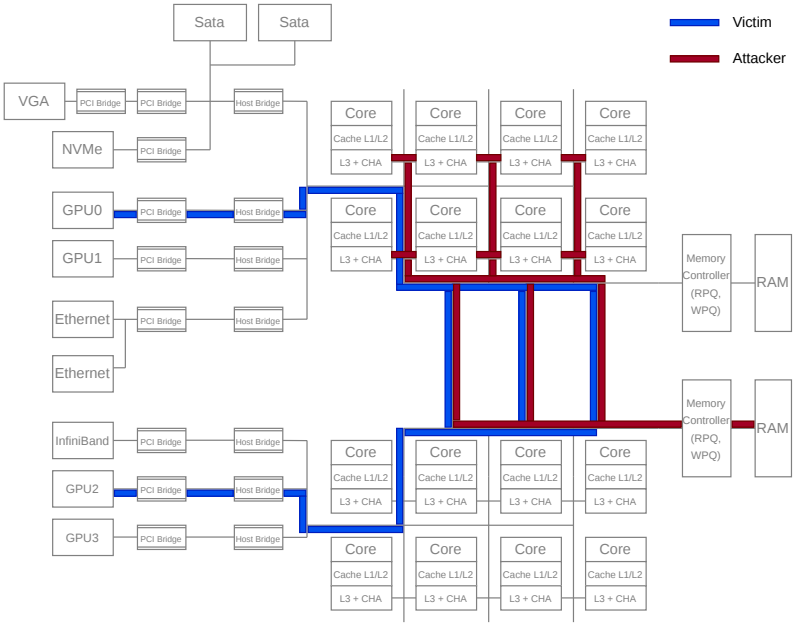
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Congesting Distributed AI Within the Host Network

Master's thesis in Computer science and engineering

WILLIAM ERIKSSON KLING

SIMON GUNNARSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Congesting Distributed AI Within the Host Network

WILLIAM ERIKSSON KLING

SIMON GUNNARSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Congesting Distributed AI Within the Host Network

William Eriksson Kling, Simon Gunnarsson

Supervisor: Muoi Tran, Department of Computer Science and Engineering
Examiner: Miquel Pericas, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Example of the data traffic paths of a congestion attack on the host network

Typeset in LaTeX
Gothenburg, Sweden 2026

Congesting Distributed AI Within the Host Network

William Eriksson Kling, Simon Gunnarsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As artificial intelligence models scale, they rely on distributed, multi-tenant hardware. In these environments, the internal host network of e.g. CPUs, GPUs, memory controllers and inter-socket links is critical to transferring data efficiently. The internal host network is often viewed as a trusted environment, but congestion of the shared paths can have substantial effect on performance.

Consequently, this thesis addresses the following research question: Can a co-located adversary intentionally utilize host network congestion to execute an attack against distributed AI workloads?

To evaluate this potential threat, we developed adversarial workloads mainly targeting the memory and the inter-socket link. These attacks were executed concurrently with Transformer and Graph Neural Network (GNN) models while capturing performance and hardware statistics.

The results show that an adversary can weaponize the host network and cause substantial end-to-end performance degradation for the AI models. The magnitude of the performance degradation depends on how the models use the hardware. Models that continuously rely on the CPU memory for data suffered the largest performance drops, when an attacker saturated CPU memory. In contrast, the model that kept it's data in the GPUs' VRAM and synchronized peer-to-peer was immune to this kind of attack. It was however vulnerable to an attack against the inter-socket path.

Essentially, this thesis demonstrates a vulnerability in multi-tenant high-performance computing. An adversary does not need to break virtual machine boundaries or have elevated privileges to cause harm. Exploiting the host network can cause damage in terms of performance degradation. Expensive GPUs being underutilized translates to a substantial indirect financial cost.

Keywords: Host network, distributed AI, hardware security, congestion attacks.

Acknowledgements

We would like to thank our supervisor, Muoi Tran, for his guidance throughout this thesis project. We also thank our examiner, Miquel Pericas, for providing valuable feedback during our midterm presentation, and Weijia Shao for the many helpful discussions. Finally, we are grateful to the administrators of the Vera and Minerva clusters for providing the essential computing infrastructure that made our experiments possible.

William Eriksson Kling, Simon Gunnarsson, Gothenburg, 2026-06-11

Contents

List of Figures

List of Tables

1

Introduction

The rise of deep learning models has driven a high demand for distributed and high performance hardware. Training modern AI models requires coordination of large clusters with accelerators such as GPUs. When the accelerators' computational power has increased, another bottleneck has emerged: the host network.

The *host network* is the internal communication network of a computing host for efficient data transfer between CPUs, memory and peripherals. Distributed AI training relies heavily on synchronization of tensors and gradients across the host network. The processing power of GPUs has increased and can outpace the bandwidth of cross-device communications. Data traversal across the internal host network can account for a substantial portion of the total training time [1].

To maximize utilization of hardware, high-performance computing clusters rely on multi-tenancy and co-location of workloads. In shared environments, multiple processes from different users are being executed concurrently on the same silicon. Studies have shown that the host network is vulnerable to interference between workloads [2]. Benign co-located workloads cause contention within memory controllers and other shared components, negatively affecting end-to-end application performance. The consequences of performance degradation due to contention could be severe. Modern hardware is expensive, and performance degradation could cause wasted compute budgets and delayed release of models.

Despite this performance vulnerability, the security of the host network remains largely unexplored. Previous studies have focused on virtualization, resource isolation and data confidentiality [3]. The internal host network often lacks hardware-level Quality of Service (QoS) [4] and is in practice treated as a trusted environment.

This leaves a research gap regarding adversarial use of host network contention. If benign contention can cause performance degradation, a malicious actor could theoretically exploit hardware bottlenecks to orchestrate a QoS degradation attack. This thesis addresses that research gap by asking this question: Can a co-located adversary intentionally utilize host network congestion to execute an attack against distributed AI workloads?

To answer this, we provide three main contributions. First, we evaluate the host network as a viable attack vector for QoS degradation attacks. Second, we develop adversarial workloads that target specific bottlenecks in the host network, e.g. memory queues and inter-socket links. Finally, we compare and analyze how different

AI workloads and hardware impact the resilience against such attacks.

The remainder of this thesis is structured as follows. Chapter 2 provides a background on host networks and introduces software tools that were used. Chapter 3 details the threat model, victim and attack workloads and the evaluation procedure. Chapter 4 presents the experimental results and throughput degradation metrics. Finally, chapter 5 discusses implications of these findings and concludes the thesis with a section about future work.

2

Theory

This chapter gives a theoretical background for the thesis by explaining the host network, introducing the software tools used and describing the AI model architectures.

2.1 Host Network and Contention

Within a host computer, the network that enables data transfer between the processor, memory, and peripheral interconnects is called the *host network*. This is how Vuppalapati et al. define it in their research-article "Understanding the Host Network" [2]. Using the concept of domain-by-domain credit-based flow control, one can analyze how contention can affect sub-networks within the host network and the impact it has on different running applications. The credit-based flow works such that each component (e.g. processor) is assigned a set amount of credits, where one credit is consumed for sending a message, and then restored when receiving an acknowledgment from the recipient [2].

There are two different application types that can be tested for the host network: P2M applications generating peripheral-to-memory traffic and C2M applications generating compute-to-memory traffic [2]. According to the study "Understanding the Host Network", there are two scenarios that appear during contention when running P2M and C2M applications concurrently. First, the performance of both P2M and C2M applications can suffer during simultaneous memory writes. Second, the performance of C2M applications can degrade with between 1.2-1.7x with minimal to no effect on the performance of P2M applications, even without saturating memory bandwidth. The scenarios were tested on two different testbeds to show that this is a general observation worth examining according to the authors [2].

A simple host network can be described as a set of interconnected components:

- Processor cores and their designated L1/L2 caches (with line filler buffer (LFB) inbetween).
- The last-level cache (LLC) with a caching and home agent (CHA).
- The memory controller (MC) and DRAM.
- The integrated IO controller (IIO) with connected peripheral devices [2].

This network can be divided into C2M datapaths and P2M datapaths, and then

further be grouped into domains, or sub-networks, for the credit-based flow control (see figure 2.1). The "bottleneck" domains are as follows:

- C2M-Read: From LFB to CHA, CHA to MC, MC to DRAM
- C2M-Write: From LFB to CHA
- P2M-Read: From IIO to CHA, CHA to MC, MC to DRAM
- P2M-Write: From IIO to CHA, CHA to MC

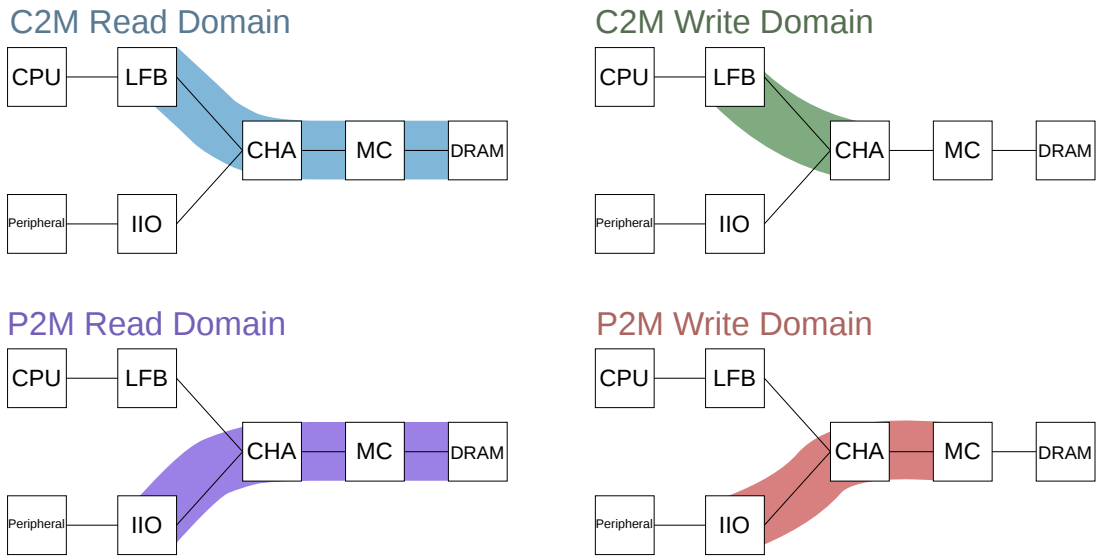


Figure 2.1: Paths for the Core-to-memory (C2M) and Peripheral-to-memory (P2M) read and write domains in the host network. Adapted from "Understanding the Host Network" [2]

C2M and P2M requests only traverse the entire datapath if they encounter cache misses at every level of the hierarchy.

Different domains have a different number of credits and, due to the specific components included in the domain, different unloaded latencies (latency without other traffic). This is also what determines the calculation of end-to-end performance for analysis when traversing the sub-network between sender and receiver [2]. Additionally, the MC has two different queues for reads and writes, which it operates on separately by switching modes. This adds idle time when switching. According to Vuppalapati et al. one can calculate a bound for maximum throughput for any domain using:

$$T \leq \frac{C \cdot 64}{L} \tag{2.1}$$

Where T is maximum throughput, C is hardware-specific number of credits, L is latency for traversing domain, and 64 is the cacheline size in bytes [2].

There are three ways for C2M to experience degraded performance without affecting P2M applications: C2M-Read with P2M-Write, C2M-Read with P2M-Read, and C2M-Read/Write with P2M-Read [2]. This phenomenon can be explained by the

processor-core's ability to issue instructions fast and queuing when reaching the MC. First, instructions are issued fast enough to fill the LFB, which is used to issue requests to the L2 cache. Second, since the DRAM is part of the C2M-Read domain, any potential queuing at the MC will determine the performance. This can happen before memory bandwidth saturation. Any row misses at the DRAM will result in a delay for the memory banks, no longer allowing the MC to hide any latency as a consequence of opening and closing rows [2]. Requests to memory banks are not load balanced, because of how memory addresses are mapped using a hash function, leading to blocking of requests (and resulting in queuing).

Both P2M and C2M applications suffer when doing C2M-Read/Write and P2M-Write. Vuppalapati et al. show in "Understanding the Host Network" that when C2M applications are assigned to less than three cores, their performance degrade in a similar manner to what is mentioned above [2]. However, when memory bandwidth is saturated (typically as a result of assigning C2M to three or more cores), P2M applications start to experience throughput degradation. Because the P2M domain, but not the C2M domain, spans to the MC, when the MC queues are filled up it will impact the performance of P2M workloads. When assigning more cores (beyond four cores in the study) to C2M tasks, the CHA starts to limit requests due to limited buffering resources [2]. As a result of this, the CHA applies backpressure to the cores, which then blocks requests earlier and impacts C2M domains too.

2.2 Performance monitoring units (PMUs)

When investigating host network contention, software-based profiling such as execution time is not sufficient on its own. Application-level metrics can indicate that there is contention, but provides no information on where the contention occurs in the host network. To be able to observe contention more in detail, many processors have hardware performance monitoring units (PMUs) integrated in the silicon.

In recent Intel architectures, there are two main categories of PMUs:

- **Core PMUs:** Located inside of the CPU cores. They monitor events such as L1/L2 cache accesses, instruction retirement and branch mispredictions [5].
- **Uncore PMUs:** Located in the mesh network, outside the CPU cores [6]. These counters are crucial when capturing host network congestion, since they monitor the resources that are shared among the CPU cores. Shared resources are for example the Caching and Home Agents (CHAs), Integrated Memory Controllers (iMCs), Integrated I/O blocks (IIOs) and UPI links between sockets.

2.3 LIKWID

LIKWID, "Like I Know What I'm Doing", is a set of tools developed at university of Erlangen-Nuremburg, giving users tools to monitor and benchmark their systems and programs [7]. One of LIKWIDS main tools, called "LIKWID-perfctr", is used to monitor hardware counters available using PMUs, entirely in the user space. Perfctr

offers performance groups with pre-selected events to monitor the counters, but the option of creating your own set of events is also available to the user. To decide what to monitor, Likwid also offers lists of all available events and counters (the number of events that can be captured is limited to the number of physical counters) [7]. For use within HPC clusters, LIKWID offers a security solution called "accessD" to counter the risks of allowing read and writes to model-specific register (MSR) device files. By running a daemon for each LIKWID application that needs MSR read or write permission, only the daemon will have access to the MSR files. It will check and validate any attempts to access a register by the LIKWID application, and if access to a register is denied, the attempt will be dropped. Any allowed register is hard-coded into the daemon as a security measurement [7].

2.4 Perf

Perf is a performance tool to monitor performance counters in the kernel space, included in the Linux kernel [8]. It allows monitoring of both software events using kernel counters and hardware events using PMUs, and a list of available events can be provided by Perf. The events are selected using either a moniker or by providing the hexadecimal for the event and the event-mask for any sub-events. There is no limit to the number of events to monitor; however, there are two limiting factors [8]. Each event uses one file descriptor, thus any limit set by kernel in regards to maximum number of open file descriptors can set a limit of events. In addition, the maximum number of events that can be monitored at any given time is limited to the number of counters available. Perf handles more events by introducing multiplexing, where events are programmed onto counters and then switched in a round-robin fashion to allow all events to access the hardware (assuming no conflicts between events running at the same time). Perf then gives an estimate result by scaling the result for each event by the total running time of the application.

2.5 Apptainer

Apptainer is a platform for building and running software containers, designed for use with HPC clusters by Lawrence Berkeley National Laboratory [9]. The platform provides its users with the ability to create a portable and reproducible environment in which to run applications. The user first creates an Apptainer definition file, a file that defines environment options such as; base operating system or an already created container image to build from, environment variables, software, container validation tests, and more. This definition file can then be built into a Singularity Image File (SIF), the environment. Security is implemented by running the container as the user starting it and applying security controls, such as file permission, in the same way as running a normal process [9].

2.6 Topology

By mapping out the topology of a computer, it is easier for an attacker to find vulnerabilities that can be used to create congestion within the host network. For the task of studying the network, there are many available tools. Some tools require elevated permissions or root privileges to display all information, which can hinder an attacker. The following is a list of tools available.

- **lscpu**: This is part of a collection of utilities called util-linux [10]. The output of the command show CPU-information such as model, sockets and cores, flags, cache sizes, and more. The output can also show a list of know vulnerabilities for the model and status information of the attack. For example, "Vulnerability Meltdown: Not affected" or "Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and ___user pointer sanitization". The displayed information is gathered from sysfs, /proc/cpuinfo/, and other libraries used for CPU architecture.
- **numactl**: This tool when used together with setting "–hardware" will show current available NUMA nodes [11]. It will show all CPU cores associated with a node, as well as the memory of the node (both total memory and currently free memory).
- **lspci**: This command will display all PCI buses for the host network, together with any devices connected to it [12]. It can show which subsystem a device is within, to which NUMA node it is connected to, details about PCI bridges, and more. This tool is also a part of the pciutilities project.
- **ibv_devinfo**: This tool is part of the "RDMA Core Userspace Libraries and Daemons" (RDMA core) project [13]. It will output all available RDMA devices and detailed information for each. Examples of the information are host channel adapter (NIC), transport (e.g. InfiniBand), and current status of the port.
- **nvidia-smi**: For Nvidia specifically, this command can be used to display information about all the detected Nvidia devices [14]. The result is represented as grid showing the connections between GPUs, connection between a GPU and NIC, but also to which NUMA node a GPU belongs to.
- **lstopo**: As part of the "Hardware Locality" (hwloc) project, the use of this tool will display, either graphical or through the terminal, an image of the host networks topology [15]. It will show each NUMA node with its cores and respective cache, host bridges and PCI bridges connecting devices with the CPU, as well as which device is connected at each end.

2.7 Deep Learning Models

The following subsections introduce the machine learning concepts relevant to this thesis. First neural networks are introduced and then the targeted architectures Graph Neural Networks and Transformers are described.

2.7.1 Neural Networks

Neural networks, which are inspired by how the human brain works, are some of the most used machine learning models today [16]. These networks are made up of components called perceptrons that work in a way similar to neurons in the brain. These perceptrons, combined into different interconnected layers, take raw input for which it learns to find what parts are important, any useful patterns in the input, and context between elements. This trained model is then used to make predictions, given a specific task, on new data that are given [16].

2.7.2 Graph Neural Network

A Graph Neural Network, or GNN, is a neural network model that takes as input both a graph of elements and a graph that represents the connections between different elements. Some applications of GNNs include computer vision, such as capturing relationships between objects in the image, and genomic sequencing or drug discovery in pharmaceutical research. As an answer to the limitations of convolutional neural networks, which expect a grid-like input, and recurrent neural networks, which expect sequence structured input, a node in a GNN model can have connections to many different other nodes. The number of connections for each node can also vary in the graph [17].

The prediction task of a GNN model can be sorted into three different types of predictions: graph-level, node-level, and edge-level [18]. Graph-prediction tasks look at the properties of the entire graph to, for example, predict traits of a molecule. This is comparable to labeling an image or predicting the emotion of a sentence. Node-level predictions can be used for tasks such as predicting the identity of each node, similar to image segmentation tasks or predicting the word classes of each word in a sentence. Lastly, edge-level predictions can, for example, be on the type of relationship different nodes have to other nodes. It can also be used to predict whether there should be a connection between two nodes or not.

Information about the three levels can be represented as embeddings [18]. The data representing the input graph are mapped into different feature vectors that represent the edges, nodes, and graph, respectively. When training the model to learn more about the graph, it will use these vectors together with a message-passing technique. This is done by gathering all embeddings of its neighbor and aggregating these "messages" with the node's embedding vectors, consequently updating the information a node can represent. The size of the neighborhood a node represents depends on the layers in the GNN [18]. Each layer of the model adds one hop during the message passing, e.g. a 5-layered GNN model can gather information about neighbors 5 hops away. All messages are also passed through a function to update the weights of the GNN, which are global and shared between all nodes.

2.7.3 Transformer

Before 2017, tasks that are based on predicting the next item in a sequence, such as translating or generating text, were dominated by recurring neural networks

(RNNs). RNNs are limited in the sense that they require everything to be processed sequentially, thus not being able to make good use of parallelism in modern hardware. In the paper "Attention Is All You Need" [19], the authors introduce the transformer, which could replace the step-by-step RNN with a mechanism called Self-Attention.

Instead of reading items left-to-right, a transformer evaluates all items simultaneously to calculate how much each item relates to all other items. This is done by using three vectors; Queries (Q), Keys (K) and Values (V). The attention is calculated by taking the dot-product of Q and K, scaling it, and then applying a softmax to get weights. The weights are then multiplied with V to get the output:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.2}$$

In order to recognize multiple correlations simultaneously, this math is done in parallel by using Multi-Head Attention. Since transformers process everything in parallel and therefore lose the word order, positional encoding are added so the model can understand distance between words.

3

Methods

This chapter describes the methodology used to evaluate AI workload resilience against host network contention attacks. First the threat model is defined. Then the hardware topologies and orchestration framework are characterized. Then victim AI applications and the implemented attacks are described. Finally, the experimental procedure is outlined.

3.1 Threat Model

This thesis evaluates host network vulnerabilities with a multi-tenant threat model. The adversary poses as a legitimate, unprivileged user on a shared HPC cluster (e.g. Vera or Minerva). The adversary is able to submit jobs, ask for resources, and use commands in the same way as an ordinary user. The goal of the attacker is to degrade the performance of co-located AI workloads. The attacker assumes or has knowledge that other AI models are running on the shared computer cluster. However, they do not know the exact host network setup of the cluster nodes and will have to find this information to decide on the point of attack. Note that during the experiments, victim and attack workloads were run within the same batch jobs to allow orchestration. This deviates slightly from the theoretical threat model.

3.2 Hardware Setup (host network probing)

Each host network included in the tests was probed for information about the structure of the system. This was done using the tools described in 2.6 together with the information displayed on the websites for each node cluster [20][21]. From an adversary's perspective, it is important to recognize that the output of tools such as lscpu, numactl and lspci reveal different information depending on the user's privilege level. However, an adversary does not need complete and highly detailed architecture blueprints. Access to basic information such as NUMA core affinity and GPU socket placement in combination with information from the publicly available websites gives enough information to weaponize the host network.

3.2.1 Vera Cluster

Vera is a Linux-based high-performance computer system hosted by Chalmers and C3SE [20]. The HPC cluster is made up of different hardware configurations per node, with components shown in table 3.1. The structure of a node can also be found in figure 3.1.

Table 3.1: Vera cluster hardware specifications

Subsystem	Specification
CPU	2x Intel Xeon Gold 6338
Core Topology	64 physical cores (32 per socket)
Socket Interconnect	3x UPI links at 11.2GT/s each
Memory	512/768/1024/1536/2048 GiB DDR4
GPU	Nvidia A40 (45 GiB VRAM per device)

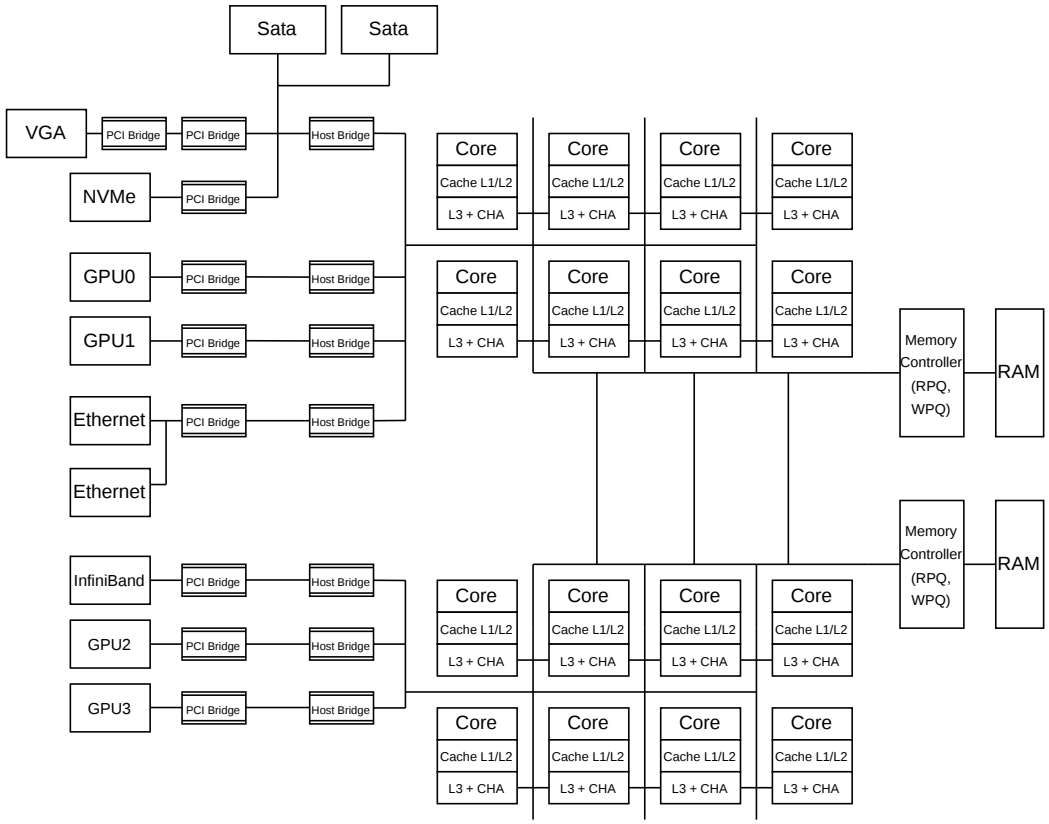


Figure 3.1: Structure of a node in the Vera cluster

3.2.2 Minerva Cluster

Minerva is a computer cluster hosted by the Computer Science and Engineering department at Chalmers and the University of Gothenburg [21]. All nodes have the same setup except for the GPUs, where there are two different setups, see table 3.2. For the node structure, see figure 3.2.

Table 3.2: Minerva cluster hardware specifications

Subsystem	Specification
CPU	2x Intel Xeon Gold 6548N
Core Topology	64 physical cores (32 per socket)
Socket Interconnect	3x UPI links at 20GT/s each
Memory	512 GiB DDR5
Local storage	2x 1.92 TB SSD
GPU	8x Nvidia L4 (24 GiB VRAM per device)

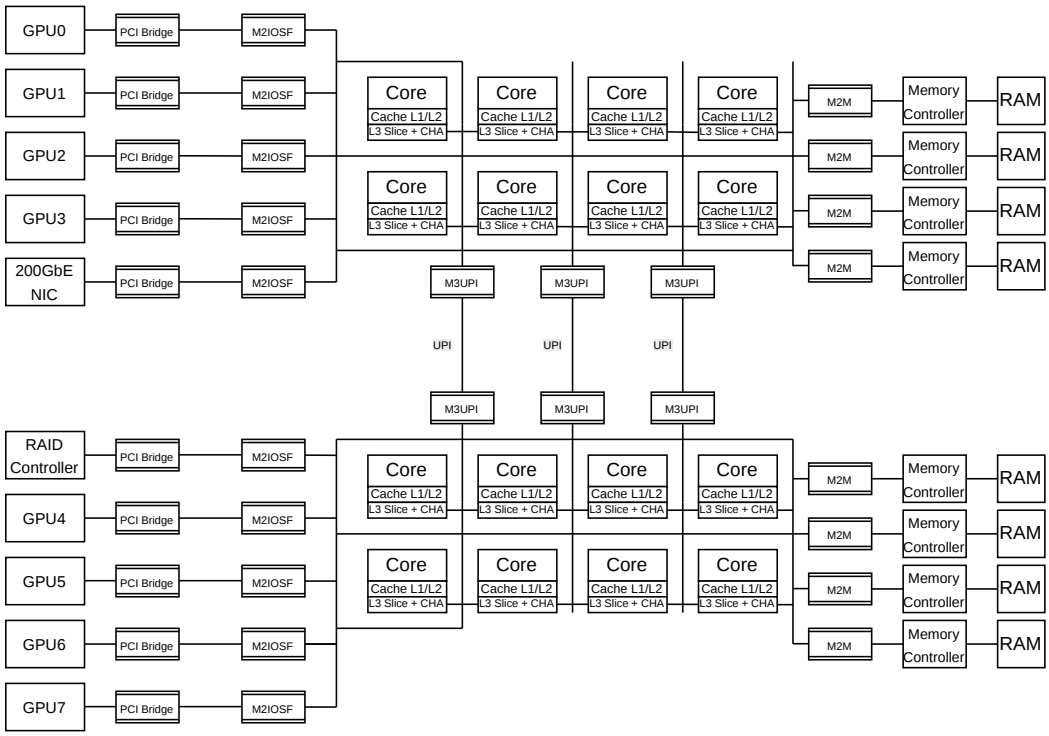


Figure 3.2: Structure of a node in the Minerva cluster

3.3 Orchestration and Measurement Framework

To run experiments with two co-located applications and quantify results, the tool Mio from the "Understanding the host network" paper [2] was used. Mio provides the following functionality:

- **Deterministic CPU core pinning:** To ensure that any recorded performance degradation is due to contention - rather than the operating system's CPU scheduling and context switching - Mio enforces core affinity. The AI workload is isolated to a subset of physical cores while the attacker script is isolated to another non-overlapping subset of cores.
- **NUMA memory binding:** In a multi-socket environment, physical memory allocation affects which paths memory accesses traverse. To control this allocation and prevent the operating system from dynamically moving data, Mio enforces NUMA binding. This strict allocation enables controlling whether traffic stays on the local socket or needs to traverse the inter-socket connection.
- **Execution synchronization:** To capture accurate measurements, it is important that the co-located applications are started at the same time. This is done in Mio by first letting the two applications initialize. When both are done with their initialization, they are started simultaneously.

However, Mio relies on reading Model Specific Registers (MSRs) to monitor uncore PMUs. Accessing MSRs typically requires sudo/root privileges. To ensure security, users of the clusters do not have root or sudo access. To bypass this limitation, the tool LIKWID was considered as an alternative. LIKWID was used to print information about available counters; however, it was found to have the same limitation as Mio, it being unable to read MSRs. The solution to this issue was to use the Linux performance analyzing tool, *Perf*. Perf is integrated with the Linux kernel and can expose uncore PMUs to user-space, given that the administrator configures perf_event_paranoid to be 0. The Minerva cluster already had that as default. For Vera, the administrator implemented a setting in Slurm that allow exclusive jobs to run with perf_event_paranoid set to 0. By combining Mio with Perf, this setup provides both workload orchestration and hardware performance measurements, without the need for sudo privileges. It is worth noting that access to uncore counters was required for this thesis to quantify host network contention. However, an attacker can still blindly execute the adversarial workloads and cause performance degradation without perf_event_paranoid set to 0.

3.4 Victim

This section describes the AI applications used as victim workloads for the experiments in this thesis. Specifically, it outlines how each model accesses and synchronizes data across the host network.

3.4.1 CPU-bound Transformer

The transformer is implemented with PyTorch and executed with CUDA. The Vera environment utilized Pytorch 2.1.2 and CUDA 12.1 while Minerva used Pytorch 2.9.1 and CUDA 12.8. The workload performs the scaled dot-product attention described in equation 2.2 and then a backward pass to calculate gradients. The workload uses the following dimensions: a batch size of 16, a sequence length of 1024, and 32 heads

with a head dimension of 128. Calculations are done on the GPUs but the tensors (Q, K and V) reside in the host's CPU memory. This data then has to traverse the CPU mesh to reach the GPUs in every iteration. First the GPU requests data from memory. The data is then fetched from the DRAM modules by the Memory Controller. Then the data is sent over the mesh to the PCIe bus to the GPU.

3.4.2 GPU-GPU Transformer

The GPU-GPU Transformer relies on the same PyTorch and CUDA environments described in section 3.4.1, with the addition of NVIDIA Collective Communications Library (NCCL) [22] for GPU-GPU communication. It performs the scaled dot-product attention described in 2.2 and then a backward pass. It uses the same parameters as the CPU-bound workload in section 3.4.1. However, the tensors are stored directly on the GPU memory (VRAM) instead of the CPU memory. The calculation outputs are then aggregated between the GPUs. By forcing direct peer to peer communication, data is sent over the mesh and UPI links and bypassing the CPU memory.

3.4.3 Graph Neural Network

The Graph Neural Network (GNN) model is implemented using PyTorch Geometric 2.8.0, a library built on PyTorch specifically for GNNs [23]. The model and library is run using PyTorch 2.8.0 and CUDA 12.8.

Table 3.3: Configuration of a simple graph neural network

Layer	**Input Dim**	**Output Dim**	**Info**
Input Data	$N \times d_{\text{in}}$	—	$d_{\text{in}} = \#\text{features}$
Graph Conv 1	$N \times d_{\text{in}}$	$N \times 32$	Hidden Channels: 32
Activation Function	$N \times 32$	$N \times 32$	ReLU Function
Regularization	$N \times 32$	$N \times 32$	Dropout Probability $p = 0.2$
Graph Conv 2	$N \times 32$	$N \times 1$	Output Channels: 1
Output Activation	$N \times 1$	N	Sigmoid, Flattened

The description of the GNN model can be found in table 3.3. It is made up of 2 graph convolution layers, connected using a ReLU activation function and a dropout layer. The output layer is a sigmoid function, used for binary classification, and is combined during training with binary cross entropy as a loss function. During each test, the model is training for 5 epochs. The optimizer used for training is an AdamW optimizer, with a learning rate of 0.01 and a weight decay of 1^3. Finally, as a loss function, a binary cross entropy function was used.

The input graph used for testing is acquired by generating random nodes, node features, and dummy labels. It includes 500 000 nodes with 1.25×10^8 randomly connected edges. Each node contains 512 features, each value randomly generated using the PyTorch library, and each node is assigned a random binary classification between 0 and 1 also using the PyTorch library.

The model can be run on either the CPU or a GPU, and the graph is saved on the DRAM. When running it on a CPU, the data is fetched from the memory and calculations are done on the CPU cores. When running the model on a GPU it instead uses a batch loader of size 4096 that fetches smaller sub-graphs or neighborhoods from the DRAM to the GPU. Each neighborhood is sampled using hops, communicating with 15 neighbors for the first layer and 10 for the second.

3.5 Attacker

This section introduces the adversarial workloads used in this thesis. Each attack is designed to target a specific part of the host network, with the overall goal of congesting shared resources and cause performance degradation.

3.5.1 Memory Attack

The memory attack is designed to saturate the DRAM by generating a large volume of read-modify-write requests. The attacker is implemented as a multi-threaded C application, spawning one thread for each core assigned to the adversary. The design of this attack is motivated by the background presented in section 2.1. This attack acts as an aggressive Compute-to-Memory (C2M) application. The objective is to congest the memory controllers, CHA queues, and DRAM channels, and thereby increase the memory access latency for the co-located C2M or Peripheral-to-Memory (P2M) workload. An example of the data paths for this attack on the Vera architecture can be seen in figure 3.3.

To ensure that the memory traffic reaches the DRAM and not just uses the CPUs cache, each thread allocates a 64 MB memory buffer. With this size exceeding the size of the Last-Level Cache (LLC), requests have to be served by the main memory. After allocation, the buffer is initialized using memset to prevent lazy allocation.

Once the GO signal from Mio is received by the attacker, each thread iterates through it's buffer. This is done with a 64 byte stride, corresponding to the cache line size of the processor. This access pattern ensures that every iteration touches a new cache line, forcing cache line fills from the main memory. The attacker's traffic traverses the whole C2M path (LFB $\rightarrow$ CHA $\rightarrow$ MC $\rightarrow$ DRAM), consuming the finite flow-control credits assigned to those domains. The application reads a single byte, inverts the bits, and then writes it back to the memory.

For further testing, the memory attack can be design to work in bursts instead where the data access is done in batches. The bursts are calculated by comparing the difference in time between the start of the burst attack and the current time with a set amount of time for each burst. Between each burst of data, the process sleeps for 5 seconds.

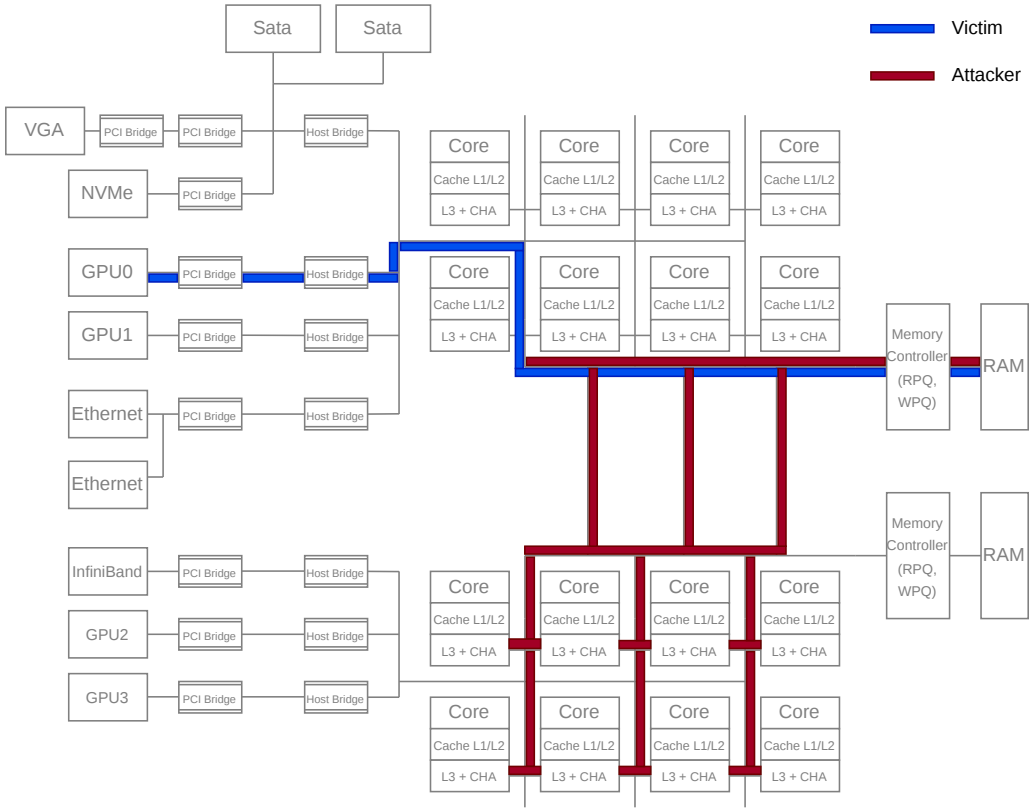


Figure 3.3: Travel path of victim data and attacker data for memory attack

3.5.2 UPI Attack

The UPI attack uses the same underlying C program as the memory attack, but exploits the Non-Uniform Memory Access (NUMA) architecture of the node to target the inter-socket links. The two sockets are connected by Intel Ultra Path Interconnect (UPI) links.

The UPI attack spawns multiple threads on one socket (e.g. socket 0), corresponding to the number of cores assigned to the attacker. Then the threads allocate their memory buffer to the other socket (e.g. socket 1) using numactl. This forces all memory accesses to be remote. Forcing remote memory accesses extends the attacker's C2M path across the UPI link. As shown in figure 3.4, all requests must traverse the inter-socket link. By flooding this link, the attacker consumes credits in the UPI link domain. When credits are exhausted, requests are queued waiting for credits to be restored. This could cause performance degradation for applications that rely on cross-socket communication. An example of a cross-socket application is AI training that uses multiple GPUs attached to different sockets.

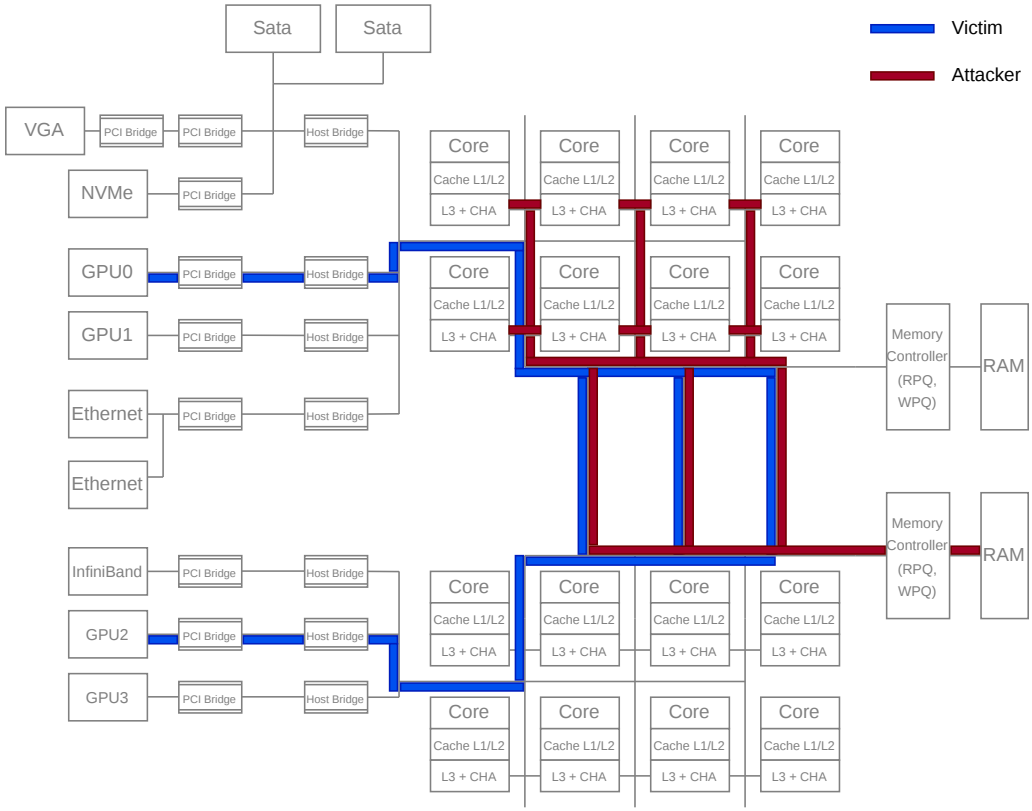


Figure 3.4: Travel path of victim data and attacker data for UPI attack

3.5.3 Mesh Attack

The mesh attack is designed to create traffic between cores. Unlike the previously mentioned attacks that target external points, this attack is targeting the internal mesh architecture. This is done by weaponizing the cache coherence protocol.

The attack forces continuous cache invalidations between physically separate cores. Threads are spawned according to the number of attacker cores defined through Mio. The cores are then split into pairs. For each pair, a block of memory is allocated. In the attack loop, the two threads write to their shared memory location. When thread A on core 0 writes to a cache line, the Caching and Home Agent (CHA) marks the line as modified and it is in that core's private L2 cache. Then thread B that resides on core 1 tries to write to the same memory location. Since that cache line has been modified, it has to be evicted from core 0's private cache and sent to core 1. Core 1 then writes and the data goes to core 1's private cache. Then when core 0 tries to write again to the same memory location, the same process begins but in reversed order.

3.5.4 GPU-GPU Attack

A GPU-GPU attack is based on utilizing the ability of GPU devices to communicate directly with each other using Peer2Peer or similar methods. For this project, a GPU device that runs an AI model is designated as a victim, with one or more GPU devices designated as attackers. The base setup is to assign a victim GPU and an attacker GPU to the same NUMA node on the host network. Any other GPU devices used for an attack are assigned to the other NUMA node. There are two different ideas behind the attacks that were tested.

3.5.4.1 Communication Within Same NUMA Node

The attack involves one victim and one attacker. The goal is to create congestion focused at the host bridge, and following PCI bridges, connecting the victim GPU to the CPU and NUMA node. This is done by establishing some sort of communication between the victim device and the attacker device as shown in figure 3.5.

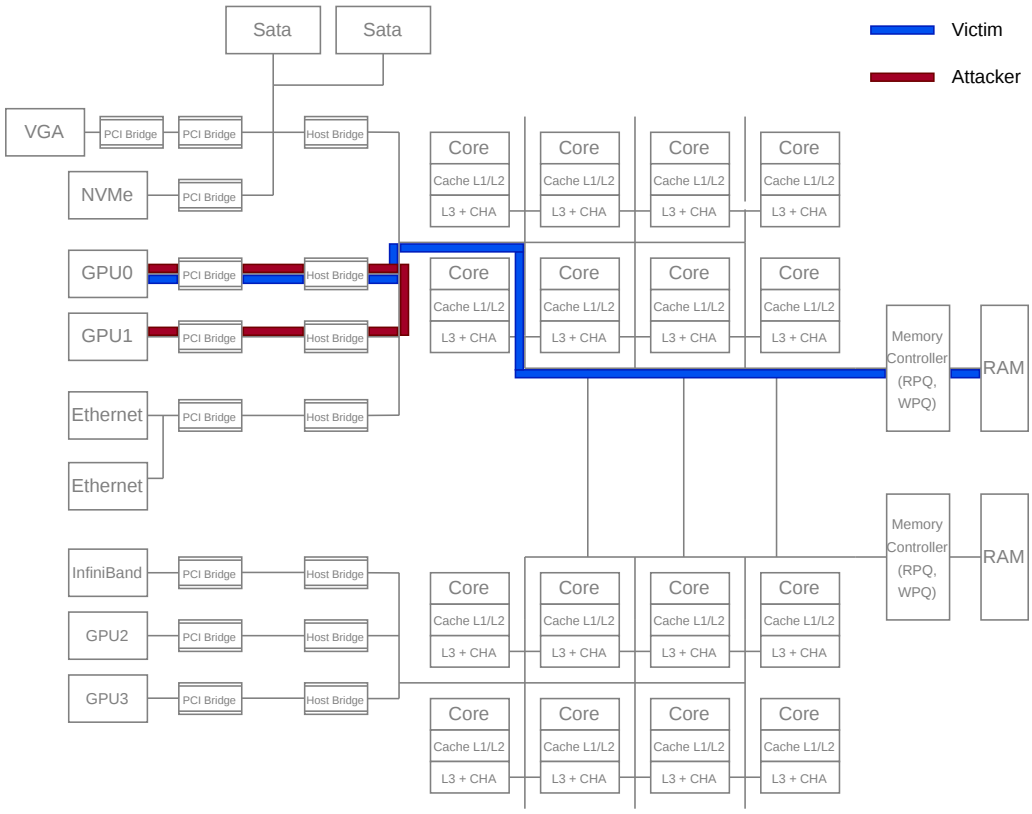


Figure 3.5: Travel path of victim data and attacker data for communication within same NUMA node

First, an attacker initializes a tensor filled with random values on the attacking GPU and an empty tensor on the victim GPU. The data on the filled tensor is copied to the empty tensor to generate traffic between the different GPU devices. This is tested in incrementally longer bursts to not risk blocking "heartbeat" signals to the

victim GPU and potentially signaling a theoretical admin. The attack can also be extended by running multiple threads or sending data from the victim GPU to the attacker GPU too.

3.5.4.2 Communication Between NUMA Nodes

This attack includes one victim and two (or more) attackers. The focus of this attack is to only target the host bridge connected to the victim, and not the following PCI bridges and data-busses. Data now travels from the attacking GPU, over the UPI bus between NUMA nodes, to the other attacking GPU without directly sending data to the victim GPU. See figure 3.6

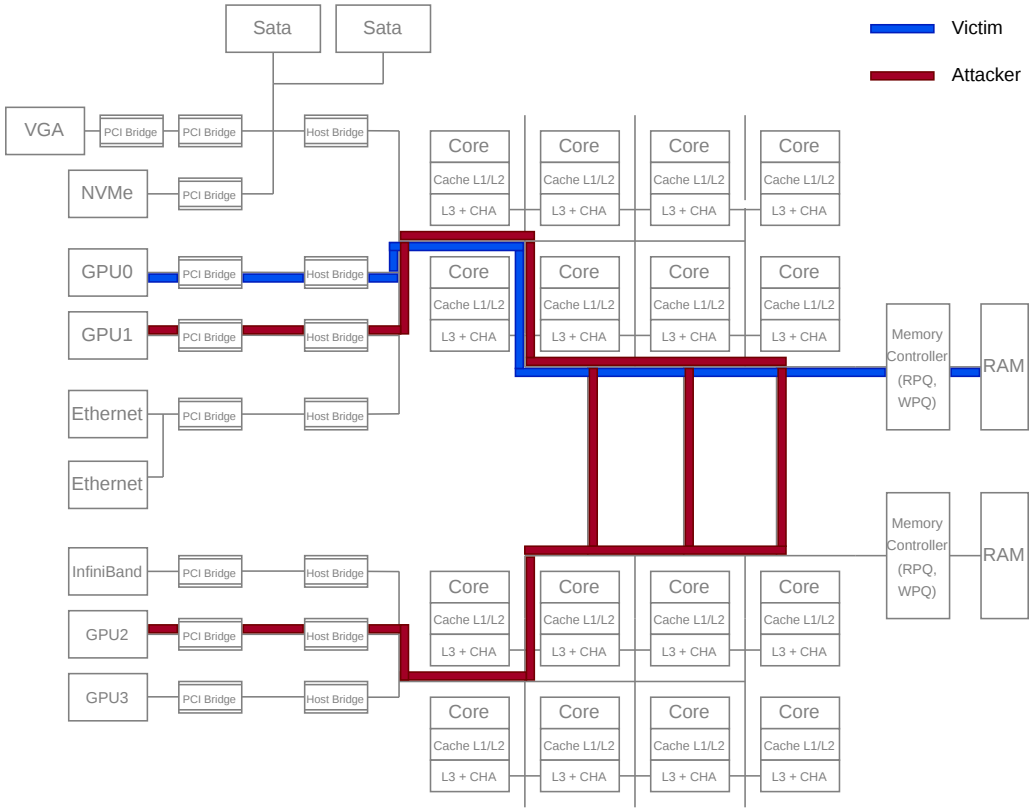


Figure 3.6: Travel path of victim data and attacker data for communication between NUMA node

The attacker initializes an empty tensor on the attacker GPU that runs on the same NUMA node as the victim. A tensor filled with random values is initialized on the attacker GPU running on the other NUMA node. The attack is run in the same way as for when communicating within the same NUMA node. The attack can also be extended by running it with multiple threads or running the attack with full-duplex.

3.5.5 Same Path Communication

The goal for this attack is to create congestion along the same path as the data for a victim model running on a GPU by assigning the victim and the attacking GPU to the same NUMA node. Any data sent to the memory on this node will share a travel path from their shared host bridge as displayed in figure . An empty tensor is created for the GPU attacker and a tensor with random values is pinned to the main memory on the NUMA node. The data is copied between the two in a full-duplex manner, executed using stream queue for consistent data requests.

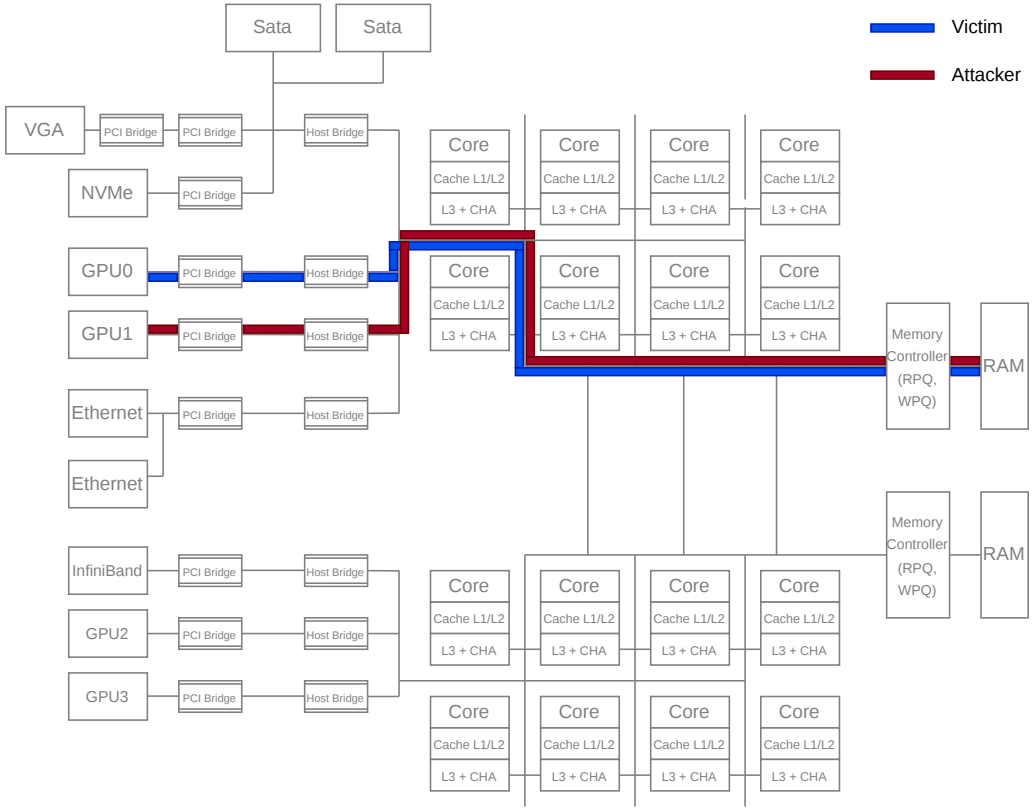


Figure 3.7: Travel path of victim data and attacker data for same path communication

3.6 Test Procedure and Evaluation

To quantify the host network degradation, the following procedure was used:

- **Idle baseline:** Before any workload was being executed, hardware statistics for the idle system were captured. This was done by running the sleep command and wrapping it within Perf.
- **Isolation baseline:** Next, the AI workload was executed in isolation on a node. This established a baseline throughput (e.g. tokens per second) and baseline hardware performance statistics.
- **Attack co-location:** The victim AI workload was then co-located with an attack using the Mio framework described in section 3.3. To enable synchronization, both the victim and attacker had to be launched within the same exclusive SLURM job. With this setup, any job-level Quality of Service (QoS) limits is bypassed, giving both the victim and attacker full access to all hardware resources.
- **Evaluation:** The main metrics for evaluation are AI model performance and time to serve a request (latency). The transformer's end-to-end performance is measured in tokens per second and the GNN's performance is measured in time (seconds) to train 5 epochs. The latency is determined by using Little's law [24]:

$$Latency\,(\#\,of\ cycles) = \frac{Occupancy}{Inserts} \tag{3.1}$$

  For example, to calculate the average latency of a read request from the CHA to the DRAM, the counter `unc_cha_tor_occupancy_ia_miss_drd_ddr` is divided by the counter `unc_cha_tor_inserts_ia_miss_drd_ddr`.

3.6.1 Performance Counters

Different monitored performance events and their respective performance counters were captured depending on which part of the host network was evaluated. Six different measurements were used for this project; CHA read, CHA write, RPQ, WPQ, IIO, and UPI. Examples of estimated event monitoring and counter placement are shown in figure 3.8.

It is important to mention that information and descriptions of performance counters are scarce. Consequently, the evaluation metrics used are our interpretations and understandings of the information about performance events and their use.

The CHA read counter captures the average latency for a main memory read request from the caching agent. This is done by monitoring the number of read requests and averaging over the total number of clock cycles that all requests have to wait to receive data. Similarly, the CHA write counter monitors the average latency of write requests by dividing the number of write requests by the total waiting time of all requests. The final results in cycles are divided by the CHA clock frequency to show the value in nanoseconds.

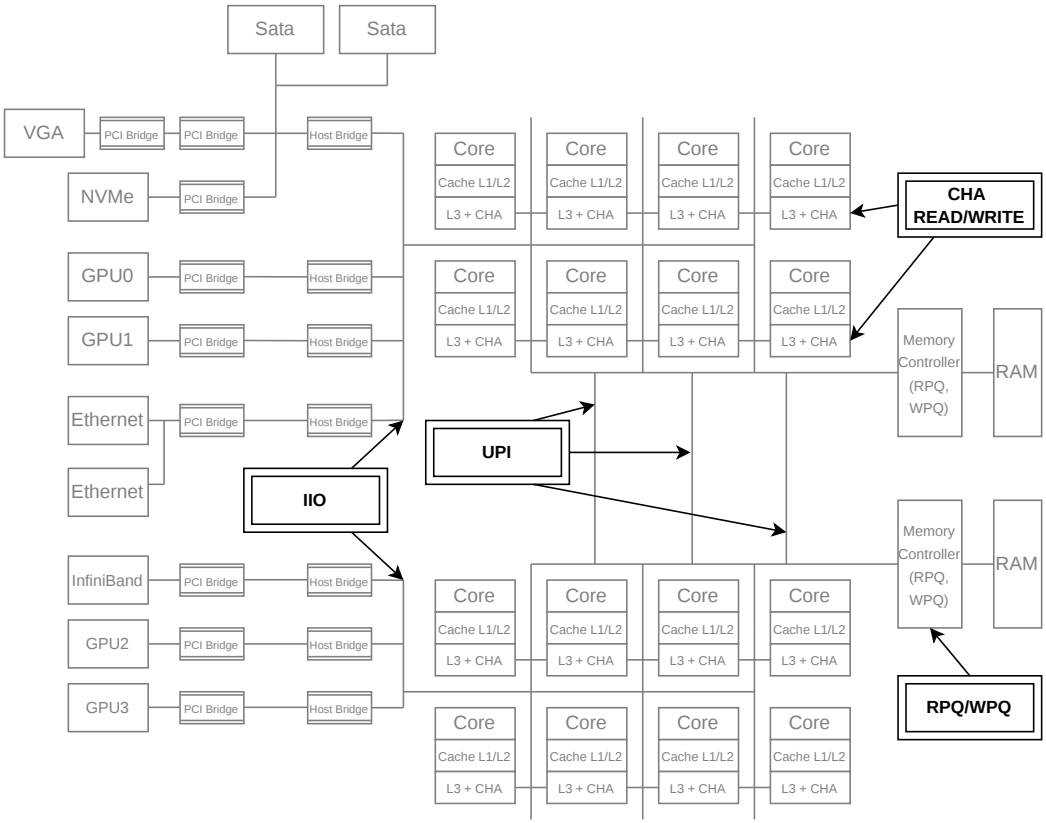


Figure 3.8: Estimated placements of performance counters on the Vera architecture

To calculate the RPQ latency, you can calculate the average queue depth by monitoring the total number of queue waiting cycles over the total number of requests. Each part is the sum of two pseudo-channels on the host network's DRAM. RPQ latency in clock cycles can also be converted to nanoseconds using the IMC clock frequency. The WPQ is instead calculated as the total active queue time. The total number of active cycles, where at least 1 request is in the queue, is averaged over the total number of cycles during the test.

IIO latency is measured by finding the average request processing time in the IIO completion buffer. This is done by dividing the total time in buffer by the total number of incoming requests to get the latency in cycles, and divide by the IIO clock frequency for the result in nanoseconds. This was only measured for the Vera cluster, as capturing one specific event on Minerva did not work despite showing in the list of available Perf events. An estimation of the inbound bandwidth was also determined by observing the events showing the total number of bytes in, which are incremented every 32 bytes.

The UPI counter is not used to calculate the direct latency by Little's law, but rather to monitor the total number of flits transmitted and received. Flits are the atomic pieces that form the data packages sent over the UPI link.

4

Results

This chapter presents results and analysis of the attacks executed against AI workloads. Rather than presenting raw data in isolation, the data is continuously analyzed to give possible explanations for the observed performance degradation. Finally, this chapter also documents negative results, i.e. attacks that were found to be ineffective.

4.1 Results for CPU-bound Transformer

In this model, the tensors reside in the CPU memory of Socket 0. The workload is distributed across GPUs split evenly among two sockets. Running this model creates Peripheral-to-Memory (P2M) traffic as the GPUs continuously read tensors from the CPU memory on socket 0 and writes back the update tensor. For the GPUs on socket 1, this P2M traffic crosses the UPI inter-socket links. During evaluations, both the memory and UPI attacks reside on socket 0 and target memory on socket 0 and socket 1 respectively. Each experimental configuration was executed twice per hardware counter set and then averaged. There are six sets of counters, so the end-to-end throughput presented in these results is the average of those 12 runs.

As seen in figure 4.1, this model is vulnerable to both the memory attack and the UPI attack across both hardware topologies. On the Vera node, the model achieves a baseline performance of 267515 tokens per second. However, when the 16 core memory attack is running simultaneously, throughput drops with ∼58% to 113614 tokens/s. The inter-socket UPI attack results in a smaller but still severe degradation of ∼40% down to 161204 tokens/s. When running the same tests on a Minerva node there is a smaller performance degradation. The maximum degradation on Minerva was ∼37% during the memory attack and ∼19% during the UPI attack. Since the model depends on the CPU memory, Minerva's DDR5 memory (compared to DDR4 on Vera) gives it an advantage. DDR5 provides higher bandwidth and a more efficient queue architecture, making it more difficult to saturate.

The two attacks target different parts of the host network. The memory attack targets the local memory at Socket 0. This can be seen in figure 4.2 which shows that the Read Pending Queue (RPQ) latency in socket 0 is affected by the Memory attack on Vera. When the time spent in the RPQ increases, every request to read a tensor becomes more time-consuming. Since the GPUs rely on continuously reading

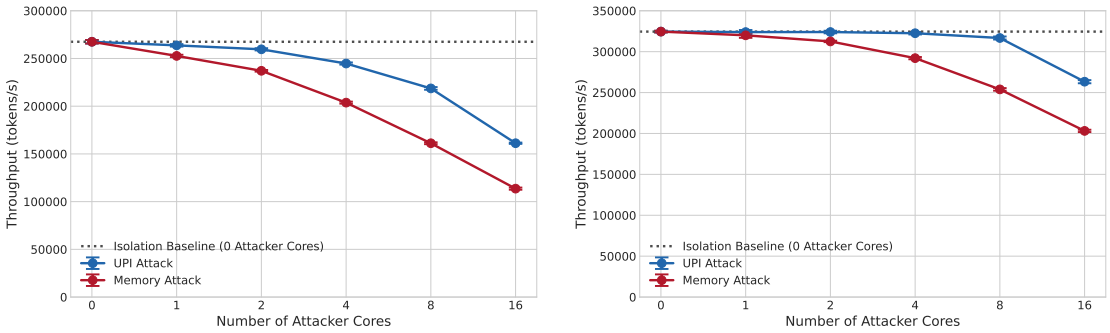


Figure 4.1: End-to-end throughput degradation of the CPU-bound Transformer on a Vera 4xA40 GPU node (a) and Minerva 8xL4 GPU node (b). The memory attack causes the most severe throughput degradation, but the UPI attack is also effective. Across both attack vectors, Minerva is more resilient. (Error bars: ±1 standard deviation over 12 runs)

tensors from the memory at Socket 0 (which is now flooded by the attacker), this inflated access time becomes a bottleneck.

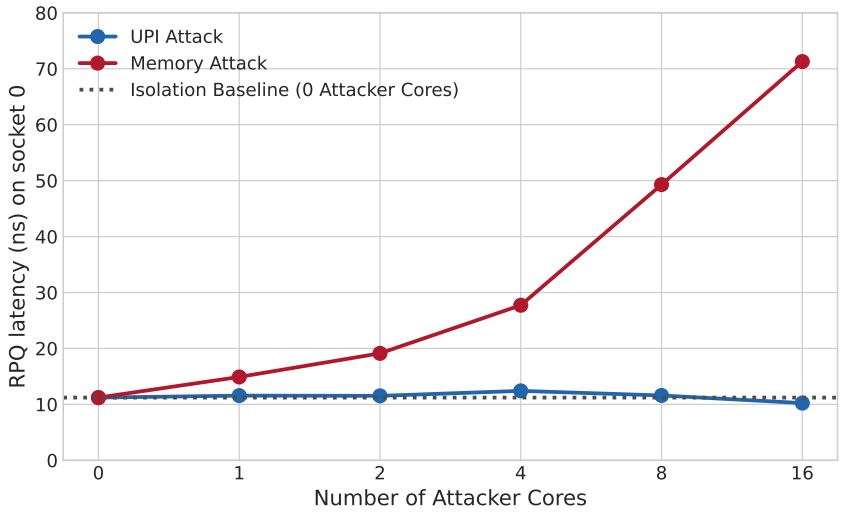


Figure 4.2: RPQ latency on socket 0 during attacks against the CPU-bound Transformer on a Vera 4xA40 GPU node. The increase during the memory attack indicates queue congestion, whereas the latency during the UPI attack remains flats since it targets the memory on socket 1.

Since the RPQ latency of socket 0 is unaffected by the UPI attack, that performance degradation must be caused somewhere else in the host network. One latency measurement that shows a big difference between the memory and UPI attacks is the CHA write latency for socket 0, see figure 4.3. It measures the time to write data back to memory from the CPU cores. During the 16 core memory attack it goes to ∼70 ns, compared to ∼31 ns without an attacker on Vera. When the 16 core UPI attack is running, it reaches ∼266 ns.

The reason for this difference is that the attacker's C2M traffic now passes from socket 0 to the memory on socket 1. Instead of just going from the socket 0 CHA to the memory controller, it goes through this path: S0 CHA → S0 UPI Interface → UPI Link → S1 UPI Interface → S1 CHA → S1 Memory Controller. By flooding this path, the attacker exhausts the flow-control credits assigned to the UPI domain. When the credits are exhausted, backpressure is applied to the socket 0 CHA which gets it's queue filled up. Consequently, the legitimate traffic between the GPUs on socket 1 and memory on socket 0 is throttled by flow-control credit starvation along the inter-socket path.

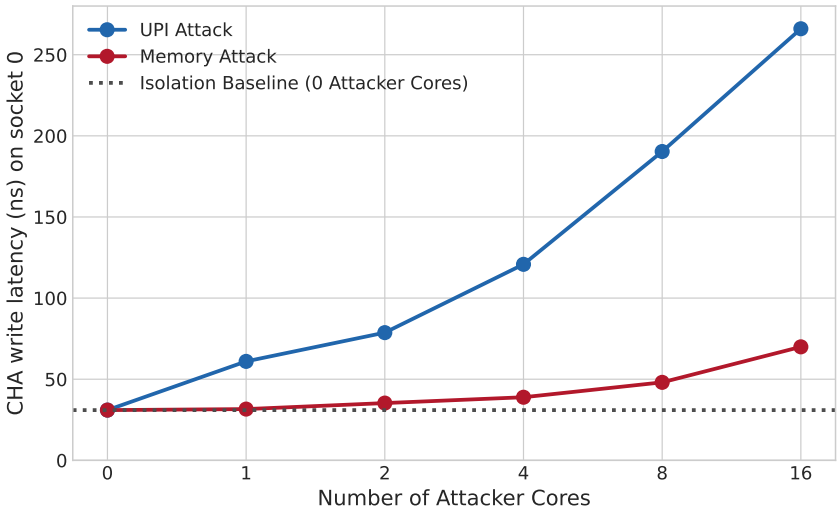


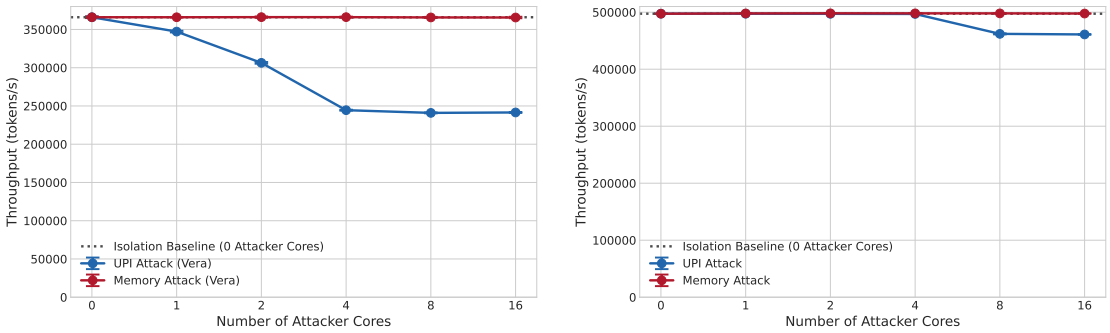
Figure 4.3: CHA Write latency on socket 0 during attacks against CPU-bound Transformer on a Vera 4xA40 GPU node. The higher latency during the UPI attack indicates that the inter-socket link becomes a bottleneck when the attacker (on socket 0) floods the memory on socket 1.

4.2 Results for GPU-GPU Transformer

In this model, the tensors reside on the GPUs. The workload is distributed evenly across GPUs on two sockets. Tensors are synchronized directly between GPUs in a peer-to-peer manner. This is done using Nvidia's NCCL library. This model will mainly create traffic directly between GPUs. When GPUs reside on different sockets, this traffic needs to cross the inter-socket links. During evaluations, both the memory and UPI attacks reside on socket 0 and target memory on socket 0 and socket 1 respectively. Each experimental configuration was executed twice per hardware counter set and then averaged. There are six sets of counters, so the end-to-end throughput presented in these results is the average over 12 runs.

Figure 4.4 shows the impact of the Memory attack and UPI attack on this model. The GPU-GPU Transformer model is vulnerable to the UPI attack, but throughput is unaffected by the memory attack. The model's baseline performance is 365983 or 497381 tokens per second without an attacker on Vera and Minerva respectively. When the UPI attack is executed with 4 cores on Vera, the throughput dropped with

∼33% to 244491 tokens/s. After 4 attacker cores, the throughput stays flat. The degradation on Minerva is smaller (∼7%), and it is unaffected by UPI attacks using less than 8 cores.



(a) (b)

Figure 4.4: End-to-end throughput degradation of the GPU-GPU Transformer on a Vera 4xA40 GPU node (a) and Minerva 8xL4 GPU node (b). This result demonstrates that keeping data in the GPUs' VRAM makes the model immune to the memory attack, but the UPI attack is still a vulnerability. Furthermore, the Minerva node is more resilient against the UPI attack compared to Vera. (Error bars: ±1 standard deviation over 12 runs)

The unaffected throughput during the memory attack indicates that the GPU-GPU model is immune to an attacker trying to saturate the CPU memory. During the memory attack, RPQ latency on socket 0 increases ∼7.6x and ∼5x on Vera and Minerva respectively (see figure 4.5). Despite this bottleneck to access the CPU memory, the model's throughput remains flat. Because the tensors reside on the GPUs' VRAM and are synchronized peer-to-peer, they bypass the typical Peripheral-to-Memory (P2M) path. A saturated CPU memory therefore does not affect the model's performance.

The model is however vulnerable to the UPI attack. The attacker creates traffic between socket 0 and socket 1, which affects the synchronization traffic between GPUs residing on different sockets. Both the victim and attacker are forced into the same UPI link domain. This creates contention for the finite flow-control credits of the domain. When the attacker exhausts the credits, there will be head-of-line blocking which affects the victim workload. Figure 4.6 shows the UPI traffic for the Vera and Minerva nodes respectively. Minerva has ∼2.3x more UPI traffic than Vera during the 16 core UPI attack, but the effect on AI throughput is smaller than on Vera.

The different results for Vera and Minerva are likely caused by multiple factors. One such factor is that they have different UPI link specifications. Vera has 3 links of 11.2 GT/s each. Minerva on the other hand has 3 links with 20 GT/s each. With almost double the theoretical inter-socket bandwidth, more attack traffic is needed to saturate the link. Second, the GPU density on Minerva (8 vs. Vera's 4) changes the synchronization topology. More synchronization can be done locally on

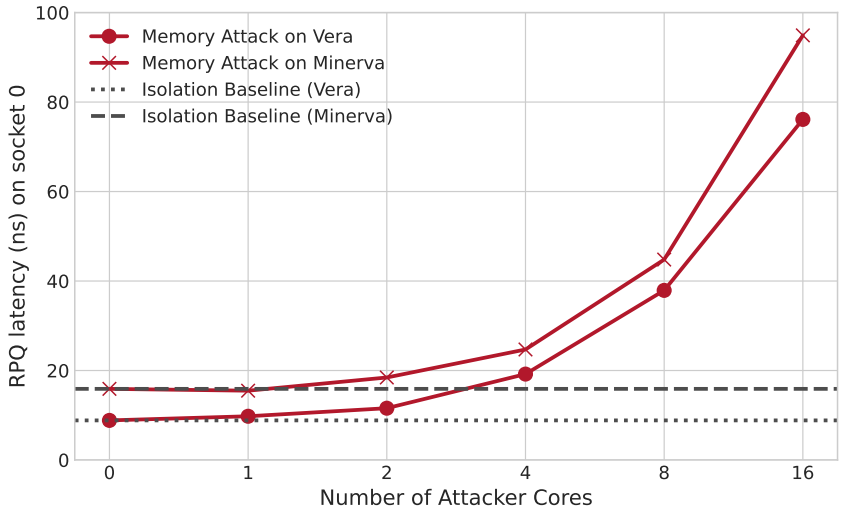


Figure 4.5: RPQ latency for GPU-GPU Transformer co-located with Memory attack on Vera and Minerva. This result confirms that the memory attack increases the read latency. However, since the data is synchronized peer-to-peer between GPUs, the model bypasses this CPU memory bottleneck.

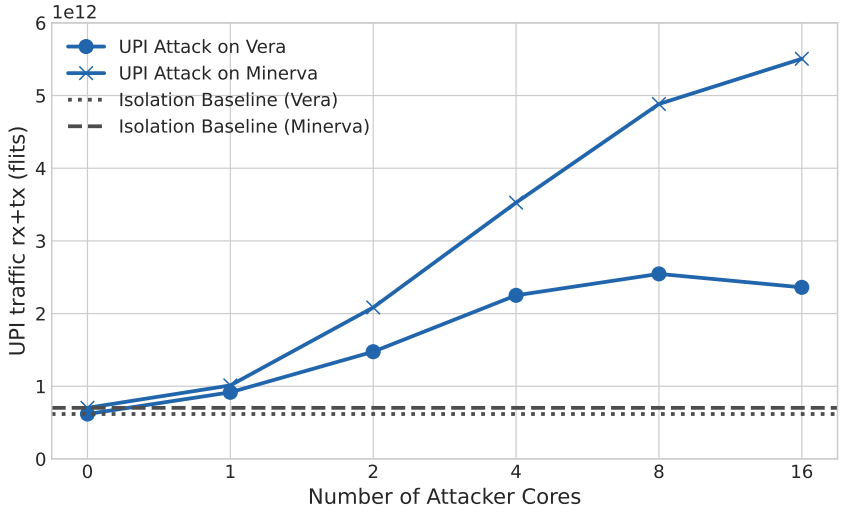


Figure 4.6: Total UPI traffic during 240 seconds of executing the GPU-GPU Transformer on Vera and Minerva co-located with UPI attack. This result shows that while the Minerva node handles ∼2.3x more UPI traffic, it experiences less end-to-end throughput degradation than Vera.

the same socket before sending data over the UPI link to the other socket. Lastly, Minerva's newer CPU generation is a factor. The newer generation has a more efficient mesh for internal routing and also DDR5 memory which yields a higher memory bandwidth. The individual impact of these factors cannot be perfectly isolated with this experimental setup, but combined they make Minerva more resilient against congestion attacks.

4.3 Results for Graph Neural Network

The results for this section will be presented as a comparison between the model running on a CPU and the models running on a GPU, resulting in testing both C2M and P2M traffic. The data in each figure show the result of a measurement with regard to one type of attack. Each model is running on NUMA node 0 with data stored in the memory positioned on this NUMA node, while any attacks run on NUMA node 1 with a maximum of 16 cores running the attack scripts simultaneously. All metrics and attacks were tested for 5 runs each, with the final result showing the average and standard deviation. The results presented with the figures and discussed in this section are what was deemed most interesting. For the remaining results, see the appendix A.

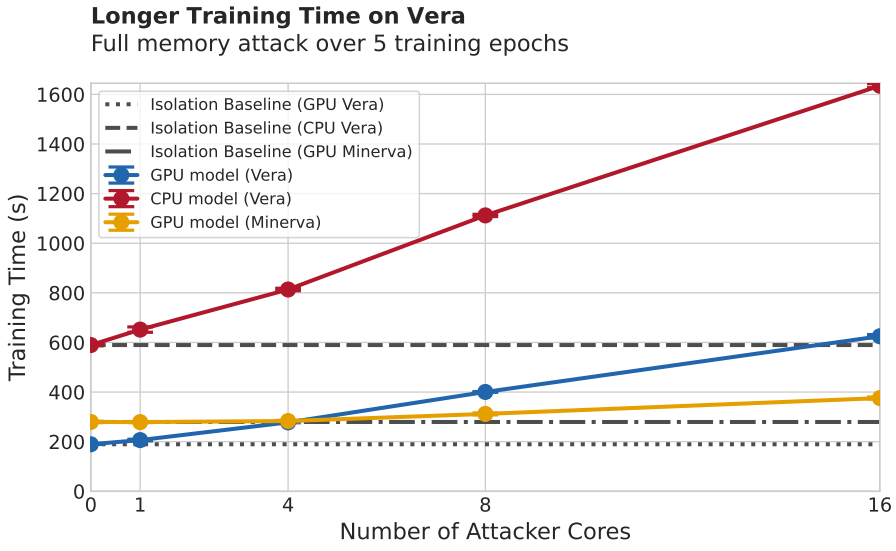
When studying the final training time results in figure 4.7, we find an expected result with the CPU model being, at full attack capacity, between 2.6x-3x slower than the GPU model running on Vera and 2.1x-4.35x slower than the Minerva-run model. This can be explained by the superior computing power for AI tasks of the present-day GPU device models. Also, figure 4.7a shows that the Vera cluster models are more vulnerable to full attacks with a steeper increase in training time compared to the Minerva-run model.

When comparing each model to itself, we find that in figure 4.7a the Vera-run GPU model is more sensitive to a full attack in comparison to the CPU model. The difference in training time between 0 attacker cores and 16 attacker cores shows a 3.3x increase for the GPU model, but only about a 2.8x increase for the CPU model (and around 1.35x increase for the Minerva model). Looking at the result in figure 4.7b, we find that the difference in latency for each model is more similar. The biggest increase is still 1.38x for the GPU model on Vera, and around a 1.33x increase for both the CPU model and the Minerva-run GPU model.

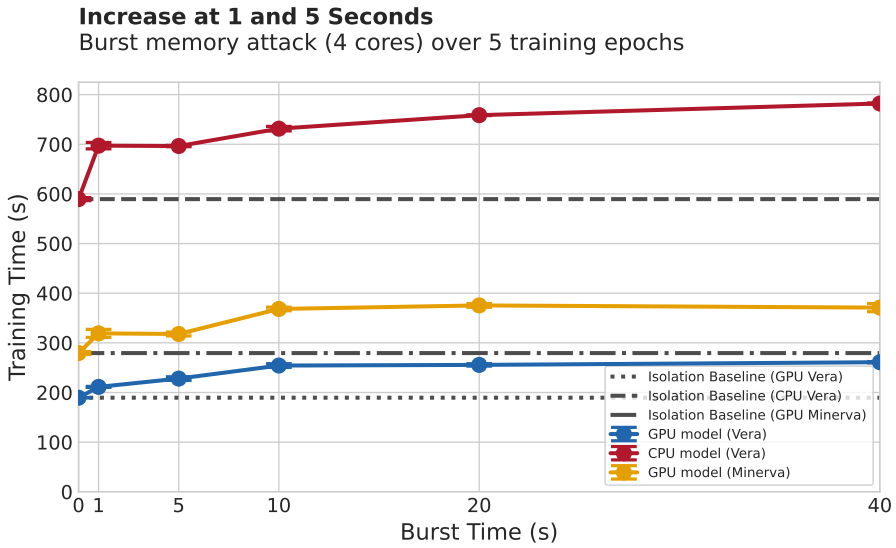
When investigating what was monitored by specific performance counters, there are results that might be connected to how the total training time, shown in figure 4.7, changes with the type of attack and scale of attack resources. As seen in both figures in 4.8, the Minerva-run model shows a higher latency than the Vera model when fewer attacker cores are used. From 4 attacker cores and forward, latency for both model setups on the Vera cluster exceed the Minerva model until it reaches around a 1.6 times difference as seen in figure 4.8a and 1.5 as in figure 4.8b.

We find that after using 4 attacker cores, a full attack is more effective at increasing the latency for the CHA unit on Vera compared to Minerva. We do not have an exact answer as to why, but one possibility is the difference in CPU architecture used for each cluster. As mentioned in section 4.2, the new intel generation CPU used for Minerva has a more efficient mesh for passing data. A potential higher bandwidth can make the Minerva-run model more resilient to handling more intensive congestion attacks.

However, both results for CHA read and write in figure 4.8 also contradict this when monitoring latency for the baseline compared to an adversary attacking. Figure 4.8b specifically shows that an attack never achieves the same latency delay as the



(a)

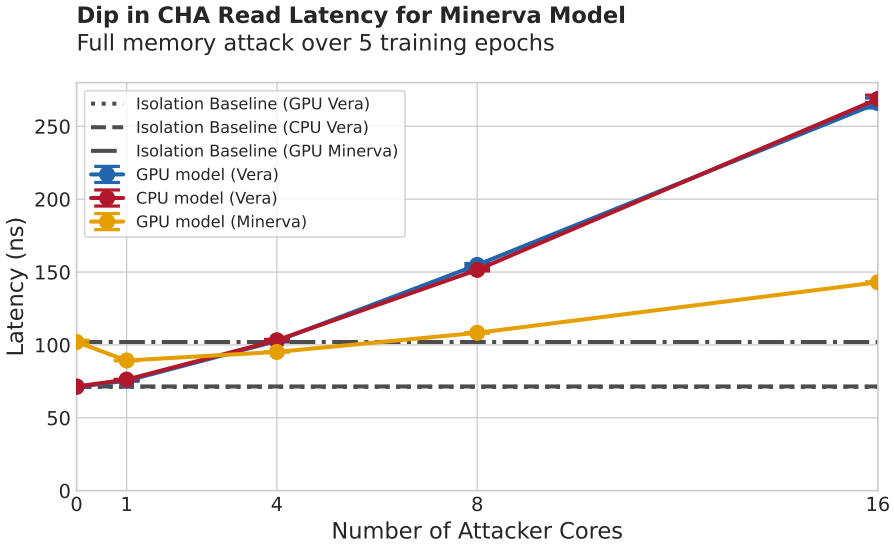


(b)

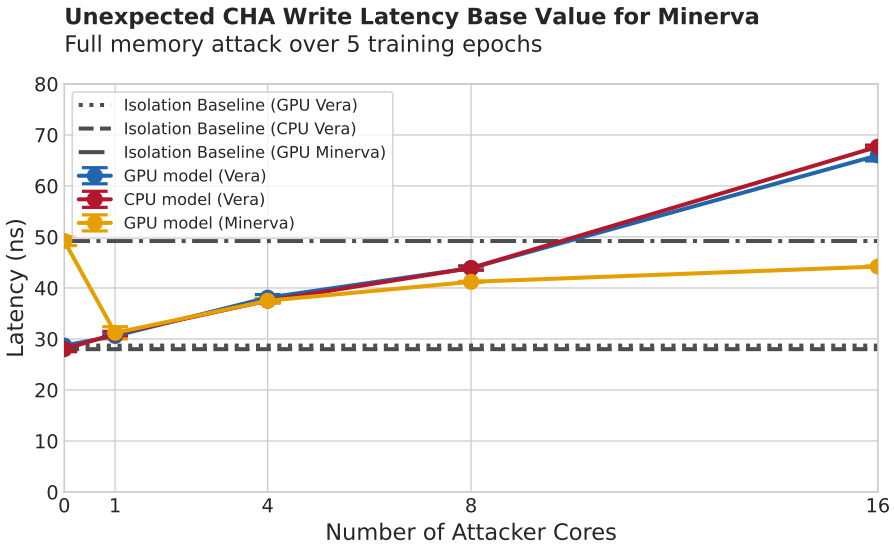
Figure 4.7: Training time increase over 5 epochs on the GNN model against a constant memory attack (a) and burst memory attack (b). The results show that the CPU-model is up to 4.35x slower than the GPU-models. Individually, the Vera-run GPU model is most vulnerable to an attack with up to 3.3x increase in training time.

baseline. A possible explanation for this behavior would be the background noise monitored by the performance counters. The mesh structure of the Minerva CPU, Intel Xeon Gold 6548N, is divided into two physical silicon blocks, where the cores and memory are placed. These communicate constantly and might be more active when the cores are idle. Another possibility is CHA "snooping". This is used to track cache coherency between cores and to notify other cores when a cache line is modified or evicted, which could be triggered when there are many different background processes running. It could also be noise from an OS daemon in connection with

running on a shared HPC cluster. A daemon running on the Minerva cluster might be more active when there is an idle NUMA node on a host, which is the case for monitoring the baseline, trying to serve other users.



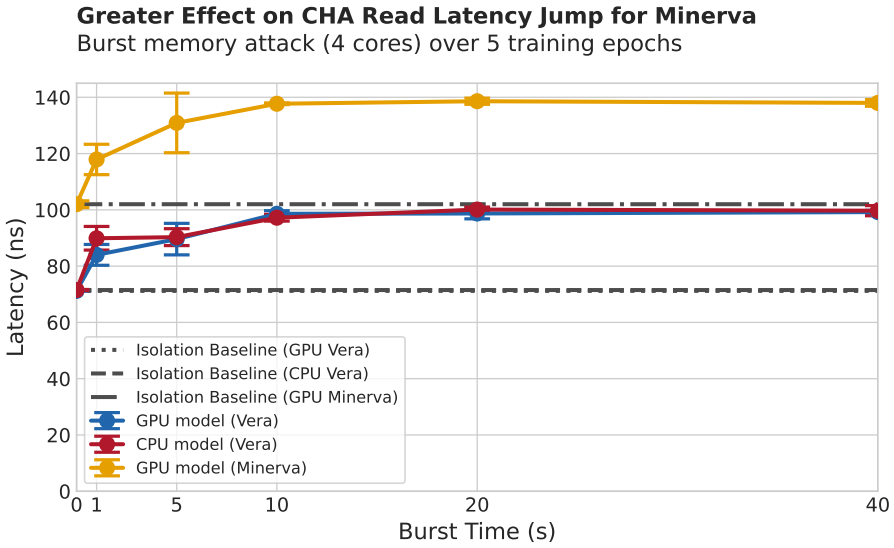
(a)



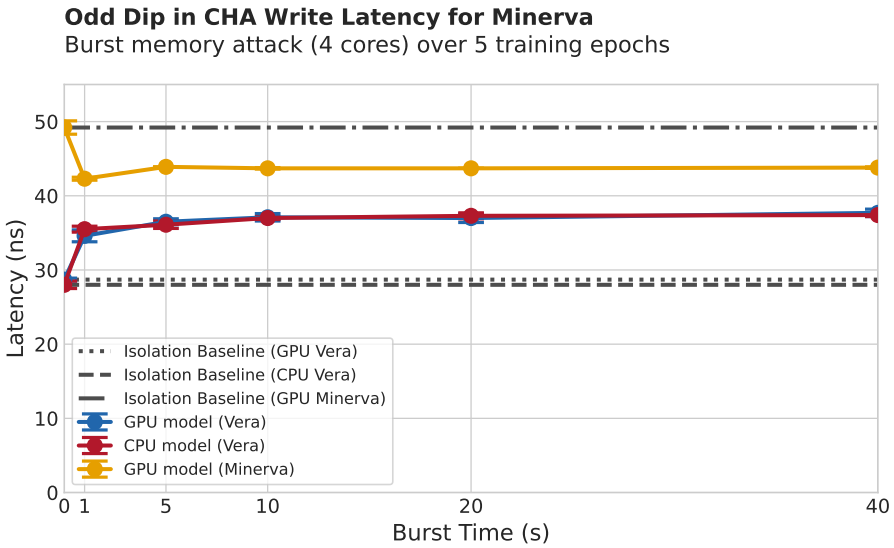
(b)

Figure 4.8: Latency of CHA read (a) and write (b) on the GNN model against a constant memory attack. The results shows that a full memory attack is more effective on the Vera cluster when using 4 or more cores. The baseline for the Minerva-run model show signs of being vulnerable to noise.

Examining the results in figure 4.9, we observe similar behavior for the CHA latency as seen in figure 4.7b. There are two distinct latency increases at 1 second burst attacks for both read and write, and then at 10 seconds for read. The 1 second burst attack shows that the CHA unit is instantaneously affected by an increase in workload. The result showing bursts of 10 seconds or longer is more interesting. As the attack is carried out using 4 attacker cores, and when compared to the latency when using 4 attacker cores in figure 4.8, the stagnation in attack effectiveness after 10 seconds informs us that this is the minimum needed attack resources for full congestion. The CHA unit appears to limit any additional data requests, possibly as a result of depleting the host network credits available when running 4 cores. This also demonstrates the resource limitations of an attacker. Additionally, the CHA write latency in figure 4.9b displays the same behavior as in figure 4.8b. In all likelihood, this is further information that shows that the performance counter monitoring the baseline is vulnerable to noise on the Minerva cluster.



(a)



(b)

Figure 4.9: Latency of CHA read (a) and write (b) on the GNN model against a burst memory attack. CHA limits further requests after 10 seconds burst attacks due to credit limitations. The CHA write baseline anomaly suggests that it is vulnerable to noise.

Examining the figure 4.10, we can observe that the RPQ of the memory controller also reaches full capacity after 10 seconds. This suggests a direct connection to the C2M read domain and the P2M read domain presented in section 2.1, showing that CHA and the memory controller share a credit-based sub-network. The higher latency for the Minerva-run model compared to the Vera-run models also indicates that the credit-based domains on Minerva operate using more credits. The RPQ latency, as described in section 3.6.1, is calculated as the average waiting time of a request. By allowing more requests, directly related to credits, it can show as higher

latency when monitoring the RPQ performance counters.

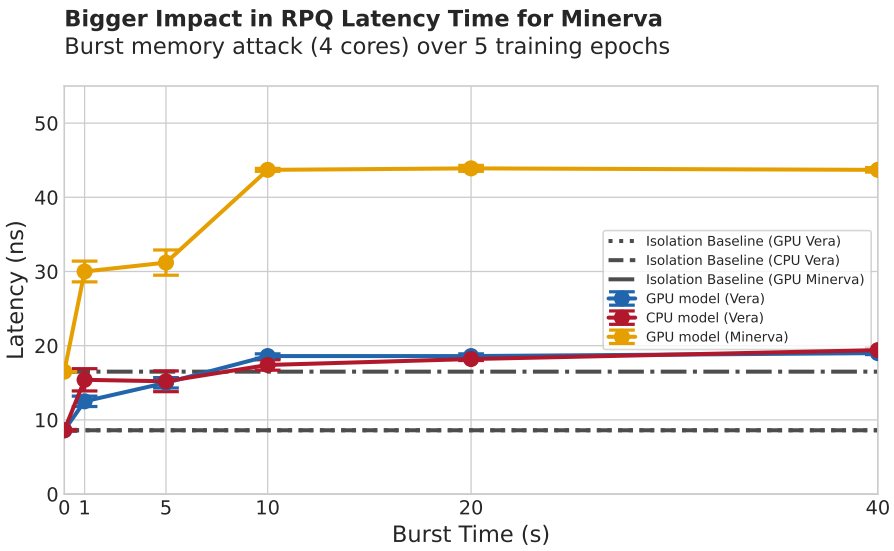


Figure 4.10: Latency of RPQ on the GNN model against a burst memory attack. Increase after 1 second bursts show that RPQ is affected by a congestion attack. Almost no change in latency after 10 seconds shows that max credits have been used

4.4 Unsuccessful Attack Tests

Some of the attacks mentioned in section 3.5 did not yield results due to various reasons. They are still included in this report as they are deemed to have the potential of being successful attacks. Below, we will discuss our conclusions as to why these did not work and if there are any scenarios where they could be used by an attacker.

4.4.1 Mesh Attack

This attack was not successful in causing performance degradation. The reason could be that the processor is designed with a mesh topology for the core interconnections. Data can travel vertically and horizontally between cores, in a distributed fashion. The attack simply is not able to generate enough traffic to saturate this kind of architecture. If the processor would have used a ring topology instead, the attack might have been effective. In the ring architecture, the cores share a single cyclic pathway which might have been easier to saturate. Another possible reason that this attack was ineffective is the fact that the Caching and Home Agent (CHA) and it's table of requests (TOR) is physically distributed across the cores. This means that the requests would only fill the TOR on the local CHA slices, and not affecting the AI workload being executed on other cores.

4.4.2 Communication Within Same NUMA Node

The goal of the first attack tested, to transfer data between the attacker and a tensor on the victim GPU running in parallel with the model, was to use and possible exhaust resources the model uses. In theory, the attack would compete over resources while the model is running a training epoch, or monopolize them during batch data transfers of the model. The reason why this test did not show results was due to the GPU devices running in Nvidia EXCLUSIVE_PROCESS-mode, restricting parallel processing. This is likely a security measure to ensure that the process can run in an isolated environment and is a good standard for working with share cluster networks. However, a GPU device can be set to run parallel processes, and thus is a potential vulnerability that a malicious actor can take advantage of.

4.4.3 Communication Between NUMA Nodes

When testing the attack that uses communication between attacker GPU devices on different NUMA nodes, we saw no significant impact on the victim. Further investigation found that this may not have been a problem with this specific type of attack, but rather an issue with the capabilities of the hardware. There is a significant difference in transfer speeds between the UPI and the PCIe used for the graphics cards. The UPI has a transfer speed of 11.2 GT/s for the Intel Xeon Gold 6338 processor with a maximum of 3 links, while the GPU uses PCIe Gen 4 x16 with a theoretical transfer rate of 16 GT/s per lane. This results in the GPU device being able to receive and handle any data from the other attacker device before the shared host bridge with the victim GPU devices is affected. In other circumstances, this could have the possibility of being a viable option for an adversary depending on the hardware.

4.4.4 Same Path Communication

This test, transferring data between the GPU device on the same NUMA node as the victim and the same node's memory, may not have shown any results in regards to how it affects the victim model, but showed that the host network had a possibly unintentional defense already. Most of the tests for this attack were terminated early by the Vera clusters watchdog for inefficient GPU utilization. According to the message notifying the user of the job termination, this was likely due to excessive copying of the tensor data and not distributing any of the computations over multiple GPUs. First, this shows that running some sort of watchdog for a cluster's resource utilization already provides a very good basic countermeasure for congestion control. Second, an adversary must be more aware of how to disguise an attack targeting this type of congestion path. The adversary has to study or monitor in real time the GPU usage of the other devices to hide any malicious intent.

4.5 Summary of results

Table 4.1 shows an overview of performance impact across AI workloads and attacks. The results indicate that workloads relying on CPU memory, such as the CPU-bound Transformer and the GNN model, are vulnerable to a memory flooding attack. The CPU-bound Transformer experiences up to 58% performance degradation when co-located with the memory attack. The saturated memory queues inflate the memory access latency, throttling the rate at which data can be sent to the GPUs.

Looking at individual performance counters of the GNN models, we see a common trend of increasing latency when running successful attacks. A clear exception to this is the Minerva-run model, seemingly susceptible to noise and possible background processes when calculating a baseline. When studying the overall training time results for the GNN models, we find that the CPU model runs up to 4.35x slower than the other models during an attack. However, it is the Vera-run GPU model that is the most vulnerable with a 3.3x training time increase when comparing the baseline to a full capacity attack. When comparing results from performance counters, the Minerva-run model often shows a higher latency than the Vera-run models. However, this is presumably the result of credit-based domains distributing more total credits to serve data requests.

The GPU-GPU Transformer shows immunity to the memory attack, by synchronizing between GPUs peer-to-peer instead of relying on CPU memory. It was however not immune to an attack targeting the UPI inter-socket link. The UPI attack resides on socket 0 but accesses the memory on socket 1 and thereby floods the UPI link. This path is also used to synchronize between GPUs on different sockets. The GPU-GPU Transformer suffered a throughput degradation of up to 33% on the Vera hardware. On Minerva, there was only a 7% performance degradation. This difference shows the benefits of Minerva's hardware: a higher UPI bandwidth, an 8 GPU topology, and a more recent processor generation with more efficient routing and DDR5 memory.

Finally, a few of our attacks were found to not be effective, see the last part of table 4.1. One of them is the mesh attack, which possibly was ineffective due to the high bandwidth and distributed nature of the CPU mesh. The other unsuccessful attempts, directly targeting GPUs, also failed to yield any performance degradation.

Table 4.1: Summary of results. Maximum impact is expressed as the percentage drop in throughput for the Transformer models and as the multiplied increase in training time for the Graph Neural Network.

AI Workload	**Attack Vector**	**Max Impact**	**Probable Cause**
CPU-bound Transformer	Memory	∼58%	Local Memory Saturation
CPU-bound Transformer	UPI	∼40%	UPI Link Saturation
GPU-GPU Transformer	Memory	0%	Bypassed local memory by communicating P2P
GPU-GPU Transformer	UPI	∼33%	UPI Link Saturation
Graph Neural Network	Memory	∼3.2x	Local Memory Saturation
Graph Neural Network	Memory (burst)	∼1.38x	Intermittent Memory Saturation
Unsuccessful Attack Vectors (Negative Results)			
Mesh Attack		0	Absorbed by high mesh bandwidth
Same NUMA Comm.		0	GPU and NCCL restrictions
Cross-NUMA GPU Comm.		0	PCIe and UPI transfer speed difference
Same-Path GPU Comm.		0	Cluster watchdog detection

5

Conclusion

This thesis investigated the question of whether a co-located adversary can make use of host network congestion to attack distributed AI workloads. The results clearly answer this question: a malicious actor can successfully create performance degradation by weaponizing shared resources in the host network. The effectiveness of such attacks however, depends on the victim workload.

Workloads that rely on the CPU memory, such as the CPU-bound Transformer, are vulnerable to a memory attack. Flooding and thus saturating the local memory was a very effective attack vector against this type of workload. The CPU memory becomes incapable of serving the GPUs with enough data to keep up the AI workload's throughput.

In contrast, the GPU-GPU Transformer showed immunity to this kind of local memory saturation. By keeping the data in the GPUs' VRAM and using peer-to-peer synchronization, this workload bypasses the saturated CPU memory. The immunity did however not apply to the inter-socket link. This link is used to communicate between GPUs on different sockets. An attack that floods the inter-socket UPI link can significantly degrade performance of the AI workload.

The graph neural network models showed different results depending on the type of attack and the host network architecture that runs the model. Often, the Minerva-run model was more vulnerable to lighter attacks, while instead having a lower latency increase when intensifying the attack compared to the Vera-run models. The total training time results show that the CPU model on the Vera cluster takes the longest. However, when comparing each model to itself, we find that it is actually the Vera-run GPU model that is the most vulnerable to attacks.

While this thesis demonstrates performance degradation specifically on the Vera and Minerva clusters, these findings suggest some broader implications for high performance computing (HPC) and multi-tenant cloud environments. The internal host network is currently treated as a trusted environment without any strict hardware Quality of Service (QoS) enforcement. This thesis indicates that an adversary doesn't need to compromise virtual machine boundaries or have high privileges to cause harm. Since the congestion attacks cause performance degradation rather than Denial of Service, they are likely more difficult to detect for system administrators. As AI models scale, GPUs and other hardware represent a huge investment. By exploiting shared interconnects and resources in the host network, an attacker can degrade

the performance of the host network and force the expensive hardware to not be fully utilized. With customers paying for GPUs on a per hour basis, undetected adversarial host network attacks cause prolonged training times and thus financial costs.

While this thesis shows some specific host network vulnerabilities, there are some limitations. First, the experiments were conducted exclusively on specific hardware, namely Intel Xeon processors and NVIDIA GPUs. The observed behavior of the host network and the performance degradation might not be applicable to hardware with other specifications and topologies. Second, there is inherent uncertainty when measuring micro-architectural events. The uncore counters are not strictly exact. Polling the counters creates overhead in addition to overhead from background tasks such as SLURM. Third, to achieve synchronization between the victim and attacker, they were co-located within the same SLURM job. Any job level QoS rules on the clusters were therefore bypassed and the victim and attacker shared the full allocation of hardware resources.

5.1 Future work

Since this thesis only looked at specific hardware setups, it would be valuable to test the attacks on different processors with other specifications. Attacks that failed in our tests might be effective on other systems and vice versa. Especially systems with direct links between GPUs (e.g. NVLink) would be interesting to evaluate.

Beyond single computers, the attacks could be expanded to larger clusters. AI can be trained on a large amount of interconnected machines, so triggering congestion across an inter-machine network would be a valuable extension of our work.

To help defend against attacks, future work might include developing software "watchdogs" that monitor tenants, to detect suspicious patterns. Those patterns could for example be abnormal queue latencies at memory controllers or UPI link saturation. A major challenge when developing such detection systems would be to minimize false alarms. The system must be able to distinguish a malicious actor creating lots of traffic from an AI training workload that simply needs a lot of resources.

Related to the unsuccessful attack "Communication within same NUMA node", other possible attacks to investigate could include exploiting NCCL by Nvidia. An attacker establishes communication directly with the victim model running on the victim GPU, where data is sent from the attacker's GPU memory directly to the victims GPU memory. Processes and respective devices that communicate are added in the beginning when creating a communicator, which means that an adversary process can try to disguise itself as a "friendly" process before the model process begins. A theoretical way to achieve this could be to eavesdrop on the broadcasting from the process creating the communicator. The attacker process could then try hijacking the initial connection request and be treated as a process that should be included.

Finally, exploring ways to stop or limit an attacker. This could possibly be done

by for example setting bandwidth limits for every tenant, e.g. with Intel's Memory Bandwidth Allocation (MBA). Since our experiment methodology bypassed any SLURM job level boundaries implemented on the clusters, a next step is to evaluate whether limiting bandwidths on a job or tenant level can mitigate congestion attacks. This would have to be designed in a way so that legitimate AI training tasks aren't affected. Finding a way to isolate an attacker without unintentionally hurting the performance of the victim remains a challenge.

Bibliography

[1] Z. Wang, J. Sim, E. Lim, and J. Zhao, "Enabling efficient large-scale deep learning training with cache coherent disaggregated memory systems," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 126–140. DOI: 10.1109/HPCA53966.2022.00018.

[2] M. Vuppalapati, S. Agarwal, H. Schuh, B. Kasikci, A. Krishnamurthy, and R. Agarwal, "Understanding the host network," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24, Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 581–594, ISBN: 9798400706141. DOI: 10.1145/3651890.3672271. [Online]. Available: https://doi.org/10.1145/3651890.3672271.

[3] Y. He et al., *Cross Container Attacks: The Bewildered eBPF on Clouds Cross Container Attacks: The Bewildered eBPF on Clouds*. 2023. [Online]. Available: https://www.usenix.org/system/files/usenixsecurity23-he.pdf.

[4] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018. DOI: 10.1007/s13389-016-0141-6. [Online]. Available: https://doi.org/10.1007/s13389-016-0141-6.

[5] Intel Corporation. "PerfMon Events Documentation." Accessed: 2026-03-19. [Online]. Available: https://perfmon-events.intel.com/.

[6] *Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual*, Document Number: 336274, Intel Corporation, 2017. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/671389/intel-xeon-processor-scalable-memory-family-uncore-performance-monitoring-reference-manual.html.

[7] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[8] Linux, *Perf: Linux profiling with performance counters*, Last update: 2024-12-01, 2024. [Online]. Available: https://perfwiki.github.io/main/.

[9] L. Foundation, *Apptainer user guide*. [Online]. Available: https://apptainer.org/docs/user/latest/index.html.

[10] M. Kerrisk, *Lscpu(1) - linux manual page*, 2026. [Online]. Available: https://man7.org/linux/man-pages/man1/lscpu.1.html.

[11] M. Kerrisk, *Numactl(8) - linux manual page*, 2026. [Online]. Available: https://man7.org/linux/man-pages/man8/numactl.8.html.

[12] M. Kerrisk, *Lspci(8) - linux manual page*, 2026. [Online]. Available: https://man7.org/linux/man-pages/man8/lspci.8.html.

[13] M. Kerrisk, $Ibv_devinfo(8) - linux manual page$, 2026. [Online]. Available: https://man7.org/linux/man-pages/man1/ibv_devinfo.1.html.

[14] NVIDIA, *Nvidia-smi - nvidia system management interface program*, Current version: v590. [Online]. Available: https://docs.nvidia.com/deploy/nvidia-smi/.

[15] Die.net, *Lstopo(1) - linux manual page*. [Online]. Available: https://linux.die.net/man/1/lstopo.

[16] F. Lee. "What is a neural networks?" IBM. [Online]. Available: https://www.ibm.com/think/topics/neural-networks.

[17] J. Noble. "What is a gnn (graph neural network)?" IBM. [Online]. Available: https://www.ibm.com/think/topics/graph-neural-network.

[18] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, 2021, https://distill.pub/2021/gnn-intro. DOI: 10.23915/distill.00033.

[19] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon et al., Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[20] C3SE and NAISS. "Chalmers centre for computational science and engineering."[Online]. Available: https://www.c3se.chalmers.se/.

[21] M. Karppa. "Minerva." Accessed: 2026-03-20. [Online]. Available: https://git.chalmers.se/karppa/minerva.

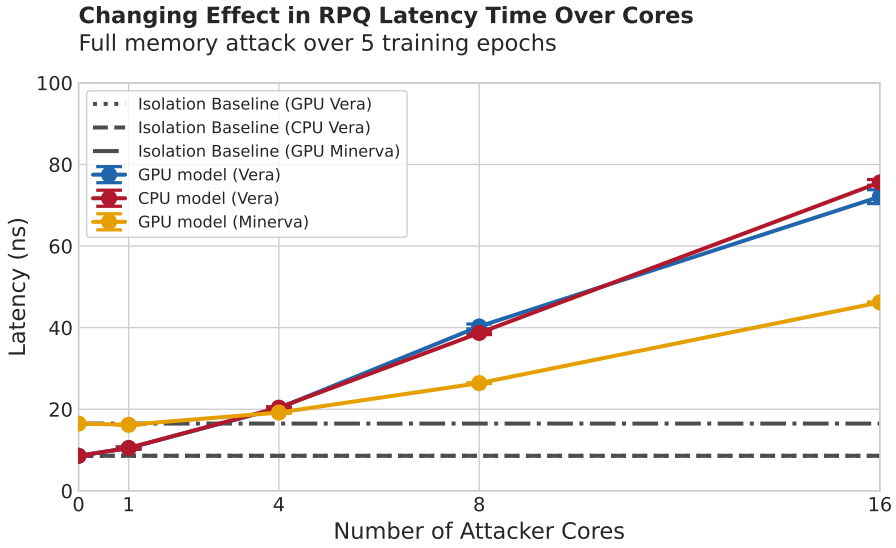
[22] NVIDIA Corporation, *NVIDIA Collective Communication Library (NCCL)*, https://github.com/NVIDIA/nccl, GitHub repository. Accessed: 2026-04-22, 2026.

[23] P. Team, *Pyg documentation*. [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/.

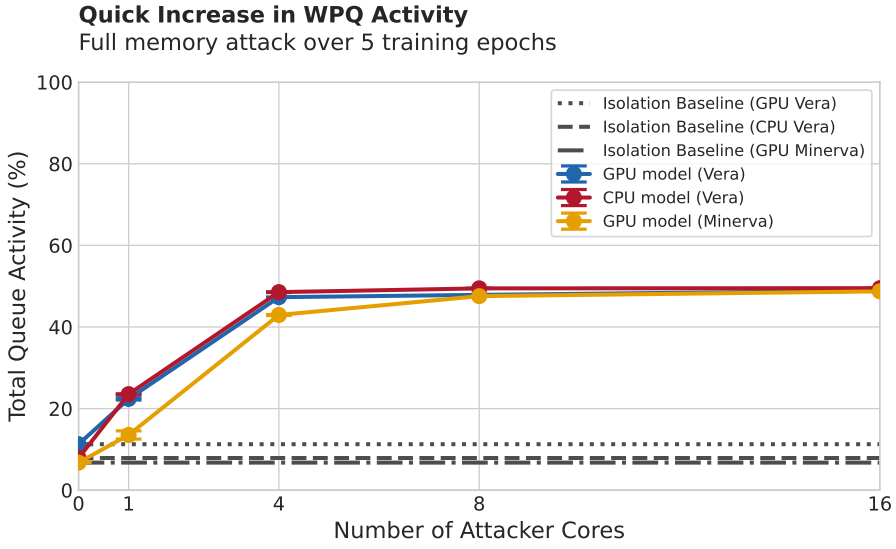
[24] Y. Z. Alexander Antonov. "Analyzing uncore perfmon events for use of intel data direct i/o technology in intel xeon processors (new)." From: Intel Vtune Profiler Cookbook, Intel. [Online]. Available: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2025-4/analyze-uncore-perfmon-events-for-intel-ddio.html.

A

Results: Graph Neural Network Model



(a)



(b)

Figure A.1: Latency of RPQ (a) and WPQ activity (b) on the GNN model against a constant memory attack

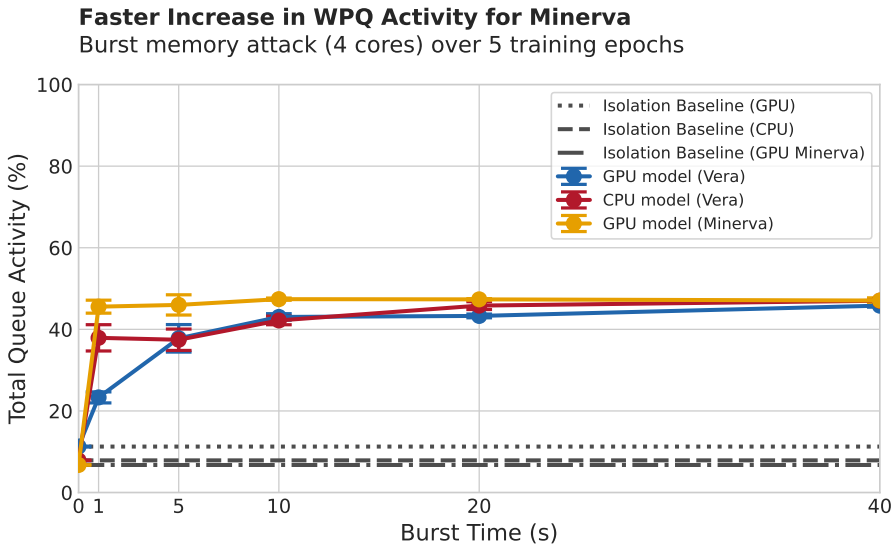
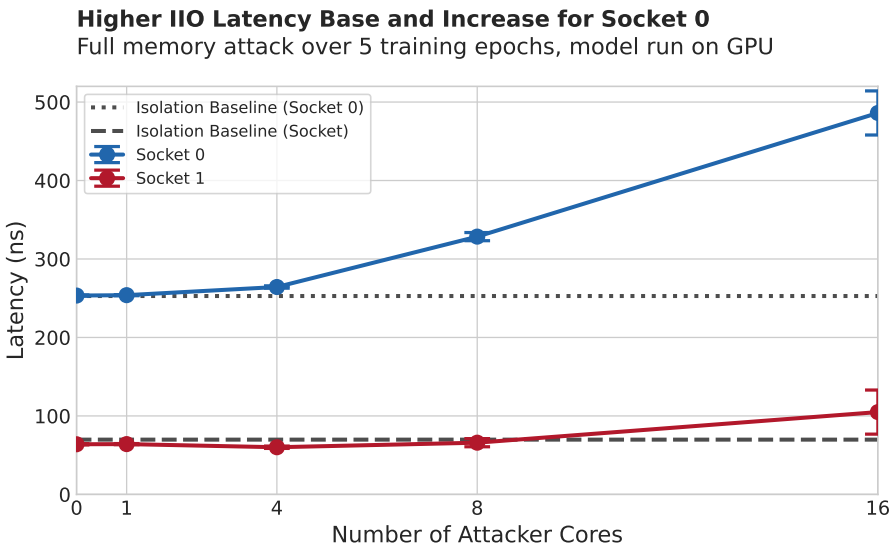
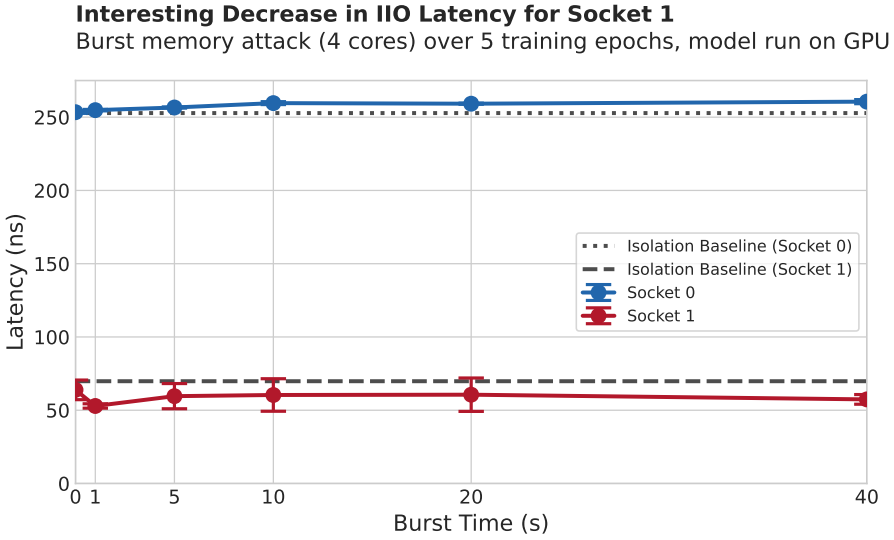


Figure A.2: WPQ activity on the GNN model against a burst memory attack



(a)



(b)

Figure A.3: Latency of IIO on the GNN model (Vera cluster) against a constant memory attack (a) and burst memory attack (b)