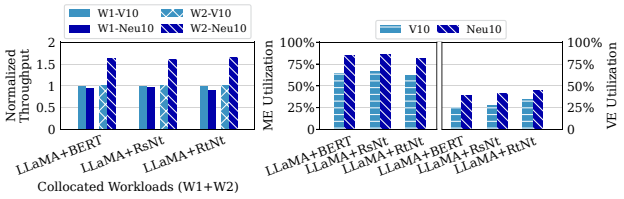



Fig. 27: Performance of collocating LLM (LLaMA-13B with batch size 8 and input sequence length 512) and other models in Neu10.

memory bandwidth. However, the underutilized MEs cannot be harvested by the collocated workload due to the temporal sharing mechanism. Neu10 enables the spatial sharing of the MEs/VEs, so the collocated workload can harvest the spare MEs/VEs for throughput improvement (by up to 1.6×). Meanwhile, LLaMA suffers from negligible overhead while using fewer MEs/VEs. As hardware vendors continue to scale the HBM bandwidth, Neu10 will bring more benefits because of the alleviation of memory contention. Note that the owners of LLM inference service can follow the pay-as-you-go model to allocate multiple vNPUs with large memory. The vNPU abstraction of Neu10 offers the flexibility for resource allocation while enabling dynamic scheduling.

## VI. Related Work

**System virtualization for accelerators.** As we employ hardware accelerators for ML services, cloud platforms prefer to virtualize them for improved resource utilization [6], [26], [33], [34], [55], [63], [64]. Prior studies have investigated virtualization techniques for GPUs [36], [41], [42], [49] and FPGAs [33], [34], [38], [63], [64]. Unfortunately, these techniques cannot be directly applied to NPUs, as they target different architectures. AvA [62] investigates hypervisor interposition techniques for virtualizing accelerators. However, they do not focus on improving the resource utilization. To the best of our knowledge, Neu10 is the first to investigate the system and architectural techniques for NPU virtualization.

While different tensor processors have been developed recently [8], [30], [35], [47], most of them have specialized matrix engines and generic vector engines, given the continuing trend that DNN computations are dominated by these operations. As the imbalanced ME/VE demands are intrinsic to DNN workloads (§II-B), these processors also suffer from resource underutilization. The design of Neu10 can be adapted to virtualize these accelerators for utilization improvement.

**Accelerator resource sharing and scheduling.** There have been various techniques [12], [13], [26], [33], [34], [38], [40], [57], [58], [63], [64] for supporting multi-tenant workloads on accelerators. PREMA [16] proposed a preemptive scheduling mechanism, but it causes high context-switch overhead. Prior studies [9], [43], [59] investigated the imbalance of compute units and memory. Planaria [19] studied the spatial underutilization of systolic arrays. V10 [59] enabled fine-grained preemption. There are also software techniques to fuse DNN workloads at graph level [60], [61]. However, they force two DNN inference tasks to launch together, which cannot work for the unpredictable incoming requests in the cloud. None of them systematically enables NPU virtualization. Neu10 addressed the systems and architectural challenges of NPU virtualization.

As the resource demand of a DNN workload changes drastically over time (see §II-B), a static resource allocation is insufficient. Neu10 proposes NeuISA and enables dynamic scheduling to mitigate the underutilization, it is orthogonal to the higher-level workload collocation techniques.

**Architectural support for virtualization.** Prior studies have proven that architectural techniques are effective for facilitating system virtualization [10], including Intel VT-x and AMD SVM for CPU virtualization [3], Intel EPT [53] and AMD NPT for memory address translation [11], and SR-IOV for I/O virtualization [18], [54]. Similarly, NPU virtualization also needs architectural support. In this work, we identify the unique architectural challenges (see §I) with NPU virtualization, and present the corresponding ISA extension and architectural supports for enabling fine-grained NPU virtualization.

## VII. Conclusion

We identify the key challenges of virtualizing NPUs for cloud platforms, including the need for fine-grained system abstraction and resource scheduling and the necessity of architectural support. We present a holistic solution Neu10 for enabling NPU virtualization. It improves both NPU utilization and performance isolation for multi-tenant ML services.

## Acknowledgements

We thank the anonymous reviewers for their helpful comments and feedback. We thank Haoyang Zhang for his insightful discussion on the NeuISA design. This work was partially supported by NSF grant CCF-1919044, NSF CAREER Award CNS-2144796, and the Hybrid Cloud and AI program at the IBM-Illinois Discovery Accelerator Institute (IIDAI).

## References

[1] "FreePDK15." [Online]. Available: https://eda.ncsu.edu/freepdk15/
[2] "Memory segmentation." [Online]. Available: https://en.wikipedia.org/wiki/Memory_segmentation
[3] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, USA, 2006.
[4] Altexsoft, "Comparing Machine Learning as a Service: Amazon, Microsoft Azure, Google Cloud AI, IBM Watson," 2021. [Online]. Available: https://www.altexsoft.com/blog/datascience/comparing-machine-learning-as-a-service-amazon-microsoft-azure-google-cloud-ai-ibm-watson/
[5] AMD, "AI Engine: Meeting the Compute Demands of Next-Generation Applications," 2023. [Online]. Available: https://www.xilinx.com/products/technology/ai-engine.html
[6] AWS, "Amazon EC2 F1 Instances," 2022. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/
[7] A. AWS, "Machine Learning on AWS Innovate faster with the most comprehensive set of AI and ML services," 2022. [Online]. Available: https://aws.amazon.com/machine-learning/
[8] A. AWS, "Aws inferentia," 2023. [Online]. Available: https://aws.amazon.com/machine-learning/inferentia/



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

[9] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*, Virtual Event, 2020.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, 2003.

[11] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, Seattle, WA, USA, 2008.

[12] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, Xi'an, China, 2017.

[13] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Atlanta, GA, 2016.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Carlsbad, CA, 2018.

[15] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, Salt Lake City, UT, 2014.

[16] Y. Choi and M. Rhu, "PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*, San Diego, CA, USA, 2020.

[17] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, T. Juhasz, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao, and D. Burger, "Accelerating Persistent Neural Networks at Datacenter Scale," in *Proceedings of HotChips'17*, Cupertino, CA, USA, 2017.

[18] T. P. P. de Lacerda Ruivo, G. B. Altayo, G. Garzoglio, S. Timm, H. W. Kim, S.-Y. Noh, and I. Raicu, "Exploring infiniband hardware virtualization in opennebula towards efficient high-performance computing," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'14)*, Chicago, IL, USA, 2014.

[19] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*, Virtual Event, 2020.

[20] Google, "System architecture - cloud TPU," 2022. [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

[21] Google, "Create production-grade machine learning models with TensorFlow," 2023. [Online]. Available: https://www.tensorflow.org/

[22] Google, "Supported reference models," 2023. [Online]. Available: https://cloud.google.com/tpu/docs/tutorials/supported-models

[23] Google, "XLA: Optimizing Compiler for Machine Learning," 2023. [Online]. Available: https://www.tensorflow.org/xla

[24] Graphcore, "Graphcore IPU overview," 2022. [Online]. Available: https://www.graphcore.ai/products/ipu

[25] L. Gwennap, "Tenstorrent scales ai performance: New multicore architecture leads in data-center power efficiency," 2020. [Online]. Available: https://www.linleygroup.com/mpr/article.php?id=12287

[26] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences," in *Proccedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, Carlsbad, CA, USA, 2022.

[27] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[28] J. Hui, "AI Chips Technology Trends and Landscapes (Mobile SoC, Intel, Asian AI Chips, Low-Power Inference Chips)," 2020. [Online]. Available: https://jonathan-hui.medium.com/ai-chips-technology-trends-landscape-mobile-soc-intel-asian-ai-chips-low-power-inference-4db701dbe85d

[29] N. Jia and K. Wankhede, "Vfio mediated devices," 2023. [Online]. Available: https://docs.kernel.org/driver-api/vfio-mediated-device.html

[30] Y. Jiao, L. Han, and X. Long, "Hanguang 800 npu – the ultimate ai inference solution for data centers," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, Palo Alto, CA, USA, 2020.

[31] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, June 2020.

[32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA'17)*, Toronto, Canada, 2017.

[33] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Carlsbad, CA, USA, 2018.

[34] J. Landgraf, T. Yang, W. Lin, C. J. Rossbach, and E. Schkufza, "Compiler-driven fpga virtualization with synergy," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, Virtual Event, 2021.

[35] H. Liao, J. Tu, J. Xia, and X. Zhou, "Davinci: A scalable architecture for neural network computing," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, Los Alamitos, CA, USA, 2019.

[36] Z. Lin, L. Nyland, and H. Zhou, "Enabling efficient preemption for simt architectures with lightweight context switching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*, Salt Lake City, UT, USA, 2016.

[37] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, Portland, OR, USA, 2015.

[38] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory fpga platforms," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[39] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, 2011.

[40] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram, "Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, Carlsbad, CA, USA, 2022.

[41] Nvidia, "Multi-Instance GPU user guide," 2022. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/

[42] Nvidia, "Virtual GPU software user guide," 2022. [Online]. Available: https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/

[43] Y. H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *2021 IEEE*




Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.

*International Symposium on High-Performance Computer Architecture (HPCA'21)*, Seoul, Korea, 2021.

[44] E. Onose, "Machine learning as a service: What it is, when to use it and what are the best tools out there," 2022. [Online]. Available: https://neptune.ai/blog/machine-learning-as-a-service-what-it-is-when-to-use-it-and-what-are-the-best-tools-out-there

[45] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," in *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'17)*, Long Beach, CA, USA, 2017.

[46] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," 2020.

[47] RUN:AI, "Google TPU Architecture and Performance Best Practices," 2022. [Online]. Available: https://www.run.ai/guides/cloud-deep-learning/google-tpu

[48] Stephen J. Bigelow, "pay-as-you-go cloud computing (PAYG cloud computing)," 2022. [Online]. Available: https://www.techtarget.com/searchstorage/definition/pay-as-you-go-cloud-computing-PAYG-cloud-computing

[49] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[50] The KubeVirt Contributors, "Kubevirt.io," 2023. [Online]. Available: https://kubevirt.io/

[51] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[52] A. Vahdat and M. Lohmeyer, "Enabling next-generation ai workloads: Announcing tpu v5p and ai hypercomputer," 2023. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/introducing-cloud-tpu-v5p-and-ai-hypercomputer

[53] VMWare, "Performance Evaluation of Intel EPT Hardware Assist," 2009. [Online]. Available: https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf

[54] VMWare, "vSphere Networking," 2009. [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/8.0/vsphere-esxi-vcenter-802-networking-guide.pdf

[55] K. Wiggers, "Microsoft and nvidia team up to build new azure-hosted ai supercomputer," 2022. [Online]. Available: https://techcrunch.com/2022/11/16/microsoft-and-nvidia-team-up-to-build-new-azure-hosted-ai-supercomputer/

[56] Wolfram Alpha LLC, "Wolframalpha: Computational intelligence," 2023. [Online]. Available: https://www.wolframalpha.com/

[57] Y. Xue, Y. Liu, and J. Huang, "System virtualization for neural processing units," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, Providence, RI, USA, 2023.

[58] Y. Xue, Y. Liu, L. Nai, and J. Huang, "Hardware-assisted virtualization for neural processing units," in *The 1st Workshop on Hot Topics in System Infrastructure (HotInfra'23)*, Orlando, FL, USA, 2023.

[59] Y. Xue, Y. Liu, L. Nai, and J. Huang, "V10: Hardware-assisted npu multi-tenancy for improved resource utilization and fairness," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA'23)*, Orlando, FL, USA, 2023.

[60] Q. Yang, T. Yang, M. Xiang, L. Zhang, H. Wang, M. Serafini, and H. Guan, "GMorph: Accelerating multi-dnn inference via model fusion," in *Proceedings of the 19th European Conference on Computer Systems (EuroSys'24)*, Athens, Greece.

[61] F. Yu, S. Bray, D. Wang, L. Shangguan, X. Tang, C. Liu, and X. Chen, "Automated runtime-aware scheduling for multi-tenant dnn inference on gpu," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[62] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach, "AvA: Accelerated virtualization of accelerators," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[63] Y. Zha and J. Li, "Virtualizing fpgas in the cloud," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[64] Y. Zha and J. Li, "When application-specific isa meets fpgas: A multi-layer virtualization framework for heterogeneous cloud fpgas," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, Virtual Event, 2021.

[65] H. Zhang, Y. Zhou, Y. Xue, Y. Liu, and J. Huang, "G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, Toronto, ON, Canada, 2023.

[66] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui, F. Yang, M. Yang, L. Zhou, A. Cidon, and G. Pekhimenko, "ROLLER: Fast and efficient tensor compilation for deep learning," in *Proceedings of 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, Carlsbad, CA, USA, 2022.



Authorized licensed use limited to: University of Illinois. Downloaded on March 07,2025 at 16:15:55 UTC from IEEE Xplore. Restrictions apply.