Topology-Aware Virtualization over Inter-Core Connected Neural Processing Units

Dahu Feng*
fengdh21@mails.tsinghua.edu.cn
CBICR, Tsinghua University
Beijing, China

Erhu Feng*
fengerhu1@sjtu.edu.cn
IPADS, Shanghai Jiao Tong University
Shanghai, China

Dong Du
Dd_nirvana@sjtu.edu.cn
IPADS, Shanghai Jiao Tong University
Shanghai, China

Pinjie Xu
xupinjie321@outlook.com
SenseTime Research
Beijing, China

Yubin Xia
xiayubin@sjtu.edu.cn
IPADS, Shanghai Jiao Tong University
Shanghai, China

Haibo Chen
haibochen@sjtu.edu.cn
IPADS, Shanghai Jiao Tong University
Shanghai, China

Rong Zhao†
r_zhao@tsinghua.edu.cn
Center for Brain-Inspired Computing Research, IDG/McGovern Institute for Brain Research and Department of Precision Instrument
Tsinghua University
Beijing, China

Abstract

With the rapid development of artificial intelligence (AI) applications, an emerging class of AI accelerators, termed Inter-core Connected Neural Processing Units (NPU), has been adopted in both cloud and edge computing environments, like Graphcore IPU, Tenstorrent, etc. Despite their innovative design, these NPUs often demand substantial hardware resources, leading to suboptimal resource utilization due to the imbalance of hardware requirements across various tasks. To address this issue, prior research has explored virtualization techniques for monolithic NPUs, but has neglected inter-core connected NPUs with the hardware topology.

This paper introduces vNPU, the first comprehensive virtualization design for inter-core connected NPUs, integrating three novel techniques: (1) NPU route virtualization, which redirects instruction and data flow from virtual NPU cores to physical ones, creating a virtual topology; (2) NPU memory virtualization, designed to minimize translation stalls for SRAM-centric and NoC-equipped NPU cores, thereby maximizing the memory bandwidth; and (3) Best-effort topology mapping, which determines the optimal mapping from all candidate virtual topologies, balancing resource utilization with end-to-end performance. We have developed a prototype of vNPU on both an FPGA platform (Chipyard+FireSim) and a simulator (DCRA). Evaluation results demonstrate that when executing multiple NPU workloads on virtual NPUs, vNPU achieves performance improvements of up to 1.92x and 1.28x for the Transformer and ResNet models, respectively, in comparison to the MIG-based virtualization method. Furthermore, the hardware performance overhead associated with the virtualization itself is minimal, incurring less than 1% reduction in end-to-end performance.

CCS Concepts

• **Hardware** → *On-chip resource management*.

Keywords

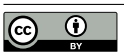
Virtualization, System-on-Chip, Accelerator, AI

ACM Reference Format:
Dahu Feng, Erhu Feng, Dong Du, Pinjie Xu, Yubin Xia, Haibo Chen, and Rong Zhao. 2025. Topology-Aware Virtualization over Inter-Core Connected Neural Processing Units. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3695053.3731002

1 INTRODUCTION

With the rising popularity of AI applications such as ChatGPT [55], self-driving [67], and AI agent [74], machine learning has become integral to both cloud and edge computing environments. However, the significant computational demands of these AI workloads cannot be efficiently satisfied by CPUs or even GPUs. To maximize the utilization of hardware computing resources, manufacturers have introduced specialized AI accelerators known as Neural Processing Units (NPUs). There are two primary types of NPU designs: inter-core connected NPUs and monolithic NPUs (lacking inter-core connections). In this paper, we mainly focus on the inter-core connected NPU, which is a more powerful NPU implementation that adopts a data flow architecture. Examples of such NPUs include

*Both authors contributed equally to this research.
†Corresponding author.



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3731002

the Graphcore IPU [27], AWS NeuronCore [6], Tenstorrent [69], and Groq [1]. These inter-core connected NPUs demonstrate superior performance by efficiently leveraging the data flow inherent in AI workloads. Recent studies [49] have shown that by utilizing inter-core connections, the IPU can outperform the A100 GPU by a factor of 3.3x, while using only 80% of the available FLOPS.

Meanwhile, current AI models also exhibit significant variation in model size, leading to diverse requirements for hardware resources. For example, a classic ResNet-50 model contains only 25 million parameters [32], while contemporary large language models (LLMs) like Llama3 offer a wide spectrum of sizes, including options with 8, 70 and 405 billion parameters [71]. Nevertheless, the hardware design of inter-core connected NPUs has been progressively scaled up, which mainly concentrates on how to accommodate these larger models, but often results in suboptimal resource utilization for traditional, smaller ML models such as convolutional neural networks (CNNs). To bridge the gap between NPU hardware design and the diverse requirements of various ML models, there is a growing demand for **NPU virtualization** — *a technique that can provide multiple virtual NPUs for different users/tasks based on one physical NPU, and different virtual NPUs have strong isolation, configurable NPU capacity, and hardware agnosticism.*

Designing a virtualization mechanism for inter-core connected NPUs presents several challenges due to their fundamentally different architectures compared to GPUs and monolithic NPUs. First, prior efforts only explored virtualization for SIMT architectures (e.g., GPU virtualization solutions), but can not tackle spatially programmed accelerators like inter-core connected NPUs. Spatially programmed accelerators utilize a data flow architecture with the hardware topology, where each NPU core occupies a unique topological position and is interconnected with other cores. This setup allows for direct data transfers between NPU cores without the need for additional load/store operations. In contrast, SIMT accelerators employ thousands of identical hardware threads/cores within a von Neumann architecture, where computing tasks can be offloaded to any available threads or cores, and leverage global memory to exchange intermediate results.

Second, previous efforts have primarily addressed memory virtualization within the classical memory hierarchy (cache and global memory) used by CPUs and GPUs. In contrast, inter-core connected NPUs employ an SRAM-centric memory system. To optimize memory bandwidth for NPU cores, there is no cache coherence between the on-chip SRAM and off-chip global memory. Instead, data transfer between global memory and on-chip SRAM is orchestrated through DMA operations at a coarse granularity, which proves inadequate for the fixed-size, fine-grained page-level memory virtualization [20, 36]. Therefore, due to these hardware differences, virtualizing the NPU topology with the SRAM-centric memory system is essential for inter-core connected NPUs. However, these aspects remain unaddressed by existing virtualization mechanisms, including the GPU virtualization [16, 63, 70, 73, 79, 81] and monolithic NPU virtualization [41, 77].

This paper introduces the first comprehensive virtualization design for inter-core connected NPUs, termed vNPU. Different from traditional virtualization approaches for CPUs and GPUs, vNPU specifically targets topology virtualization with data flow architectures. It incorporates three key techniques for NPU virtualization:

- **vRouter for NPU route virtualization:** Given that the inter-core connected NPU features a hardware topology among its cores, it incorporates a router for instruction dispatch and data transmission within the on-chip network. Consequently, a virtual NPU must also inherit a virtual topology, and employ a virtual router to efficiently redirect both instruction flows and data flows from the virtual NPU core to the appropriate physical NPU core.
- **vChunk for memory virtualization within NPU topology:** The inter-core connected NPU exhibits distinct memory hierarchy for its SRAM-centric memory system and NoC-based interconnection. Unlike the classical memory system that relies on load/store instructions in the cache granularity, NPUs employ DMA operations to transfer a large chunk of model weights from backup memory to SRAM, and further eliminate load/store requests for intermediate results using the inter-core connection. Consequently, page-based memory virtualization is inefficient in the NPU scenario, as a TLB miss can obstruct substantial data transfers. vNPU introduces a range-based memory virtualization that fully utilizes NPU's memory access patterns.
- **Topology mapping for NPU core allocation:** Since a virtual NPU also requires a hardware topology for NPU cores, this presents a new challenge in allocating NPU cores while considering resource utilization. To address this problem, the hypervisor can adopt various topology mapping strategies, such as exact mapping and similar topology mapping. vNPU analyzes these different strategies and proposes a balanced solution that optimizes both NPU utilization and performance for ML tasks.

We have developed a prototype of vNPU on both an FPGA platform (Chipyard [3]) for micro tests and a software simulator (DCRA [50, 56]) to evaluate large-scale real-world ML applications. Our implementation extends the Gemmini NPU [23] with inter-core connections, featuring an architecture akin to the Graphcore IPU [27]. On the software side, we have modified the KVM module in the Linux kernel to manage all meta tables for virtual NPUs. Evaluation results show that the hardware extensions for NPU virtualization incur only negligible overhead (about 1% ∼ 2%) in micro-benchmark, and less than 1% reduction in end-to-end evaluations. With the support of virtual topology, vNPU achieves higher resource utilization through more flexible NPU core allocation. As a result, in multi-task scenarios, vNPU delivers up to a 1.92x and 1.28x performance improvement for Transformer and ResNet models compared to the SOTA virtualization mechanism like MIG.

2 BACKGROUND

2.1 Inter-core Connected NPU Architecture

The inter-connected NPU represents an emerging class of data flow accelerators designed for AI applications. While there are various NPU implementations such as Graphcore IPU [27], AWS's NeuronCore [6], Tenstorrent [69], DOJO [68], Sambanova [58], Simba [62], MTIA [21], Cerebras [47] and Groq [1], these NPUs exhibit several common characteristics. As depicted in Figure 1, a typical inter-core connected NPU comprises multiple NPU cores, each equipped with its own on-chip interconnection and SRAM. To optimize memory bandwidth for AI workloads, these NPUs leverage (1) inter-core connections to directly transfer intermediate results to target nodes,

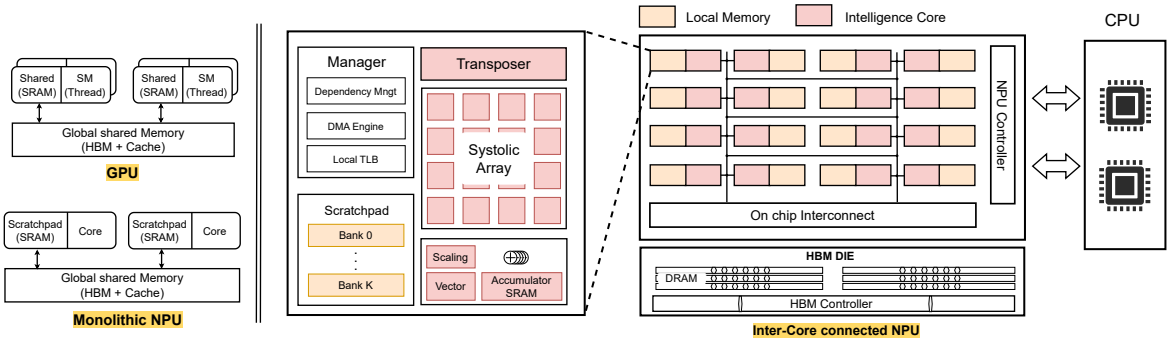


Figure 1: An inter-core connected NPU contains multiple cores with the interconnection, on-chip SRAM and off-chip HBM.

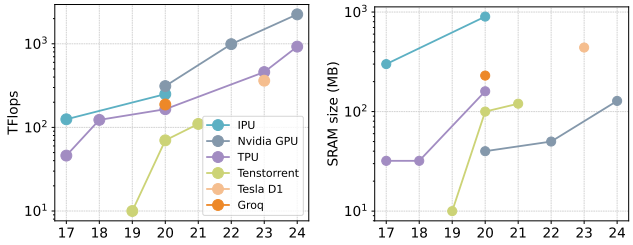
and (2) a simplified memory architecture utilizing large local memory (i.e., scratchpad) without address association or coherence with global memory. The data transfer between global memory and local memory is facilitated by the DMA engine. As for computation, NPUs are equipped with matrix computation units (e.g., systolic arrays) and vector units. For instance, a typical inter-core connected NPU like Graphcore's IPU features thousands of cores on a single chip and up to 900MB of on-chip SRAM. As for the interconnection, IPU achieves all-to-all 65TB/s aggregated bandwidth for on-chip network. Similarly, SambaNova [58] comprises 1,040 RDU cores with 520MB of SRAM and 64GB of HBM.

Other NPUs, such as Apple NPU, Intel NPU, and Qualcomm NPU, are integrated into mobile System-on-Chips (SoCs) and may not employ inter-core connections due to hardware resource constraints. In this paper, we do not concentrate on these NPU architectures; instead, we can apply existing solutions [41, 77] to address the virtualization challenges with such NPUs.

2.2 Underutilization of NPU Resources

As shown in Figure 2, the latest NPUs [1, 27, 38] are equipped with a large amount of computing units (>100 TFLOPS), on-chip memory (>200MB SRAM), and external high-bandwidth memory (>50GB HBM), to support the substantial computational demands of LLMs. However, traditional small-scale ML models such as CNNs remain prevalent in both cloud and edge computing. Platforms like Google [25] and Amazon [7] offer a range of ML APIs that cater to tasks such as label detection, object localization, and etc. These tasks typically leverage smaller ML models, like ResNet [32], GoogleNet [65] and MXNet [12]. Moreover, the size of LLMs varies significantly, ranging from the tiny LLMs like Qwen2-0.5B [2] to giants LLMs such as GPT-3.5 [55] with 175 billion parameters. Therefore, deploying relatively small ML models on large NPU chips can lead to significant resource underutilization.

We evaluate different ML models on a widely-used cloud NPU: Google TPU [38], as shown in Figure 3. The results indicate that the majority of traditional ML models utilize less than 50% of the TPU core's FLOPS. Even increasing the batch size during model inference did not fully capitalize on the TPU's total FLOPS capacity. This underutilization stems from the imbalance between hardware resources, including FLOPS, memory size, and memory bandwidth, across various ML models. Moreover, different phases during model inference exhibit distinct resource demands [57]. The prefill phase of transformer-based models is computing-intensive,



(a) FLOPS (b) SRAM

Figure 2: The evolution of NPU hardware resources: FLOPS and SRAM (2017–2024).

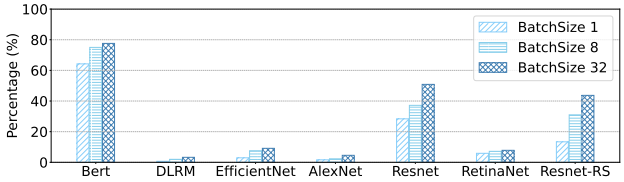


Figure 3: Overall FLOPS utilization in Google TPU with different ML workloads.

while the decode phase is memory-intensive. Therefore, to fully utilize all hardware resources in current large NPU chips, there is an imperative need for NPU virtualization to provide various resource combinations for virtual NPUs.

2.3 Virtualization for AI Accelerators

Current cloud vendors have adopted various virtualization mechanisms for AI accelerators (GPU, NPU), as shown in Table 1. Considering GPU virtualization, API forwarding and time-sliced GPU represent two purely software-based virtualization mechanisms. **API Forwarding** techniques [8, 14, 17, 24, 29, 30, 45, 46, 59, 75, 76, 80] can be implemented at different layers, such as rCUDA [19] and vCUDA [44] within the CUDA layer, and cGPU [13] and vGPU [54] at the GPU driver layer. Users in VMs only invoke function stubs [4, 5, 43, 64], which are relayed by an API forwarding server to actual implementations on the host side. Since API forwarding does not require hardware support, it relies on the API server [18, 40] to isolate computing units and memory within the GPU. **Time-sliced GPU** method utilizes time-sharing mechanisms to provide multiple virtual GPUs [11, 48, 72, 80]. Similar to API forwarding, it depends on a software module (scheduler), to isolate the time slices allocated

Table 1: Comparison between different virtualization mechanisms of AI accelerators: Full or Partial Virtualization refers to whether users are aware of operating in a virtual environment. Software Threat Model specifies which software component is responsible for isolation between different tenants. Metric encompasses three dimensions of resource virtualization. Number of Virtual Accelerators indicates whether there is a constraint on the number of virtual accelerators.

Accelerator	Method	Full or Para Virtualization	Software Threat Model	Metric			Number of Virtual Acc.
				Instruction	Memory	Interconnection	
GPU	API Forwarding [75, 80]	Para-virtualization	API server	Yes	Yes	No	Unlimited
	MPS [53]	Para-virtualization	MPS server	Yes	Yes	No	Unlimited
	MIG [52]	Full-virtualization	Hypervisor	Yes	Yes	No	Limited, 7 in A100
	Time-sliced GPU [11, 72]	Full-virtualization	Scheduler	No	No	No	Unlimited
NPU	Aurora [41]	Para-virtualization	Runtime	Yes	Yes	No	Unlimited
	V10 [77, 78]	Para-virtualization	Hypervisor	Yes	Yes	No	Unlimited
	vNPU	Full-virtualization	Hypervisor	Yes	Yes	Yes	Unlimited

to different VMs. What's more, these software-based GPU virtualization mechanisms also face performance degradation due to software intervention. Multi-process service (MPS) and multi-instance GPU (MIG) are hardware-based GPU virtualization mechanisms. **MPS** [53] operates on a client-server architecture, where all tasks configured in the MPS mode dynamically send kernels to the MPS server. The MPS server utilizes CUDA streams to execute multiple tasks concurrently and can limit the percentage of GPU threads/SM and memory allocated to different tasks. However, the MPS operates in user space and is not compatible with the VM-based virtualization. **MIG** [52] is the latest GPU virtualization mechanism, which partitions a whole GPU into up to seven virtual GPUs, each of which can be passed through to a VM. However, as MIG offers the strongest security model, it sacrifices flexibility, supporting only seven virtual GPUs with several fixed configurations.

For NPU virtualization, recent academic research has explored para-virtualization solutions designed to monolithic NPUs. For instance, the Aurora [41] utilizes a user-space runtime to manage the mapping from virtual to physical NPU cores, thereby enabling swift migration between NPU tasks. Additionally, Xue et al. [77, 78] have proposed a conceptual design for NPU virtualization with fine-grained resource sharing and isolation. However, these studies overlook the interconnections and topology between multiple NPU cores, which is essential for inter-core connected NPUs to optimize data flow within ML workloads.

3 DESIGN OVERVIEW

3.1 Programming Models

```
Tensor v1 = graph.addVariable(FLOAT, {4}, "v1");
Tensor v2 = graph.addVariable(FLOAT, {4}, "v2");
...
Sequence prog;
prog.add(Copy(c1, v1));
...
// Create a compute set and add its execution to the program
ComputeSet computeSet = graph.addComputeSet("computeSet");
for (unsigned i = 0; i < numTiles; ++i) {
  VertexRef vtx = graph.addVertex(computeSet, "SumVertex");
  graph.connect(vtx["in"], v1.slice(i, 4));
  graph.connect(vtx["out"], v2[i]);
  graph.setTileMapping(vtx, i);
  graph.setPerfEstimate(vtx, 20);
}

prog.add(Execute(computeSet));
```

Listing 1: Official code example of IPU programs [28].

We first illustrate the programming model of the inter-core connected NPU, using the IPU as an example. Listing 1 provides a basic demonstration of an IPU task. In contrast to the programming model for traditional GPU tasks, which permits offloading GPU kernel to any isomorphic hardware threads, IPU requires developers to explicitly designate a specific NPU core for each tensor using setTileMapping(tensor, coreID). The program manipulates these tensors with a set of highly parallel tasks (compute set) executed on designated NPU cores, and fully utilizes the data flow characteristic inherent in ML tasks with inter-core communication routines. For instance, developers can provide a computational graph of ML tasks, which must be mapped onto the hardware IPU cores while considering the underlying topology. Furthermore, the copy primitive in the IPU software framework can leverage the on-chip network (interconnection), which facilitates direct data transfer between the source and destination cores. Further details regarding the programming model for the IPU can be found on its official website [28].

3.2 Overview for vNPU Design

We introduce vNPU, a comprehensive virtualization architecture designed for inter-core connected NPUs, such as IPU [27], tenstorrent [69], Groq [1], etc. These NPUs utilize the multi-core system, scratchpad-centric memory and data flow architecture to accelerate ML tasks. vNPU consists of two major hardware extensions: vRouter (for instruction and NoC) and vChunk. Additionally, it includes an enhanced hypervisor to manage all meta-tables and resources of virtual NPUs using various mapping and allocation strategies (see in §4.3).

- **vRouter:** Unlike the symmetrical cores in CPUs and GPUs, the inter-core connected NPU defines a hardware topology for multiple cores. During computation, data can be transferred directly among NPU cores, thereby reducing extra memory accesses. vNPU introduces a new module called vRouter, to virtualize the instruction flow between the CPU core and NPU cores (i.e., dispatching NPU instructions, see in §4.1.1), as well as the data flow among NPU cores (i.e., NoC, see in §4.1.2).
- **vChunk:** Unlike the traditional memory hierarchy (cache with global memory) employed by CPUs and GPUs, the inter-core connected NPU utilizes SRAM-centric (e.g., 900MB) and HBM/DRAM-backend memory architecture, without complex hardware mechanisms such as cache coherence and association. Before executing an NPU task, weights are loaded from the HBM

into the local SRAM in chunk granularity. vNPU employs a specialized memory virtualization mechanism called vChunk (see in §4.2), which fully leverages NPU memory access patterns.

4 DETAILED DESIGN

4.1 vRouter: Virtualization of NPU Instruction and Interconnection

As introduced in §3.1, each NPU tensor is associated with a specific NPU core ID. Therefore, an inter-core connected NPU must dispatch instructions and data to the appropriate cores, which relies on two additional hardware routers: the instruction router and the network-on-chip (NoC) router.

NPU instruction router: Inter-core connected NPUs comprise multiple cores at distinct topological locations. To achieve fine-grained access control for each core, developers utilize a dedicated NPU core ID combined with each NPU instruction. Once an NPU instruction is offloaded to the NPU device, a router in the NPU controller dispatches the instruction to the designated NPU core.

NoC router: Current ML tasks are structured as computing graphs, allowing NPUs to effectively leverage predefined data flows to accelerate these tasks. An NPU can be divided into multiple pipeline stages, with each stage dedicated to processing specific layers of ML models. To minimize unnecessary memory loads and stores for intermediate results, NPUs employ Network-on-Chip (NoC) among NPU cores. Each NoC node contains a hardware router to transmit NoC packets to the next step.

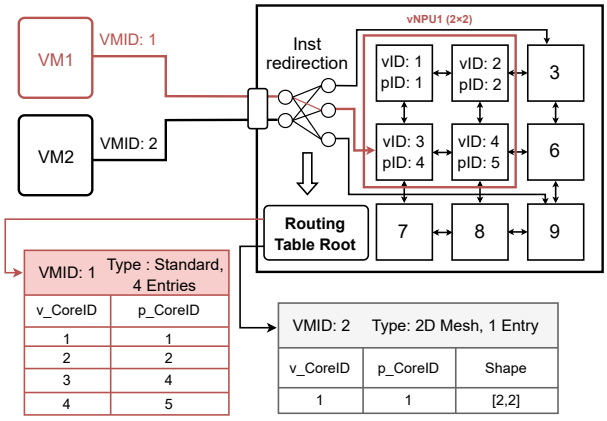


Figure 4: Virtualization support for NPU instruction router: vNPU redirects the NPU instruction from the virtual NPU core to the physical NPU core.

4.1.1 Virtualization Support for NPU Instruction Router. Different from CPU virtualization, which requires a new mode (e.g., non-root) for virtual CPUs (temporal sharing). The vRouter in the NPU controller focuses on redirecting NPU instructions from the virtual NPU core to the physical NPU core (spatial sharing). Therefore, NPU instruction virtualization must ensure that: (1) Developers have the capability to access virtual NPU cores without requiring awareness of the underlying physical NPU cores; and (2) there is strong isolation between NPU cores across different VMs.

vNPU introduces the Routing Table (RT), a crucial data structure utilized by the vRouter. Similar to the page table (e.g., EPT) [9, 37] used in memory virtualization, which translates virtual addresses to physical addresses, the routing table maps virtual NPU core IDs to physical NPU core IDs. Figure 4 illustrates the various organizations of routing tables. Each routing table is associated with a specific VM using the VMID, as well as its type and size. A standard routing table records each mapping between the virtual NPU core ID and its corresponding physical NPU core ID. However, this approach may lead to inefficient use of on-chip memory by reserving a large number of entries for the regular NPU topology. To address this inefficiency, we propose a more specific routing table structure, which adopts a regular shape (e.g, 2D mesh) for NPU topology. This optimized structure only records the initial ID of the virtual and physical NPU core, and the shape of the virtual NPU topology.

When receiving NPU instructions, the vRouter inside NPU controller translate the virtual core ID to the physical core ID according to the routing table, indexed by the VMID and the v_CoreID. To alleviate potential bottlenecks in the routing table lookup, the NPU controller stores all routing tables in SRAM.

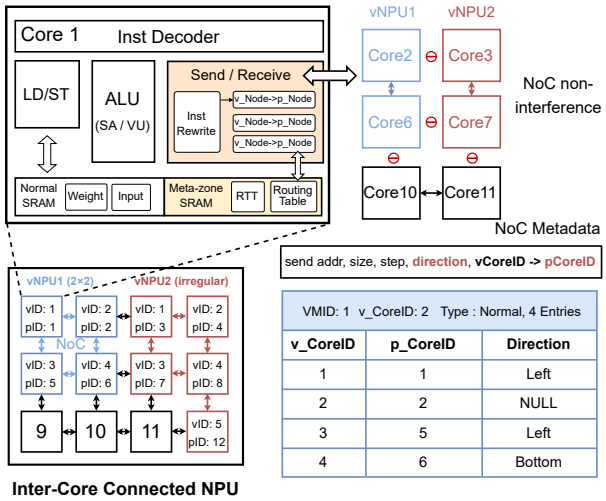


Figure 5: NPU NoC virtualization: vNPU virtualizes the interconnection between multiple NPU cores using NoC vRouter.

4.1.2 Virtualization Support for NoC router. In addition to virtualizing the NPU instruction router, vNPU also virtualizes the NoC router, providing a virtual topology for the virtual NPU. NoC's vRouter also relies on the routing table but requires additional information: direction, particularly for irregular NPU topologies. For a regular NPU topology, the dimension-order routing (DOR) algorithm can be employed to effectively prevent deadlocks in the NoC network. For example, in a 2D mesh topology, packets are routed first along the X-axis and subsequently along the Y-axis, ensuring a deadlock-free communication pattern. However, when dealing with a virtual NPU with an irregular topology, using the default direction may result in NoC packets being transferred to an NPU node belonging to another virtual NPU. We term this phenomenon as **NoC interference**. Take vNPU2 in Figure 5 as an example: If virtual NPU core 5 in vNPU2 wants to send packets to virtual NPU core 3, the default path would first transfer the packets to physical NPU core 11, and then to physical NPU core 7. However, this routing path passes through physical NPU core

11, which does not belong to vNPU2, causing NoC performance interference between different virtual NPUs. Therefore, we provide two routing strategies for NoC virtualization: (1) employing the default DOR strategy, which may lead to potential performance interference among virtual NPUs; and (2) predefining the routing direction inside the routing table to ensure that NoC packets remain confined within the virtual topology.

Figure 5 illustrates the overall design for NoC virtualization. We extend the original send/receive engine in each NPU core to rewrite the destination NPU core ID to the actual physical NPU core ID. If additional direction information is provided on the relay node, the vRouter will route the NoC packet based on this given direction. Since different virtual NPU cores may have varying routing information, the routing table needs to be stored in the local memory (Meta-zone, see in §5.1) of each NPU core.

4.2 vChunk: NPU Memory Virtualization

Although modern inter-core connected NPUs are equipped with substantial on-chip SRAM, they may also address scenarios where the model size exceeds the capacity of the SRAM and must be stored in the global HBM or DRAM. Therefore, vNPU also supports global virtual memory for NPU cores. Current memory virtualization for GPUs and CPUs primarily targets the classical memory hierarchy (cache and global memory), with fine-grained load/store instructions. In contrast, NPUs employ a distinct memory architecture characterized by *high-bandwidth, SRAM-centric local memory alongside high-capacity backend memory (HBM/DRAM)*, relying on coarse-grained memory access via DMA operations. Therefore, the traditional page-based memory virtualization may be not suitable for NPU memory virtualization due to its extremely high memory bandwidth with the burst phenomenon [36]. More specifically, to load model weights from the HBM/DRAM into the SRAM, each NPU core continuously initiates DMA operations every few cycles. Any TLB misses can cause a stall in numerous subsequent DMA requests, significantly reducing the available memory bandwidth [36, 41].

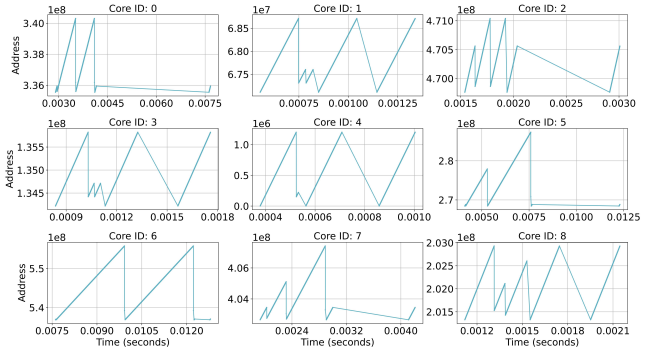


Figure 6: The trace of accessed global memory addresses for the ResNet workload across different NPU cores.

Specialized memory access patterns for NPUs: Fortunately, the distinct memory hierarchy of NPUs with dedicated NPU workloads give rise to specialized NPU memory access patterns, which provides a new opportunity for NPU memory virtualization. As for a classical NPU workflow, each NPU core first loads model weights from the global memory (HBM) into its local memory (SRAM). After the computation, activations or results are transferred directly via inter-core connections to the next layer, without additional memory access to the global memory. Based on the NPU workflow, we identify three distinct memory access patterns that arise during data transfers between SRAM and HBM/DRAM. First, model weight transfers between SRAM and HBM/DRAM predominantly occur at tensor granularity (**Pattern-1**). Second, within a single iteration of an NPU task, the memory addresses of the weight tensor accessed by each NPU core typically demonstrate a monotonic increase (**Pattern-2**). Third, given the iterative and looping nature of ML tasks, the tensor addresses required by each NPU core reset at the beginning of each iteration and repeatedly access the same set of addresses throughout iterations (**Pattern-3**). Figure 6 has illustrated these patterns by tracing accessed memory addresses of ResNet model across multiple NPU cores and iterations. In these traces, the accessed addresses demonstrate a monotonic increase within a single iteration, and exhibit repeated access to the same addresses across different iterations. Moreover, the activation can be transferred directly by inter-core communication, thus, no additional global memory access for intermediate results during the execution.

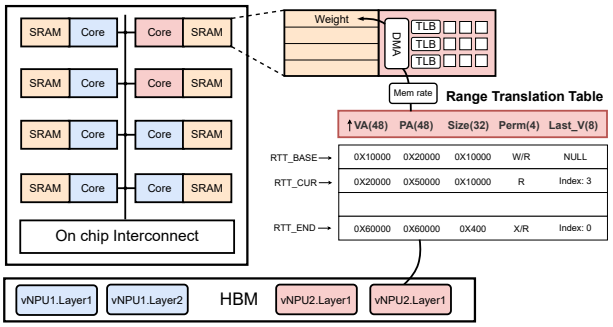


Figure 7: vNPU organizes virtual memory for each NPU core at chunk granularity, and implements an efficient indexing mechanism for the range translation table.

Therefore, vNPU introduces a customized memory virtualization mechanism for NPUs called vChunk, which fully leverages the NPU's memory access patterns mentioned above. vChunk replaces the *fixed-page-size translation* with the *range-based translation* (**Pattern-1**), using a novel structure called the Range Translation Table (RTT). Figure 7 illustrates the overall design of the RTT. Each entry in the table contains a virtual address (48 bits), a physical address (48 bits), the range size (32 bits) and associated permissions. Although some prior work [10, 22] has also proposed ranged translation, it encounters challenges in indexing entries in the range table with dynamic range sizes. However, the NPU workflow exhibits distinct memory access patterns, presenting an opportunity to optimize the indexing procedure for the range table. First, during the one iteration, the model addresses required by an NPU core typically increase monotonically (**Pattern-2**), the next RTT entry is often the one needed. As a result, RTT entries are organized in ascending order of virtual addresses, and each NPU core maintains the index of the currently used entry (*RTT_CUR*). Second, considering the iterative/looping nature of NPU workloads, the model addresses required by each NPU core remain consistent

across different iterations (**Pattern-3**). To exploit this characteristic, an additional field, *last_v*, is introduced into the RTT entry. This field records the index of the next entry accessed in the previous iteration. By leveraging this field, the accessed memory address can jump back to the initial state when transitioning to the next iteration. During a range TLB miss, the NPU core first checks for the *last_v* value to load the corresponding range table entry into the range TLB. If *last_v* is either not recorded or incorrect, the NPU core retrieves subsequent range table entries (return to *RTT_BASE* when reaching *RTT_END*) until it retrieves the required entry. Finally, it updates the *last_v* field with the entry's index and adjusts the *RTT_CUR* value accordingly.

In addition to memory virtualization, vChunk implements an Access Counter to locally track its memory access counts during the monitored time window, which is similar to the prior work [41]. The NPU controller can set the maximum memory bandwidth for different virtual NPUs according to user's requirements. Without these memory rate restrictions, virtual NPUs may experience performance degradation due to memory interference and contention.

4.3 Topology Mapping Strategies for Virtual NPUs

In addition to the hardware extensions required for NPU virtualization, the topology of the NPU introduces the challenge of core allocation during the initialization phase. A naive NPU core allocation strategy can lead to suboptimal resource utilization, a phenomenon we called **Topology Lock-in**. For instance, consider a scenario where a user requests two 3×3 2D mesh virtual NPUs from a physical NPU chip with a 5×5 2D mesh hardware topology. In such cases, the hypervisor might only be able to allocate a single virtual NPU, despite there being sufficient cores for both requests. This limitation occurs because the hypervisor cannot configure two virtual NPUs with the same topology that match the user's requirement. As a result, the topology lock-in problem leads to a wastage of approximately 64% of the NPU cores (16 out of 25 in this example). Instead of allowing such inefficiency, *can we explore strategies to utilize these idle NPU cores on a best-effort basis?*

To enhance NPU utilization, we propose a *topology-mapping* strategy for NPU core allocation. This approach relaxes the constraints of topology lock-in for virtual NPUs, thereby maximizing resource utilization. However, the topology-mapping strategy must satisfy three requirements: First, the number of nodes in the mapped topology must be identical to that of the original topology (**R-1**). Second, the mapped topology should closely resemble the original topology (**R-2**). Third, to ensure non-interference between virtual NPUs, the topology must remain connected (**R-3**).

Figure 8 illustrates NPU core allocation using different hardware topology mapping strategies. A straightforward approach involves allocating NPU cores based on their core IDs. However, it potentially leads to suboptimal performance due to increased inter-core connection overhead. To address this, vNPU uses a minimum topology edit distance algorithm to identify a topology for the virtual NPU that is similar to the original topology. The topology (or graph) edit distance [31, 51, 60, 61] quantifies the dissimilarity between two topologies by determining the minimum number of edit operations required to transform one topology into the other. These edit

Figure 8: Different topology mapping strategies for vNPU core allocation. Assume the NPU consists of 25 cores arranged in a 5×5 2D mesh topology. A user wants to allocate 2 vNPUs, each with a 3×3 2D mesh topology.

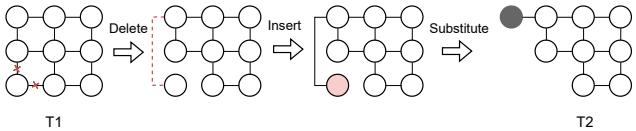


Figure 9: The topology edit distance between T1 and T2 is four, involving two edge deletions, one edge insertion, and one node substitution.

operations include the insertion, deletion, and substitution of nodes and edges. Figure 9 illustrates an example: transforming T1 into T2 requires four edit operations, specifically two edge deletions, one edge insertion, and one node substitution. Therefore, the topology edit distance between T1 and T2 is 4.

The pseudo-code implementation of the algorithm is shown in Algorithm 1. Here, T represents the entire NPU topology in hardware, T' refers to the topology of NPU cores that are already allocated to other virtual NPUs, and T_req represents the topology required by the current virtual NPU. We first select all candidate NPU topologies with required NPU cores (**R-1**). Given that the problem of determining the minimum topology edit distance is NP-hard, it is essential to prune candidate topologies to minimize unnecessary computations. We employ three intuitive pruning strategies: First, if a virtual NPU requires a guarantee of non-interference, the selected topology must be connected (Line 25), according to the **R-3** mentioned above. Second, for the same topology, we retain only one instance to reduce redundant calculations (Line 25). Third, if a candidate topology matches the required topology exactly, we directly return this topology with the corresponding NPU cores (Line 22). After selecting all candidate topologies, the topology edit distance is calculated between each candidate topology and the required topology (Line 13), and returns the similar topology with the minimum topology edit distance (**R-2**).

Heterogeneous topology mapping: Furthermore, the topology mapping algorithm also supports unbalanced networks and heterogeneous nodes within the topology. For instance, when dealing with non-uniform traffic patterns such as all-reduce, certain edges become more critical than others. In these cases, we can define a customized edge-match function (Line 8) that imposes a greater penalty if the critical path is absent in the target topology. More specifically, any delete or substitute operations on these critical paths will increase the edit distance. Similarly, when accounting for

the heterogeneous nodes in the NPU topology, additional penalties can be incorporated into the customized node-matching function (Line 3). For instance, if a node in the required topology is positioned close to the memory interface but is far from it in the actual mapped topology, users can define a customized node-matching function to assign additional penalty to these nodes. In our experience, this penalty value is determined by the difference in distances to the memory interface.

Topology fragmentation: Even with the use of virtual topology and the similar topology mapping algorithm, the fragmentation problem persists during NPU core allocation. Since users' allocation requests cannot be anticipated in advance, we are unable to transform this allocation problem into a combinational problem. If we relax the constraints to permit a disconnected virtual topology of the NPUs, fragmented NPU cores can also be utilized by virtual NPUs to improve the resource utilization. However, this approach may introduce additional overhead associated with the conflict in inter-core communication, resulting in a trade-off between performance and resource utilization.

5 IMPLEMENTATION

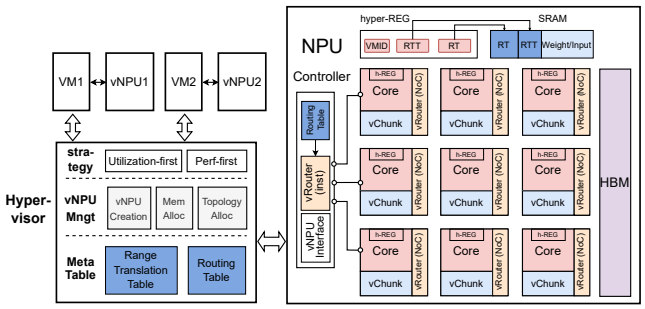


Figure 10: The overall architecture for vNPU: vNPU enhances the vRouter and vChunk modules for NPU hardware and incorporates a modified hypervisor to manage all resources of the virtual NPU.

Algorithm 1 Topology Mapping Algorithm

```
1:  Procedure NodeMatch(N1, N2)
2:  if N1[abbr] ≠ N2[abbr] then
3:      return NodeCost {A penalty is applied if the attributes of
        the two nodes differ.}
4:  end if
5:
6:  Procedure EdgeMatch(E1, E2)
7:  if E1 exists but E2 is none then
8:      return E1.cost {Different edges are assigned varying penalty
        values based on their importance.}
9:  end if
10:
11: Procedure computeTed(T_req, candidate)
12: subGraph ← T.subgraph(candidate)
13: TED ← topo_edit_distance(T_req, subGraph, NodeMatch,
    EdgeMatch)
14: return TED
15:
16: Function minTopologyEditDistance(T, T', T_req)
17: minCost ← ∞
18: remainN ← T.nodes() − T'.nodes()
19: candidates = [], topos = []
20: totalSubTopo ← comb(remainN, T_req.nodeNum)
21: for each nodes in totalSubTopo do
22:     if nodes.topo is equal to T_req then
23:         return nodes
24:     end if
25:     if connected(T, nodes) and nodes.topo is not in topos then
26:         candidates.add(nodes)
27:         topos.add(nodes.topo)
28:     end if
29: end for
30: pool ← multiprocess(CPU.count) {Parallel}
31: results ← pool.map(ComputeTED, T_req, candidates)
32: minTED ← min(results)
33: return minTED.nodes
```

5.1 Hardware Architecture for vNPU

The right portion of Figure 10 illustrates the hardware architecture of vNPU. vNPU first introduces a new feature: hyper mode for the NPU controller. Only the hyper-mode NPU controller is permitted to modify virtualization-related tables, such as the routing table and range translation table, and configures hyper registers (e.g., hyper-REG) for each NPU core. Furthermore, the hyper-mode NPU controller can collaborate with the CPU-side hypervisor. For example, only the hypervisor is authorized to map MMIO space of hyper-mode NPU controller (e.g., PF); whereas guest VMs are restricted to mapping the MMIO spaces only associated with virtual NPUs (e.g., VF).

In addition to a new mode for the NPU controller, vNPU incorporates two additional hardware modules: vRouter and vChunk to achieve full virtualization. vNPU integrates the vRouter into the centralized NPU controller and the NoC engine for each NPU core. For NPU instruction virtualization, the vRouter within the NPU controller redirects instructions from the virtual NPU cores to the physical NPU cores based on the various types of routing tables. For NoC virtualization, the vRouter in each NPU core translates the destination core ID in the NoC instructions to the corresponding physical core ID, while further ensuring that NoC packets remain within the boundaries of the virtual topology. The vChunk is implemented within the DMA engine to manage data movement between the on-chip local SRAM and the global HBM. Moreover, since both the vRouter and vChunk depend on newly introduced meta tables: routing table and range translation table, which cannot be modified by the virtual NPU itself. vNPU partitions the on-chip SRAM into two distinct regions: the meta-zone and the weight-zone. The meta-zone is designated for storing all meta tables and can only be configured by the hyper-mode NPU controller. During execution, the NPU core is restricted from modifying the context within the meta-zone. To access mapping entries in these meta tables, the NPU core incorporates hyper-mode registers (set by the

hyper-mode NPU controller), such as the base address of the routing table, etc. As for the weight-zone, it stores the model weight and intermediate result for the NPU execution.

5.2 Hypervisor Extension for vNPU

vNPU requires more complicated hardware configurations for virtual NPUs, such as topology and memory. Therefore, the hypervisor needs to manage the internal hardware resources of each virtual NPU in a fine granularity. Specifically, it must organize additional meta tables, including the range translation table and routing table, and allocate various combinations of hardware resources to each virtual NPU in accordance with user requirements.

First, to effectively manage the NPU's global memory (e.g., HBM) for various virtual NPUs, the hypervisor utilizes the traditional buddy system for memory allocation, and records address mappings in the range translation table. Unlike the page table which needs to partition blocks from the buddy system into fixed-size pages, vNPU maps an entire block directly into the RTT entry with the block size. To optimize the lookup process, the hypervisor sorts RTT entries by the virtual address, and allocates all virtual NPU's memory during the initialization phase. As for the virtual topology, the hypervisor adopts the similar topology mapping strategy (or exact mapping) to allocate the NPU cores for virtual NPUs. After the allocation, the hypervisor stores the mapping between virtual core IDs and physical core IDs in the routing table. Furthermore, the hypervisor can also define the routing direction for each node when NoC performance isolation is required. Both the range translation table and the routing table are managed by the CPU-side hypervisor, and finally need to be deployed on the NPU side by the hyper-mode NPU controller.

The hypervisor maintains the abstraction of the virtual NPU, which consists of virtual NPU cores, topology, and memory. During the creation of a VM, the user must specify the requirements for the virtual NPU, including the number of NPU cores, the required topology, and the size of NPU memory. The hypervisor then allocates these NPU resources and configures the meta table for NPU virtualization. NPU resources are spatially shared among different virtual NPUs. We discuss the different sharing models of virtual NPUs in §7.

6 EVALUATION

6.1 Experimental Setup

We implement a hardware prototype of vNPU on top of Chipyard [3], which is a customized SoC generator designed for evaluating full-system hardware. As for software components, we opt Linux-6.2 as the host kernel and Linux-6.10 as guest kernel, and modify the default KVM module in Linux kernel to manage all virtual NPU resources. The microarchitecture of the NPU design refers to the IPU [27], an inter-core connected AI accelerator. We notice that some prior work like Aurora [41] and Xue et al. [77] propose the para-virtualization for NPU based on the unified virtual memory (UVM), but does not support for the data flow architecture of inter-core connected NPUs. We have incorporated all virtualization extensions (vChunk and vRouter) of vNPU in our prototype, which can concurrently support both UVM and data flow configurations. We evaluate the vNPU performance by running micro-tests and small ML workloads using FireSim [39], a cycle-exact, FPGA-accelerated RTL simulator, and simulating the performance of large ML workloads with DCRA [50, 56], a distributed chiplet simulator. The configuration is shown in Table 2.

Comparative Systems:

- **MIG-based virtual NPU** Similar to the MIG in GPU virtualization, the MIG NPU offers several fixed partitions for the entire NPU chip, with each partition having a predetermined sub-topology among the NPU cores. Across different partitions, the MIG NPU ensures strong isolation for the NPU cores, memory, and interconnections.
- **UVM-based virtual NPU:** Some prior works [41, 77] have introduced NPU virtualization mechanisms that incorporate unified virtual memory but lack interconnection support. Despite the architectural differences of these NPUs, we still compare vNPU with these works to highlight the architectural advantages of inter-core connection design for NPUs, as well as to underscore the significance of topology virtualization.
- **vNPU:** Our solution mainly focuses on topology virtualization for inter-core connected NPUs [1, 27, 38], which is absent in the prior work. vNPU features two additional hardware modules: vChunk and vRouter for NPU resource virtualization, and extended hypervisor for NPU's virtual resource management.

Table 2: SoC configurations used in the evaluation

Parameter	FPGA	SIM
Systolic array dimension (per tile)	16	128
# of accelerator tiles	8	36
Scratchpad size (per tile)	512KB	30MB
Scratchpad size (total)	4MB	1080MB
NoC topology	2D Mesh	2D Mesh
Shared L2 size (UVM)	2MB	/
Shared L2 banks (UVM)	8	/
DRAM / HBM bandwidth	16GB/s	360GB/s
Tops (per tile)	0.5	16
Tops (total)	4	576
Frequency	1GHz	500MHz

6.2 Micro-test for vNPU Hardware Extensions

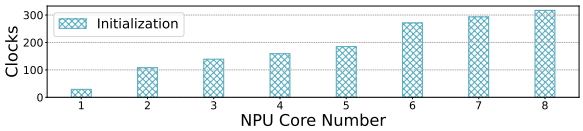


Figure 11: Configuration overhead of the routing table with different numbers of NPU cores.

6.2.1 Virtualization for Instruction Dispatch. To redirect the NPU instruction from the virtual NPU core to the physical NPU core, we have implemented a vRouter within the NPU controller. We first evaluate the configuration latency for the routing table via the NPU controller, as shown in Figure 11. The x-axis represents the number of NPU cores, while the y-axis (clocks) includes two overheads: querying for core availability and configuring the routing table. The total time of routing table setup is merely a few hundred cycles, which can be neglected during the virtual NPU creation.

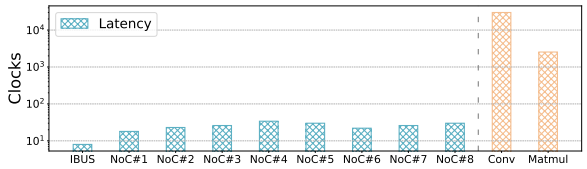


Figure 12: Latency of NPU instruction dispatch via vRouter.

Next, we consider the overhead of instruction dispatch during the NPU execution. Instructions are redirected to the corresponding NPU cores via the routing table. If consecutive instructions are directed to the same NPU core, the subsequent instructions do not need to query the routing table again. There are two implementations for instruction routing: one using the instruction BUS and the other using a separate NOC to dispatch the instruction to the corresponding NPU core. The experimental results are shown in the left side of Figure 12. NoC#1~8 indicates the delay for NPU controller dispatching instructions to core 1~8 by the separate instruction NoC. Due to varying distances between different NPU cores in the NoC, there are slight differences in latency. Although IBUS has the shortest and fixed latency, its transmission structure lacks scalability in multi-core systems. On the right side of the Figure 12, we illustrate the execution times of two common NPU instructions: convolution and matrix multiplication. The execution times of these NPU instructions are two to three orders of magnitude longer than the instruction routing latency. Therefore, the cost of instruction routing latency on overall execution is negligible.

Table 3: Micro-test of the vRouter: Compare the NoC virtualization with vRouter and the standard NoC instructions.

Number of routing packets for data transmission	Non-Virtualization		Virtualization	
	Send (clk)	Receive (clk)	vSend (clk)	vReceive (clk)
2	309	311	342	372
10	1430	1432	1432	1492
20	2810	2818	2822	2894
30	4236	4240	4240	4308

6.2.2 Virtualization for inter-core connection. Besides the virtualization for instruction dispatch, vRouter is also adopted by the NoC engine for the virtualization of inter-core connection. we first evaluate whether the vRouter will affect the performance of data transfers, as shown in the Table 3. *Send*/*Receive* represents the data transfer between NPU cores under the non-virtualization. While, *vSend* /*vReceive* means the data transfer under the vRouter mechanism. A routing packet refers to a data block that can be transmitted by a routing instruction. In our experiment, the size of a routing packet is 2048 bytes. Experimental result indicates that the vRouter mechanism only introduces a 1%~2% overhead for inter-core data transfers.

6.2.3 Different data broadcast methods for virtual NPUs: vRouter vs. memory synchronization. Virtual NPUs can employ different data broadcast methods, including the vRouter mechanism proposed in this paper and synchronization through global memory. Many prior studies [27, 47] have highlighted the critical importance of inter-core connection for large-scale NPUs. In our experiments, we find that this observation also holds in the NPU virtualization scenario. To evaluate the performance of these approaches, we

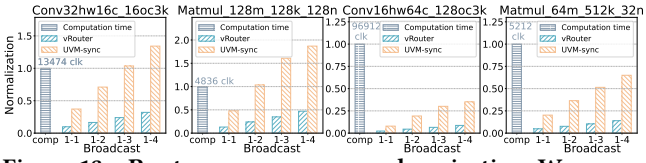


Figure 13: vRouter v.s. memory synchronization: We assess the data broadcast latency of different NPU kernels by comparing the vRouter with memory synchronization.

measure the data broadcast cost across different NPU kernels with varying ratios of senders to receivers, where each sender or receiver occupies a single NPU core. In Figures 13, the label 'comp' on the X-axis indicates the execution time of the NPU kernel, while '1:n' specifies that the kernel's execution results are broadcast to 'n' nodes. Our evaluation reveals that the vRouter mechanism achieves an average improvement of 4.24x compared to the global memory synchronization, and similar experimental results also appear in the reduce operation. This enhancement is facilitated by vRouter's ability to exploit inter-core bandwidth and using the NoC handshake protocol for synchronization. Furthermore, with the vRouter mechanism, the data broadcast cost is significantly lower than the NPU kernel execution time, allowing the broadcast overhead to be fully overlapped with kernel execution. Conversely, when using memory synchronization, the broadcast cost for the 'Matmul' kernel under the '1:4' configuration exceeds the kernel execution time. This imbalance can degrade the end-to-end performance of real-world NPU workloads, as discussed in §6.3.1.

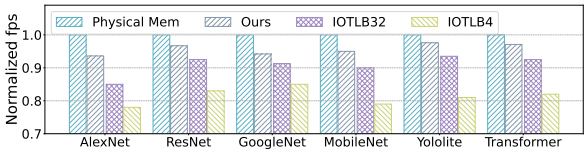


Figure 14: The normalized performance of ML workloads with different memory virtualization methods.

6.2.4 Virtualization for NPU memory. NPU suffers from a memory burst phenomenon when loading model weights from HBM/DRAM to SRAM. In this process, the DMA engine within the NPU core concurrently initiates data read requests to maximize the overall memory bandwidth, resulting a heavy translation load: triggering the translation request for every few cycles.

We evaluate the translation overhead of various ML models with different configurations: physical memory, page-based and range-based translation, as shown in Figure 14. Compared to the ideal performance without any translation overhead (physical memory), the page-based translation incurs an average of 20% overhead with only 4 TLB entries across six different models [15, 33–35, 42, 66]. This overhead can be partially mitigated by increasing the number of TLB entries within each NPU core; however, even with 32 TLB entries, the overhead remains above 9.2%. vChunk employs range-level translation, and further reduce the overhead associated with range entry lookups. Using only 4 hardware range-tlb entries (144 bits for each), the overall overhead for vChunk is minor, remaining below 4.3% in average.

6.3 Different Virtualization Methods for Real-world ML Applications

We compare vNPU to different NPU virtualization methods: UVM and MIG. Some previous studies [41, 77] have proposed virtual NPU abstractions, which are based on a unified virtual memory. Although it may be somewhat unfair to directly compare the performance of vNPU with these solutions, our aim is to highlight the significance of virtual topology in the context of large-scale virtual NPUs. We also observe that some commercial NPUs, such as the TPU-v6e, have implemented MIG-based NPU virtualization with several fixed partitions [26]. Following a similar approach, we partition the NPU topology into predefined sub-topologies and construct a comparable system, termed MIG-NPU. In this setup, NPU cores within the same virtual NPU can establish inter-core connections, while the strong isolation is maintained between NPU cores located in different sub-topologies. Due to hardware resource constraints on our FPGA platform, we are limited to synthesizing a maximum of 8 NPU cores and 4MB SRAM in our FPGA-based implementation. To assess the performance of ML workloads on a larger NPU chip (e.g., 36/48 NPU cores, 1080/1440MB on-chip SRAM), which is capable of accommodating all model weights within its on-chip SRAM using the tensor partition, such as SOTA data-flow NPUs like Groq [1] and IPU [27]. we modify a chiplet simulator: DCRA [50], with the vNPU extension and performance profiler.

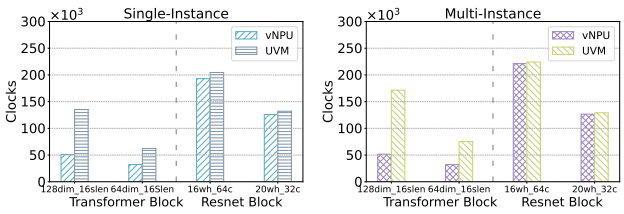


Figure 15: vNPU v.s. UVM-based virtual NPU: We evaluated the performance of different NPU virtualization mechanisms under both single-instance and multi-instance scenarios.

6.3.1 Compared vNPU with the UVM-based virtual NPUs. Figure 15 illustrates the performance comparison between vNPU and UVM-based virtual NPUs. Although the primary performance improvement in vNPU is driven by direct inter-core communication, which may present an inherent advantage over UVM-based NPUs, this evaluation is intended to highlight the benefits of virtual topology routing specifically for data-flow NPUs. In this experiment, each ML workload is executed within a dedicated virtual NPU, with four NPU cores allocated to each instance. We also assess the performance interference between multiple virtual NPUs running on a single NPU chip. Specifically, one virtual NPU runs the ResNet model, while another runs the Transformer model. Our experimental results demonstrate that in a single-instance scenario, vNPU achieves a 2.29x performance improvement for the Transformer model compared to the UVM-based virtual NPU. This improvement is primarily attributed to the virtual topology routing support provided by vNPU. However, for the ResNet model, the benefit of direct data transfer is less pronounced, yielding only a 5.4% improvement in performance. This is due to the varying layer structures in ResNet, which introduce bubbles in the data flow, ultimately leading to performance degradation.

Regarding performance isolation among the multi-instance scenario, UVM-based virtual NPUs exacerbate global memory access contention across different virtual NPUs, leading to an overall performance degradation of approximately 24%. In contrast, vNPU leverages inter-core connections, minimizing reliance on global memory. As a result, the performance interference between multiple tasks is negligible.

6.3.2 Compared vNPU with the MIG-based Virtual NPU. Current commercial NPUs, such as TPUv6-e, have introduced MIG-based virtual NPU instances with fixed topologies. MIG-based virtual NPUs can also leverage inter-core connections, making them a more equitable baseline when compared with vNPU. However, a significant limitation of MIG-based virtual NPU is its restriction to a limited set of fixed NPU topologies. When user-required NPU topologies do not match the available MIG configurations, it can result in either under-utilization of NPU resources or the assignment of multiple virtual NPU cores to a single physical NPU core, as depicted in the upper half of Figure 16. In contrast, vNPU offers flexible virtual topologies for virtual NPUs, allowing for arbitrary configurations of NPU cores and topologies.

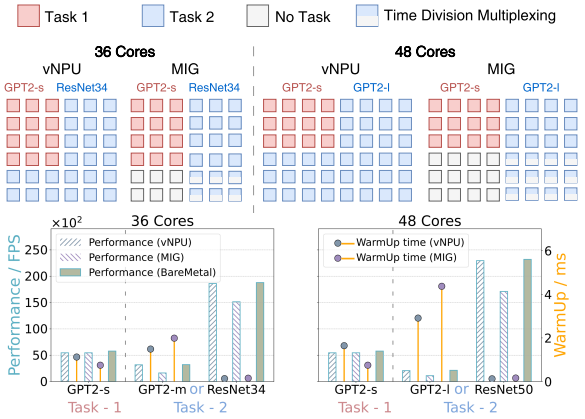


Figure 16: Performance and Warm-up latency of MIG and vNPU: We execute two virtual NPUs on a single NPU chip configured with either 36 or 48 cores. In the upper half of the figure, MIG employs time-division multiplexing when physical cores are less than virtual cores. vNPU leverages a virtual topology that accommodates arbitrary topological shapes. In the lower half, the left Y-axis (bars) represents the performance of various NPU workloads, while the right Y-axis (dots) indicates the warm-up time required before the start of NPU computation.

The lower half of Figure 16 illustrates the task execution performance and warm-up time for both MIG-based virtualization and vNPU. In this experiment, we selected two classical NPU tasks for evaluation: GPT2-small/middle/large and ResNet, and run these two NPU tasks in the two virtual NPUs with the different virtualization methods. In our test, GPT-small consistently runs on virtual NPU 1, while another NPU task operates concurrently on virtual NPU 2. If the number of NPU cores is insufficient to meet user

requirements (only occurs in the MIG solution), we temporarily share a single physical NPU core among multiple virtual NPU cores using time-division multiplexing. In contrast, vNPU consistently allocates a sufficient number of NPU cores with fine-grained allocation for the virtual NPU; Although, its virtual topology may differ from the required topology. Figure 16 presents the evaluation results comparing the performance and warm-up time of MIG and vNPU. The left Y-axis illustrates the concrete NPU performance, while the right Y-axis depicts the warmup time. Results indicate that for the NPU task like GPT-small, which only requires 12 NPU cores, the MIG-based virtual NPU in our configuration has either 18 or 24 NPU cores, wasting up to 50% computing resources (12/24 nodes). In contrast, vNPU can dynamically allocate the exact number (i.e., 12) of NPU cores needed by the virtual NPU, with the better computing resource utilization. For larger NPU workloads, such as GPT-large, which necessitates 36 NPU cores, vNPU is capable of allocating exactly 36 NPU cores with a virtual topology. In contrast, a MIG-based virtual NPU can provision only up to 24 NPU cores. Therefore, the MIG solution must temporally share a single physical NPU core among multiple virtual NPU cores, resulting in up to 1.92x performance degradation compared to vNPU. As for the ResNet model, the performance improvement for vNPU compared with MIG NPU is minor, only 1.28x in average. This is because workload imbalance in the ResNet model can be somewhat mitigated by TDM, for example, by binding a high-load virtual core with a low-load virtual core to share a single physical NPU core. We also discuss the overhead related to the virtual topology in §6.3.5

6.3.3 Compared vNPU with the bare-metal NPUs. In addition to comparing vNPU with other NPU virtualization methods, we evaluate the hardware performance overhead introduced by vNPU itself. vNPU incurs only a minimal performance cost by adding a lightweight translation module that maps virtual core IDs to physical core IDs. To quantify this overhead, we compare vNPU against a bare-metal NPU configuration without any virtualization support while maintaining the same NPU topology. As illustrated in Figure 16, the evaluation demonstrates that the end-to-end performance overhead introduced by vNPU-based NPU virtualization is less than 1% in all NPU workloads.

6.3.4 Warm-up time of virtual NPUs. We also evaluate the warmup time (shown in the right Y axis in Figure 16) of ML workloads under different virtualization methods. The warmup time is primarily spent on loading model weights from global memory into the on-chip SRAM (assume the SRAM capacity is sufficient). Therefore, virtual NPUs with larger memory bandwidth can achieve shorter warmup times. In our implementation, the total memory bandwidth allocated to each virtual NPU is proportional to the number of memory interfaces associated with that virtual NPU. However, this global memory bandwidth has minimal impact on the execution time of the NPU, as interactions between the on-chip SRAM and DRAM are infrequent during NPU execution.

6.3.5 Different Topology Mapping Strategies. vNPU adopts the topology mapping for NPU core allocation. Figure 17 illustrates different topology mapping strategies for a ResNet model. In this scenario, the upper-left and bottom-right NPU cores have already been allocated, so the hypervisor can only assign virtual NPU cores

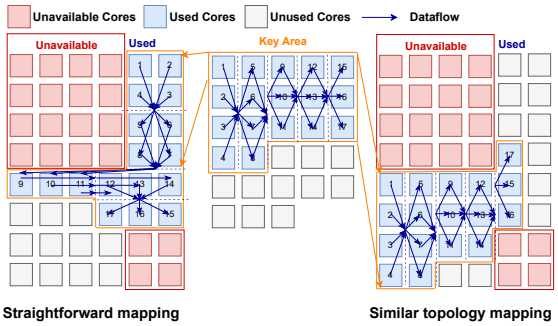


Figure 17: Different topology mapping strategies for the ResNet model.

from the remaining pool. We implement two topology mapping strategies: straightforward mapping and the similar topology mapping. In the straightforward mapping, NPU cores are allocated based on core IDs, with the idle core having the smallest ID being allocated first (or zig-zag). In contrast, the similar topology mapping utilizes the minimum topology edit distance algorithm to allocate an NPU topology that closely aligns with the user's requirement.

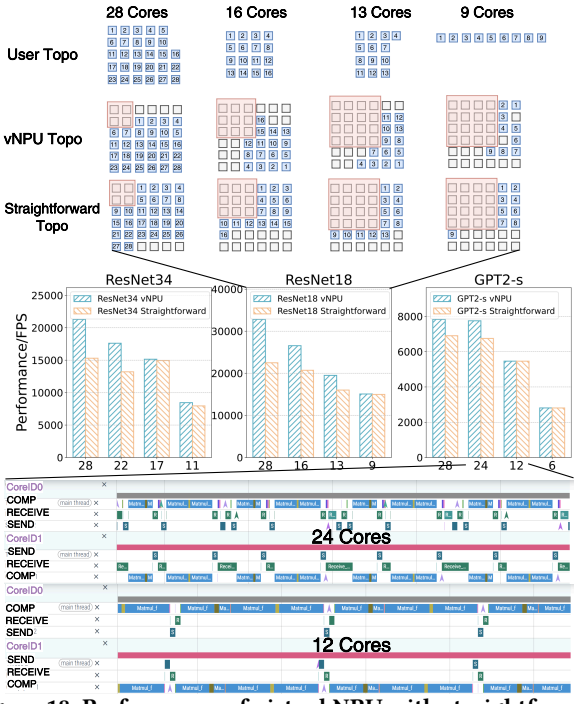


Figure 18: Performance of virtual NPU with straightforward mapping and similar topology mapping: The top part illustrates the concrete topology mappings derived from different strategies for ResNet18, while the middle part depicts the performance outcomes of different ML workloads under these topology mappings, and the bottom part shows the core trace under different number of NPU cores with vNPU topology.

In this test, we assume that the initial state of the NPU is **not empty**, as indicated by the red nodes in Figure 18. We assess the performance of the virtual NPU under different topology mappings:

the vNPU topology and a straightforward topology, and we present one of the detailed node mappings for a ResNet18 model. In the middle part of Figure 18, we present the detailed performance results of various ML workloads under different topology mappings. The X-axis represents the number of NPU cores required by the virtual NPU, while the Y-axis illustrates the FPS achieved by the NPU tasks. The evaluation results demonstrate **three trends**. First, the performance impact of the topology mapping strategy becomes more significant as more NPU cores are allocated. For example, compared with straightforward mapping strategy (zig-zag), the similar topology mapping strategy achieves a 40% improvement for ResNet34 using 28 NPU cores, but only yields a 6% improvement with 11 NPU cores. This is because the proportion of inter-core communication overhead decreases when multiple layers are mapped to a single NPU core. Second, models with more complex graph structures are more sensitive to topology mapping strategies. For instance, ResNet's performance is significantly influenced by the choice of topology mapping strategy, as vNPU's mapping outperforms straightforward mapping with 42% FPS increase in average (ResNet18/34 in 28 Cores). In contrast, GPT models, with their layers having identical structures, are less sensitive to these two mapping strategies. Even a simple zigzag mapping can achieve 89% performance of vNPU's mapping in average. Third, regarding system scalability, while an increased number of NPU cores may introduce additional communication bubbles (bottom part of Figure 18), the similar topology mapping strategy exhibits better scalability at the system level. This is because the similar topology mapping effectively reduces the overhead associated with interconnection contention, provided that the user-defined topology has minimal conflicts. In contrast, a straightforward zigzag mapping, which significantly deviates from the user-defined topology, can lead to substantial conflicts in inter-core communication during NPU execution, ultimately undermining the system's scalability.

6.4 Hardware Cost Analysis

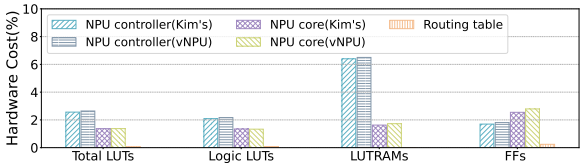


Figure 19: Hardware resource cost: Additional FPGA resource with different NPU virtualization mechanisms.

We synthesize the hardware resource consumption of two NPU virtualization methods on FPGA: vNPU and Kim's solution [41], which adopts the unified virtual memory. Figure 19 illustrates the additional resources required by the vNPU on the NPU controller and cores in terms of Total LUTs, FFs, and etc. Different configurations are labeled as vNPU (vChunk/vRouter) and Kim's (unified virtual memory) in the legend. Our evaluation demonstrates that the vNPU incurs minimal hardware overhead. Both configurations require only an additional 2% of Total LUTs and FFs. For the routing table, a 128-entry configuration requires minimal FF resources for mapping storage, with LUT requirements nearly zero.

7 DISCUSSION

Imbalanced use of VU and SA in the NPU: Prior work has identified imbalanced utilization of the matrix unit (e.g., systolic array, SA) and vector unit (VU) in NPU cores when executing different ML workloads. V10 [77, 78] introduces a tensor operator scheduler that coordinates diverse ML tasks within a single NPU core. However, this approach is not suitable for virtualization scenarios in cloud computing where different tenants do not trust each other. To address this issue, vNPU may adopt hybrid NPU cores, one optimized for matrix operations and the other for vector computations. Tenants can then allocate varying ratios of these two types of NPU cores according to their needs, using a virtual topology.

Temporal sharing v.s. spatial sharing: There are two methods for sharing NPU resources: temporal and spatial sharing. Inter-core connected NPUs typically feature multiple cores with the large on-chip memory. The context for NPU cores contain all model data reserved in the scratchpad, making context switching for NPUs a costly operation. Therefore, vNPU primarily utilizes spatial sharing among multiple NPU cores, without considering the expenses associated with NPU's context switch. However, the vNPU still allows for temporal sharing if cloud vendors wish to engage in over-provisioning.

Address translation for graph workloads like GNN: For graph workloads such as GNNs, which require large graph datasets and involve random information retrieval, our range-translation design may not be ideal. For these types of workloads, employing traditional page-level translation is recommended.

KV-cache support for NPUs: Current commercial NPUs, such as Apple and Qualcomm NPUs, utilize a pre-allocated, fixed-size KV buffer. In our implementation, we adopt this approach as well, specifying a maximum size for the KV buffer in SRAM. The dynamic KV buffer management as well as the KV cache offloading for inter-core connected NPUs will be part of our future work.

8 CONCLUSION

This paper presents the first comprehensive virtualization design for inter-core connected NPUs named vNPU. vNPU focuses on virtualizing specialized hardware structures for NPUs, including multi-core architecture, scratchpad-centric memory, and inter-core connection. To achieve this, it proposes two hardware extensions: vRouter and vChunk, as well as different topology mapping strategies. As AI models continue to evolve, incorporating more powerful NPUs, NPU virtualization will become an essential technology in future cloud computing. The topology virtualization presented in this paper is crucial for data flow accelerators (NPU) but is notably absent in CPU and GPU virtualization.

Acknowledgments

We sincerely thank our anonymous reviewers for their insightful suggestions, and we also appreciate Yaodanjun Ren for the valuable advice to paper refinement. This work is supported in part by STI 2030—Major Projects 2021ZD0200300, China National Natural Science Foundation (No. 62088102,623B2074), and Huawei Central Software Institute. Rong Zhao is the corresponding author.

References

[1] 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

[2] Alibaba. 2024. Qwen/Qwen2-0.5B. https://huggingface.co/Qwen/Qwen2-0.5B. Referenced January 2024.

[3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[4] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. 2014. I/O paravirtualization at the device file boundary. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 319–332.

[5] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. Rio: a system solution for sharing i/o between mobile systems. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 259–272.

[6] AWS. 2024. GNeuronCore-v2 Architecture. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-hardware/neuron-core-v2.html. Referenced January 2024.

[7] AWS. 2024. Use Amazon SageMaker Built-in Algorithms or Pre-trained Models. https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html. Referenced January 2024.

[8] Amnon Barak, Tal Ben-Nun, Ely Levy, and Amnon Shiloh. 2010. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2010 IEEE international conference on cluster computing workshops and posters (CLUSTER WORKSHOPS)*. IEEE, 1–7.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[10] Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. 2023. FlexPointer: Fast Address Translation Based on Range TLB and Tagged Pointers. *ACM Trans. Archit. Code Optim.* 20, 2, Article 30 (March 2023), 24 pages. https://doi.org/10.1145/3579854

[11] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. 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*. 17–32.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC]

[13] Alibaba Cloud. 2024. High GPU Utilization with cGPU. https://www.alibabacloud.com/en/solutions/cgpu?_p_lc=1. Referenced January 2024.

[14] CNCF. 2024. HAMi: Heterogeneous AI Computing Virtualization Middleware. https://github.com/Project-HAMi/HAMi. Referenced November 2024.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[16] Micah Dowty and Jeremy Sugerman. 2008. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Oper. Syst. Rev.* 43 (2008), 73–82. https://api.semanticscholar.org/CorpusID:228328

[17] José Duato, Francisco D Igual, Rafael Mayo, Antonio J Pena, Enrique S Quintana-Ortí, and Federico Silla. 2010. An efficient implementation of GPU virtualization in high performance clusters. In *Euro-Par 2009–Parallel Processing Workshops: HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Delft, The Netherlands, August 25-28, 2009, Revised Selected Papers 15*. Springer, 385–394.

[18] Jose Duato, Antonio J Pena, Federico Silla, Juan C Fernandez, Rafael Mayo, and Enrique S Quintana-Orti. 2011. Enabling CUDA acceleration within virtual machines using rCUDA. In *2011 18th International Conference on High Performance Computing*. IEEE, 1–10.

[19] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing and Simulation*. 224–231. https://doi.org/10.1109/HPCS.2010.5547126

[20] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, and Haibo Chen. 2024. sNPU: Trusted Execution Environments on Integrated NPUs. (2024).

[21] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, and Bob Dreyer. 2023. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[22] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (2016), 118–126. https://doi.org/10.1109/MM.2016.10

[23] 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)*. 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[24] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. 2010. A GPGPU transparent virtualization component for high performance computing clouds. In *Euro-Par 2010-Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31-September 3, 2010, Proceedings, Part I 16*. Springer, 379–391.

[25] Google. 2024. Extract insights from images, documents, and videos. https://cloud.google.com/vision. Referenced January 2024.

[26] Google. 2024. TPU/TPU v6e. https://cloud.google.com/tpu/docs/v6e. Referenced February 2025.

[27] Graphcore. 2024. Intelligence processing unit. https://www.graphcore.ai/products/ipu. Referenced January 2024.

[28] Graphcore. 2024. IPU/ipu-programmers-guide. https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/programming_tools.html. Referenced February 2025.

[29] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2009. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level virtualization for High Performance Computing*. 17–24.

[30] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2011. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*.

[31] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[32] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 770–778. https://api.semanticscholar.org/CorpusID:206594692

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[34] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[35] Rachel Huang, Jonathan Pedoeem, and Cuixian Chen. 2018. YOLO-LITE: a real-time object detection algorithm optimized for non-GPU computers. In *2018 IEEE international conference on big data (big data)*. IEEE, 2503–2510.

[36] Bongjoon Hyun, Youngeun Kwon, Yujeong Choi, John Kim, and Minsoo Rhu. 2019. NeuMMU: Architectural Support for Efficient Address Translations in Neural Processing Units. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019). https://api.semanticscholar.org/CorpusID:208139570

[37] Intel. 2023. Intel Virtualization Technology for Directed I/O Architecture Specification. https://cdrdv2-public.intel.com/671081/vt-directed-io-spec.pdf. Referenced April 2023.

[38] 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* (Toronto, ON, Canada) *(ISCA '17)*. Association for

Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[39] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 29–42. https://doi.org/10.1109/ISCA.2018.00014

[40] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*. 341–352.

[41] Seah Kim, Jerry Zhao, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. 2023. AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads. *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2023).

[42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[43] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. 2016. Paravirtual remote i/o. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 49–65.

[44] K. Li, H. Chen, J. Sun, and L. Shi. 2012. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Trans. Comput.* 61, 06 (jun 2012), 804–816. https://doi.org/10.1109/TC.2011.112

[45] Teng Li, Vikram K Narayana, Esam El-Araby, and Tarek El-Ghazawi. 2011. GPU resource sharing and virtualization on high performance computing systems. In *2011 International Conference on Parallel Processing*. IEEE, 733–742.

[46] Tyng-Yeu Liang and Yu-Wei Chang. 2011. GridCuda: a grid-enabled CUDA programming toolkit. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*. IEEE, 141–146.

[47] 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.

[48] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 898–908.

[49] Yiqi Liu, Yuqi Xue, Yu Cheng, Lingxiao Ma, Ziming Miao, Jilong Xue, and Jian Huang. 2024. Scaling Deep Learning Computation over the Inter-Core Connected Intelligence Processor. *arXiv preprint arXiv:2408.04808* (2024).

[50] morenes. 2024. DCRA/DCRA ae. https://github.com/morenes/dcra. Referenced February 2025.

[51] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. 2006. Fast suboptimal algorithms for the computation of graph edit distance. In *Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshops, SSPR 2006 and SPR 2006, Hong Kong, China, August 17-19, 2006. Proceedings*. Springer, 163–172.

[52] NVIDIA. 2023. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. Referenced April 2023.

[53] NVIDIA. 2024. MULTI-PROCESS SERVICE. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Referenced January 2024.

[54] NVIDIA. 2024. Unlock Next Level Performance with Virtual GPUs. https://www.nvidia.com/en-us/data-center/virtual-solutions/. Referenced January 2024.

[55] OpenAI. 2024. Introducing ChatGPT. https://openai.com/index/chatgpt/. Referenced January 2024.

[56] Marcelo Orenes-Vera, Esin Tureci, Margaret Martonosi, and David Wentzlaff. 2024. DCRA: A Distributed Chiplet-based Reconfigurable Architecture for Irregular Applications. arXiv:2311.15443 [cs.AR] https://arxiv.org/abs/2311.15443

[57] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative LLM inference using phase splitting. arXiv:2311.18677 [cs.AR]

[58] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. 2022. Sambanova sn10 rdu: A 7nm dataflow architecture to accelerate software 2.0. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 350–352.

[59] Carlos Reaño, Antonio J Peña, Federico Silla, José Duato, Rafael Mayo, and Enrique S Quintana-Ortí. 2012. CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution. In *2012 19th International Conference on High Performance Computing*. IEEE, 1–10.

[60] Kaspar Riesen and Horst Bunke. 2009. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing* 27, 7 (2009), 950–959.

[61] Kaspar Riesen, Stefan Fankhauser, and Horst Bunke. 2007. Speeding up graph edit distance computation with a bipartite heuristic.. In *MLG*. Citeseer, 21–24.

[62] 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

[63] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2016. GPUvm: GPU Virtualization at the Hypervisor. *IEEE Trans. Comput.* 65 (2016), 2752–2766. https://api.semanticscholar.org/CorpusID:6941728

[64] Michael M Swift, Brian N Bershad, and Henry M Levy. 2003. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 207–222.

[65] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. arXiv:1409.4842 [cs.CV]

[66] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[67] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep S. Sachdev. 2020. Compute Solution for Tesla's Full Self-Driving Computer. *IEEE Micro* 40, 2 (2020), 25–35. https://doi.org/10.1109/MM.2020.2975764

[68] Emil Talpes, Douglas Williams, and Debjit Das Sarma. 2022. Dojo: The microarchitecture of tesla's exa-scale computer. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 1–28.

[69] tenstorrent. 2024. Tenstorrent - Scalable and Efficient Hardware for Deep Learning. https://tenstorrent.com/. Referenced January 2024.

[70] Kun Tian, Yaozu Dong, and David J. Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-Through. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:12658735

[71] Hugo Touvron, Louis Martin, Kevin R. Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Daniel M. Bikel, Lukas Blecher, Cristian Cantón Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony S. Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel M. Kloumann, A. V. Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, R. Subramanian, Xia Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zhengxu Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *ArXiv* abs/2307.09288 (2023). https://api.semanticscholar.org/CorpusID:259950998

[72] Dimitrios Vasilas, Stefanos Gerangelos, and Nectarios Koziris. 2016. VGVM: Efficient GPU capabilities in virtual machines. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 637–644.

[73] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Manabi Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), 1–14. https://api.semanticscholar.org/CorpusID:2310493

[74] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent-v2: Mobile Device Operation Assistant with Effective Navigation via Multi-Agent Collaboration. arXiv:2406.01014 [cs.CL] https://arxiv.org/abs/2406.01014

[75] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, and Steven Swanson. 2017. {vCorfu}: A {Cloud-Scale} Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 35–49.

[76] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. 2012. Transparent accelerator migration in a virtualized GPU environment. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE, 124–131.

[77] Yuqi Xue, Yiqi Liu, and Jian Huang. 2023. System Virtualization for Neural Processing Units. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 80–86. https://doi.org/10.1145/3593856.3595912

[78] 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

[79] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2017. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow

Tasks. *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2017). https://api.semanticscholar.org/CorpusID:9810229

[80] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J Rossbach. 2020. AvA: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 807–825.

[81] Kai Zhang, Bingsheng He, Jiayu Hu, Ze ke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-NET: Effective GPU Sharing in NFV Systems. In *Symposium on Networked Systems Design and Implementation*. https://api.semanticscholar.org/CorpusID:4493567