



**POLITECNICO**  
**MILANO 1863**

SCUOLA DI INGEGNERIA INDUSTRIALE  
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

# Towards Full Peer-to-Peer GPU-NPU Communication on Edge AI SoCs

LAUREA MAGISTRALE IN COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

**Author:** MARCO LAURENZI

**Advisor:** PROF. DAVIDE CONFICCONI

**Co-advisor:** DOTT. ING. GIUSEPPE SORRENTINO

**Academic year:** 2025-2026

## 1. Introduction

For decades, computing performance scaled according to Moore's Law and Dennard scaling. However, *dark silicon* and physical limits of a single transistor dimension are strongly opposing them. Designers responded by trading generality for efficiency, giving rise to Domain-Specific Architectures (DSAs): a specialized unit supports fewer applications but reaches far higher efficiency than a general-purpose core on the workload it targets. Because no single accelerator suits every workload, and because specialization risks tying designed architecture to specific tasks – the *Hardware Lottery* – vendors moved towards heterogeneous Systems-on-Chip (SoCs) integrating several accelerators alongside general-purpose cores. Such systems sit at exactly this intersection, pairing a CPU with devices such as GPUs, NPUs, or FPGAs to combine the efficiency of specialization with the adaptability of a general-purpose host.

**Problem statement.** Heterogeneous integration is no longer an emerging idea: it is approaching saturation, which makes it important to identify where further advantage can still be gained once the architecture is fixed. In an already-integrated SoC, little can be changed at

the device level; the remaining margin lies in *how* those devices communicate.

Communication today is normally copy-based and CPU-driven. Data is physically duplicated from one device's buffer to another's, even when both devices share the same physical memory, and the CPU must mediate every kernel launch and synchronization point. Both are unnecessary overheads for an SoC with shared memory. Avoiding them, however, runs into a second obstacle: accelerator software stacks are frequently closed, undocumented, or – as our target platform – open-source but poorly documented, so the mechanisms needed to bypass redundant copies or CPU-mediated launches must first be reverse-engineered before they can be used.

Aware of such concerns, this thesis studies communication issues on the AMD Ryzen AI 300 (Strix Point) SoC, a monolithic mobile processor that integrates a Zen 5 CPU, an RDNA-based integrated GPU, and an XDNA Neural Processing Unit (NPU) on a single SoC, all sharing the same physical DRAM. Strix Point is representative of a broader trend and is well suited to this study: unlike closed platforms such as Apple's Neural Engine, which is reachable only through the high-level CoreML framework, AMD's stack

is open source and therefore directly inspectable and modifiable. As a target typical pipeline, we consider a GPU kernel followed by an NPU kernel. By default the two are programmed through entirely separate stacks – ROCm/HIP for the GPU, XRT/XDNA for the NPU – each with its own buffers and its own synchronization model, and any data/control dependency between them is realized by routing through the host CPU.

**Contributions.** This work makes four contributions. (i) **A comprehensive characterization of the AMD Strix Point SoC**, covering how the host communicates with the GPU and the NPU through their standard, vendor-intended paths, and how the memory subsystem and the Memory Management Unit (MMU) govern what one device can see of another’s data. (ii) **An analysis of data-movement techniques for GPU-NPU communication** – two zero-copy variants and the Linux `dma-buf` framework – compared against the default copy-based approach across workloads of increasing complexity. (iii) **A reverse engineering of the AMD XDNA software stack and NPU firmware**, exposing how host-to-NPU commands are actually transported and dispatched under the hood. (iv) **A method for peer-to-peer kernel-launch**, allowing the GPU to start an NPU kernel, bypassing existing software stack through an extended GPU driver and a dedicated Linux kernel module.

Together, these contributions show that meaningful efficiency gains remain available at the communication layer of an already-integrated heterogeneous SoC, even when the devices themselves cannot be changed.

## 2. Related Work

This thesis sits at the intersection of two threads: *how* data moves between the components of a heterogeneous system, and *who* is allowed to drive an accelerator. Three works bound it most closely, one on each side of that intersection plus one on the platform itself.

**Zero-copy on a unified-memory APU.** The closest work on data movement is Bertolli et al. [1], who analyze runtime-level zero-copy for OpenMP on the AMD Instinct MI300A, an APU that—like Strix Point—unifies CPU and GPU memory into single physical storage, letting the GPU access host-allocated memory with-

out an explicit transfer. Their results quantify both sides of the trade-off: a 1.2x–2.3x speedup for a production application but up to an 11% penalty on one benchmark, confirming that unified-memory zero-copy is advantageous yet workload-dependent rather than a free win. This thesis differs in two respects: it targets a consumer-class APU that additionally integrates an NPU, and it studies GPU–NPU rather than CPU–GPU data movement, a path the vendor stack does not expose directly.

**Peer-to-peer dispatch.** On the control side, the closest effort is RoPeerTo [3], an open-source, datacenter-scale architecture for peer-to-peer DMA over PCIe validated between GPUs and FPGAs. It targets the same redundant, CPU-mediated copies that degrade performance when heterogeneous accelerators exchange data, reporting a 5.61x speedup and a 37.99% reduction in GPU power by removing the CPU from the data path. The motivation is shared, but the setting differs: RoPeerTo operates over PCIe between discrete datacenter devices, whereas this work operates within a single monolithic SoC over a shared memory fabric, and additionally bypasses the host for *command submission*, not only data transfer.

**This specific NPU.** Finally, Rösti and Franz [2] are, to our knowledge, the only other work targeting this specific NPU: using AMD’s bare-metal toolchain, they fine-tune a GPT-2 model entirely on the client, offloading matrix multiplication from the CPU to the NPU and reporting over a 2.8x speedup on those operations. Their work and this thesis are complementary—both go beneath the high-level libraries to understand the XDNA NPU directly—but where they drive the NPU from the CPU as a training accelerator, this thesis reverse-engineers the same stack to drive it from the GPU and to remove data copies along the GPU–NPU boundary.

## 3. Data-Movement Techniques

We define a *mapping set* as the group of memory regions addressable by a given device: a pointer belonging to the GPU mapping set is one the GPU can directly dereference, and likewise for the CPU and NPU. A data-movement technique is a strategy for handling these mapping sets – moving data and/or references between them so

that the same physical pages become readable by more than one device.

The *copy-based baseline* makes no attempt to share memory: the GPU allocates its own buffer through `hipMalloc`, the NPU works on a separate buffer of its own, and moving data between them requires an explicit copy through the host. Two physical pages exist, one per device, and every transfer physically duplicates the bytes between them. The *HIP-to-XRT zero-copy* technique instead lets the GPU runtime own the buffer and exposes it to the NPU: `hipHostMalloc` allocates a host-accessible, pinned page; `hipHostGetDevicePointer` gives the GPU a handle into that same page; and the page is then wrapped as an XRT buffer object so the NPU operates on it directly – no copy occurs, and a single physical page lives in both mapping sets. *XRT-to-HIP zero-copy* is the mirror image: the NPU runtime allocates the buffer through `xrt::bo()`, and `hipHostRegister` followed by `hipHostGetDevicePointer` registers that same memory with the GPU runtime, reversing which side owns the allocation and which side registers it. Finally, `dma-buf` zero-copy steps below both runtimes and relies on the Linux kernel’s own buffer-sharing framework: the owning device exports its buffer as a `dma-buf` file descriptor, and the other device imports that descriptor, mapping the same physical pages into its own address space without either runtime needing to know about the other’s allocation API.

All three copy-less techniques reach the same end state – a single physical page mapped into both the GPU and NPU mapping sets – but reach it differently: the two zero-copy variants operate at the runtime level and differ only in which side allocates and which side registers, while `dma-buf` operates at the kernel level and requires no runtime-side registration API on either end. How these differences translate into measured performance, and under which workloads each pays off, is analyzed in section 5.

## 4. Reverse Engineering and Direct GPU-to-NPU Dispatch

Removing the data copy addresses only one communication concern; the other regards the commands themselves—who starts an NPU kernel and how. By default these steps are headed

by the CPU, so bypassing them requires reconstructing the kernel launch chain to see how the behavior can be replicated on other devices.

### The host-to-firmware communication path.

Host-NPU communication spans five levels: the host application, XRT, the XDNA kernel driver, the Linux DRM scheduler, and the NPU firmware. The host calls a `kernel()` function that descends through XRT to a `submit()` call entering the XDNA driver; the driver submits the command and sleeps, later woken by the DRM scheduler, which dispatches wake-ups for all queued commands. Once woken, the driver completes the job by writing a message to a specific BAR address and triggering a doorbell register.

As a PCIe device, the NPU exposes several BARs: 1 MB at `BAR0`, 8 KB at `BAR2`, a 512 KB prefetchable region at `BAR4`, and two 4 KB regions at `BAR5` and `BAR6`. The driver maps the relevant ones into kernel virtual memory: `BAR0` backs two ring buffers, one per direction—`x2i` (host → NPU) and `i2x` (NPU → host)—while the smaller BARs expose the head/tail pointer registers coordinating them. The ring is empty when head equals tail, and a producer must never advance tail past head.

Figure 1 shows the handshake. The driver writes the command packet into the `x2i` ring at the current tail offset, advances the tail register, and rings the doorbell. The firmware, polling the `x2i` tail register, reads the command, dispatches it to the AIE array, executes it, writes the response into the `i2x` ring, and advances the `i2x` tail register; the driver, woken by the DRM scheduler, reads the response and advances the `i2x` head register to acknowledge it. The handshake is driven by the pointer updates, not the ring’s content: an omitted tail-register advance—a bug encountered during reverse engineering—leaves the firmware with no signal that new data exists, and the host times out. Each message is a fixed four-dword header followed by an opcode-dependent payload; the header carries the payload size, a packet ID whose top byte is always `0x1d`, and the opcode. Direct NPU dispatch relies on opcode `0x18`, whose `0x18`-byte payload carries, among other fields, the 64-bit NPU-device-space address and size of the compiled *ctrlcode* buffer describing the work to run on the AIE array.

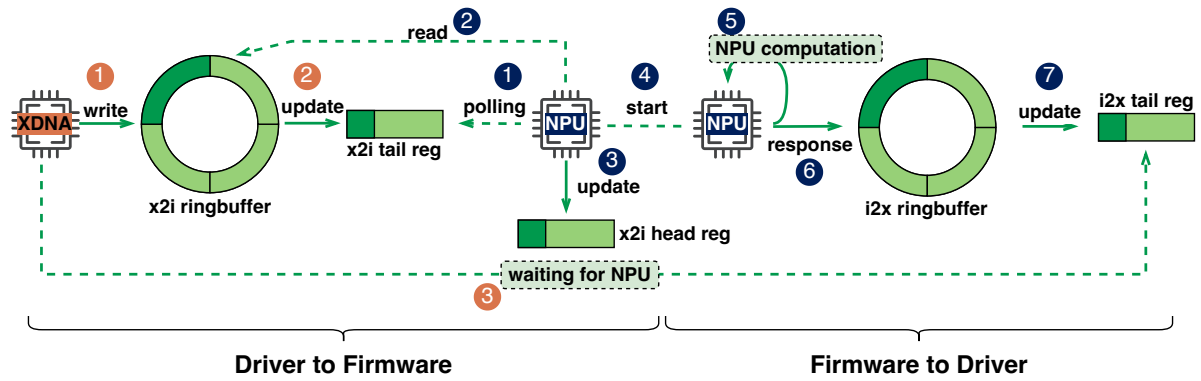


Figure 1: The mailbox handshake between the XDNA driver and the NPU firmware: the driver writes a command into the x2i ring and rings the doorbell; the firmware, polling the tail register, reads, dispatches, and executes the command, then writes its response into the i2x ring.

### From protocol to peer-to-peer dispatch.

In principle, launching an NPU kernel only requires writing this byte sequence to a few mailbox addresses—the firmware cannot distinguish a well-formed packet’s origin. The obstacle is that doing so from the GPU means writing into a PCIe BAR address, which is not backed by a kernel `struct page` and so cannot be imported through the standard `dma-buf` path like an ordinary buffer.

We therefore implement a custom kernel module that takes ownership of the NPU’s BAR and exposes the relevant window—the command ring or the doorbell register—as a `dma-buf`, deceiving the importing driver into treating the BAR address as page-backed. A second obstacle then surfaces: the AMD GPU driver’s import path only accepts `dma-bufs` exported by another GPU. We extend the `amdgpu` import path with a dedicated case recognizing buffers exported by non-GPU devices as legitimate targets, bypassing the exporter checks so the `dma-buf` is handled as a peer GPU buffer would be. The GPU can then write both the dispatch packet and the doorbell directly into the NPU’s BAR, collapsing the host-mediated path into a direct GPU-to-NPU write.

To ease NPU launches from the GPU, we implement two strategies. The *dedicated* one uses two separate launches: one performs the GPU computation, and a second—issued only once the host confirms the first completed—writes the dispatch packet and rings the doorbell, requiring no change to the compute kernel. The *integrated* strategy fuses compute and dispatch into a single

*cooperative* launch: all threads compute, a grid-wide barrier makes every thread’s writes visible, and only then does one thread ring the doorbell, without returning to the host—removing the extra launch and host synchronization, at the cost of requiring cooperative-launch support and direct modifications to the compute kernel’s source.

## 5. Results

We run the experiments on the Strix Point SoC (Ryzen AI 9 HX 370) under Ubuntu Server 24.04 with a custom-built Linux 6.14 kernel. The benchmark pipeline executes a GPU kernel followed by an NPU kernel across four workloads of increasing complexity: *Empty* (no computation, isolates pure communication overhead), *Passthrough*, *Addition*, and *Image Registration Step* (a GPU-side geometric transformation followed by an NPU-side Mean Squared Error evaluation against a reference, compute-heavy). Buffer sizes span 4 KB to 1 GB, inputs are randomized per iteration to reduce caching effects, and each workload–buffer-size pair is built and run in isolation. Each configuration runs a warmup phase followed by 50 measured iterations; results below report the *warm-start* latency, the average of the last 49 iterations, which excludes any one-time initialization cost and reflects steady-state performance.

**Data movement.** Using the *Empty* workload to isolate the pure data-transfer cost (Figure 2), all three copy-less techniques are functionally indistinguishable and outperform the copy-based baseline by **1.36x–1.54x** at large buffer sizes,

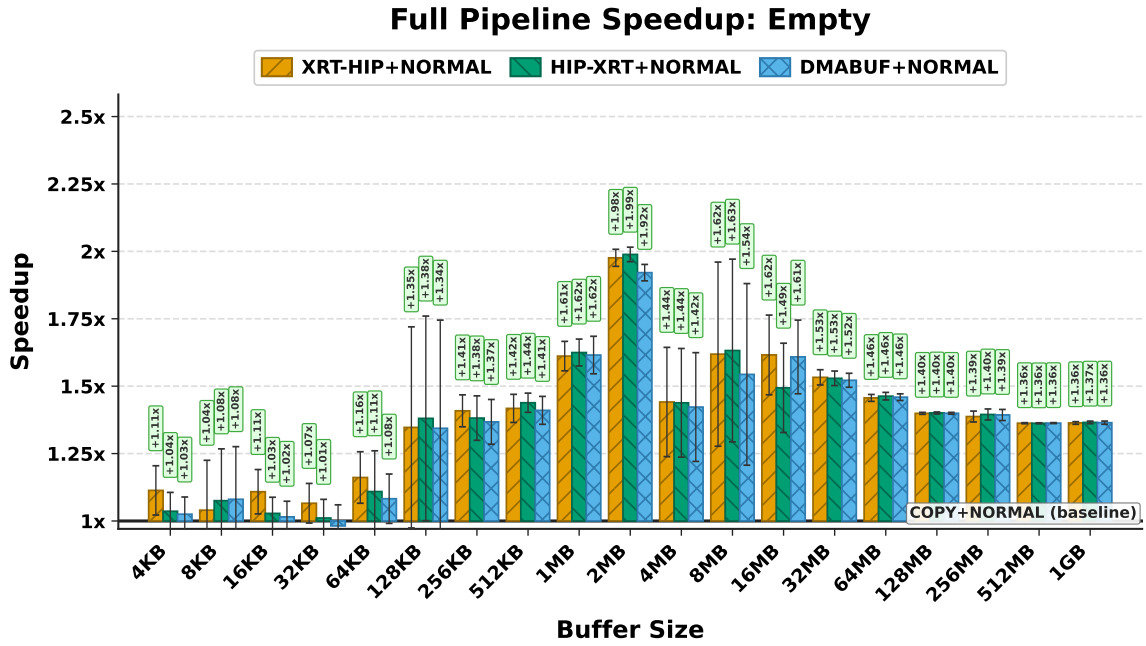


Figure 2: Speedup of each data-movement technique over the copy baseline on the Empty workload, isolating the pure data-transfer contribution.

peaking at **2x** at 2 MB; the advantage is negligible at small sizes, where fixed setup costs dominate for both approaches, and grows steadily as buffer size increases. This ordering holds across all four workloads, confirming that the gain comes from removing the copy itself rather than from the surrounding computation – and, correspondingly, that its ceiling is set by how large a share of total latency the copy represented to begin with. Since `dma-buf` requires no runtime-level buffer registration on either side, it is carried forward as the representative copy-less technique for the remainder of the evaluation. Taking `dma-buf` as the reference data path, the *dedicated* GPU launch shows the characteristic shape of a fixed-overhead mechanism: at small buffer sizes it is slower than both the copy-based baseline and a plain host launch, because the cost of constructing and issuing the command from a separate GPU kernel outweighs the CPU mediation it removes; as buffer size and, more importantly, GPU compute grow, this overhead is increasingly hidden behind useful work, and on the compute-heavy *Image Registration Step* workload the *dedicated* launch delivers **4.4x–6.5x** over the copy baseline. The *integrated* launch (Figure 3), which fuses dispatch into the compute kernel itself, dominates the *dedicated* launch as compute grows – reaching  $\sim 11x$  on

*Image Registration Step* – because the NPU can start running while the GPU is still finishing its own work, rather than only after the GPU kernel exits. The relationship inverts at low compute intensity: on the *Empty* workload, and at small buffer sizes elsewhere, the *integrated* launch is consistently equal to or slower than the *dedicated* one, since embedding the dispatch inside the kernel adds overhead that, with nothing to overlap it against, is never recovered. This makes the *dedicated* launch the more broadly useful default – it requires no access to, or modification of, the compute kernel’s source – while the *integrated* launch is the higher-performance option for workloads where the developer can and will instrument the kernel.

**Data Movement With Kernel Launching.** Because data movement and launch orchestration act on different phases of the pipeline – the transfer and the dispatch – their benefits compose rather than compete: combining `dma-buf` with the *integrated* GPU launch is never counter-productive relative to either technique alone. Figure 3 shows this combination reaching its largest observed gain,  $\sim 11x$ , on the compute-heavy *Image Registration Step* workload, where both optimizations proceed simultaneously with GPU execution. On data-bound workloads with little compute to overlap the dis-

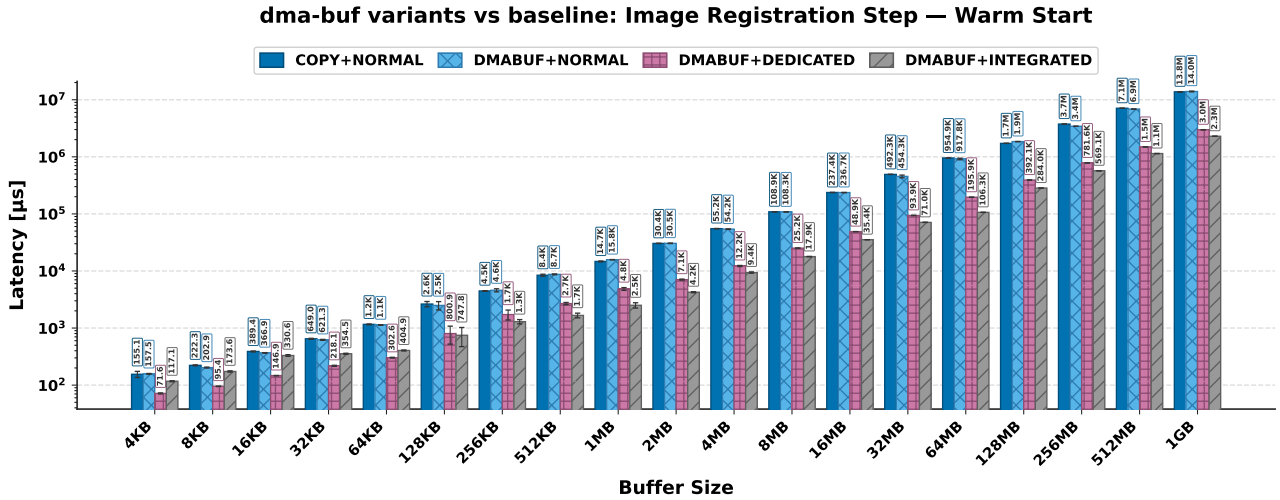


Figure 3: Speedup over the copy baseline for the *Image Registration Step* workload.

patch behind, the combined gain instead tracks the data-movement result already shown in isolation. The Empty workload remains the essential reference point throughout the evaluation: it is the floor showing what these techniques cost when there is no computation to hide their overhead behind, and every other workload’s gain should be read against it, since the benefit of peer-to-peer launch is strictly gated by how much GPU compute is available to overlap the NPU’s startup.

## 6. Conclusions

This thesis studied GPU–NPU communication on the AMD Strix Point heterogeneous SoC along two orthogonal axes: how data moves from a GPU computation to a subsequent NPU computation, and who launches the NPU kernel. On the first axis, both copy-less techniques—zero-copy and `dma-buf`—beat the copy-based baseline, with `dma-buf` the most broadly effective since it needs no runtime-level buffer registration on either device. On the second, dispatching the NPU kernel directly from the GPU outperforms the conventional host-mediated launch once GPU compute is large enough to amortize the dispatch overhead, with the integrated launch increasingly dominating as compute grows. Achieving this required reverse-engineering the XRT/XDNA stack and NPU firmware to expose its mailbox-and-doorbell protocol, plus a custom kernel module and a targeted driver modification that make the NPU’s command region writable directly from the

GPU. Together, these results show that meaningful efficiency gains remain available at the communication layer of a fully integrated heterogeneous SoC, even when the devices themselves cannot be changed.

## References

- [1] Carlo Bertolli, Thorsten Blass, Lynd Stringer, Nicole Aschenbrenner, Jan-Patrick Lehr, Doru Bercea, Dhruva Chakrabarti, Lawrence Meadows, and Ron Lieberman. Performance analysis of runtime handling of zero-copy for OpenMP programs on MI300A APUs. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024.
- [2] André Rösti and Michael Franz. Unlocking the AMD neural processing unit for ML training on the client using bare-metal-programming tools. arXiv preprint arXiv:2504.03083, 2025.
- [3] Marco Venere, Giuseppe Sorrentino, Benjamin Ramhorst, Maximilian Jakob Heer, Lucian Petrica, Dario Korolija, Marco D. Santambrogio, Davide Conficconi, Gustavo Alonso, and Kenneth O’Brien. RoPeerTo: A datacenter-scale architecture for peer-to-peer DMA between GPUs and FPGAs. In *Proceedings of the Twenty-First European Conference on Computer Systems (EuroSys ’26)*. ACM, 2026.