Building Massive MIMO Baseband Processing on a Single-Node Supercomputer

Xincheng Xie, Wentao Hou, Zerui Guo, Ming Liu

University of Wisconsin-Madison

Abstract

The rising deployment of massive MIMO coupled with the wide adoption of virtualized radio access networks (vRAN) poses an unprecedented computational demand on the baseband processing, hardly met by existing vRAN hardware substrates. The single-node supercomputer, an emerging computing platform, offers scalable computation and communication capabilities, making it a promising target to hold and run the baseband pipeline. However, realizing this is non-trivial due to the mismatch between (a) the diverse execution granularities and incongruent parallel degrees of different stages along the software processing pipeline and (b) the underlying evolving irregular hardware parallelism at runtime.

This paper closes the gap by designing and implementing **MegaStation**[1]–an application-platform co-designed system that effectively harnesses the computing power of a single-node supercomputer for processing massive MIMO baseband. Our key insight is that one can adjust the execution granularity and reconstruct the baseband processing pipeline on the fly based on the monitored hardware parallelism status. Inspired by dynamic instruction scheduling, MegaStation models the single-node supercomputer as a tightly coupled microprocessor and employs a scoreboarding-like algorithm to orchestrate "baseband processing" instructions over GPU-instantiated executors. Our evaluations using the GigaIO FabreX demonstrate that MegaStation achieves up to 66.2% lower tail frame processing latency and 4$\times$ higher throughput than state-of-the-art solutions. MegaStation is a scalable and adaptive solution that can meet today's vRAN requirements.

1 Introduction

Massive MIMO (multiple-in multiple-out)–equipping a base station with an array of radio antennas to serve many users simultaneously–has become the essential wireless technology in modern cellular networks such as 5G. It boosts spectral efficiency via multi-antenna techniques and increases network coverage, capacity, and throughput, which many 5G mid-band deployments have adopted [3–5, 12, 14, 38].

RAN virtualization has gained significant traction. It has been employed by many mobile network operators [11, 15, 35, 43, 50–53] and strongly advocated by industry players [36, 118]. It replaces specialized RAN hardware with general-purpose commodity servers or programmable accelerators for baseband signal processing, entailing a slew of benefits, such as mitigating vendor lock-in, offering better inter-operability and operational visibility, reducing the total cost of ownership, and improving network management agility.

However, realizing massive MIMO under vRAN is challenging due to the high computational demands and single-digit millisecond processing deadline of its baseband processing. It consists of a pipeline of non-trivial matrix manipulation tasks–like FFT/IFFT, equalization, and en(de)coding–and converts between radio bits and data packets for each antenna-user pair at every subcarrier frequency. The compounding effect of continuously massive MIMO dimensional scaling [63, 64, 117], the rising number of subcarrier frequencies, increasing radio units being deployed at a cell site, and stringent performance requirements [18] necessitate enormous execution parallelism and computing power, which are hardly met by today's vRAN computing substrates.

People have tackled this problem by either building rack-scale distributed baseband processing systems [71, 121] or developing specialized accelerator-based platforms [28, 58, 79, 100]. For example, Hydra [71] splits the frame processing pipelines [65] over multiple X86 servers and designs efficient synchronization to minimize the data shuffling overheads. When scaling, one has to use multiple servers, adding capital/operational costs and taking more physical space, which is a key concern for future Telco site infrastructures [6, 39]. LuMaMi [100] develops a bespoke testbed using 50 Xilinx Kintex-7 FPGAs and supports one modest-sized MIMO (100$\times$10) configuration. The system is not flexible and requires relaying out the software-defined radio array and partitioning symbol processing paths when updating the MIMO settings or accommodating user traffic vagaries. Therefore,

[1]MegaStation is available at https://github.com/netlab-wisconsin/MegaStation.

prior solutions struggle to provide adequate computing parallelism with flexible programmability at an acceptable cost.

Empowered by recent cluster interconnects (like Routable PCIe [2] and UALink [54]), infrastructure composability becomes possible, and an emerging computing platform–called **single-node supercomputer (SNC)** [44]–receives great attention and interest. It is a physically compacted testbed, comprising one server host and a composable GPU pool (enclosing tens of GPUs in standalone chassis), connected through backplane or external fabric switches, such as the GigaIO SuperNODE [44]. An SNC (a) holds substantial execution parallelism where remote GPUs behave as local ones and run host-native system stacks, (b) provisions adequate and consistent bandwidth for host-GPU and GPU-GPU communication, and (c) allows incrementally on-demand GPU scaling, making it a promising target for running highly parallel and compute-intensive applications, like AI/HPC.

In this paper, we design and implement **MegaStation** that harnesses the computing power of an SNC to process the baseband for vRAN-based massive MIMO. This is challenging because of the mismatch between the application software parallelism and the underlying hardware computing parallelism. The MIMO baseband processing pipeline comprises a sequence of stages that run at different execution granularities (such as antenna, user, and subcarrier) with incongruent parallel degrees. State-of-the-art solutions [65, 71, 100, 121] parallelize the pipeline at the frame, symbol, or task level. They are all suboptimal and would gradually make the underlying hardware parallelism irregular, adversely affecting the baseband processing performance (§3). Thus, without careful coordination and scheduling, one would experience radio bandwidth drops, frame processing deadline violations, and computing resource waste.

MegaStation addresses the challenge by revamping the baseband processing pipeline and applying techniques from the computer architecture field. *Our key idea is to model the single-node supercomputer as a tightly coupled microprocessor and employ a scoreboarding-like scheme [115] to schedule "baseband processing" instructions based on GPU hardware status.* MegaStation comprises four software modules: (a) *an instruction unit* that translates frame sequences to task instructions and analyzes their data dependency and structural hazards; (b) *processing function units*, which holds baseband processing instructions and runs over GPUs; (c) *the scoreboarding*, a centralized bookkeeper that continuously tracks the execution status of inflight instructions and performs resource accounting of the GPU pool, serving as the basis for other components; (d) *a pipeline scheduler*, orchestrating instruction execution via a new algorithm (called LROC) that sparingly synthesizes the least slack time scheduling with instruction reordering, over-commitment, and coalescing techniques. Therefore, MegaStation adjusts the execution granularity and reconstructs the baseband processing pipeline on the fly based on the dissected hardware parallelism status.

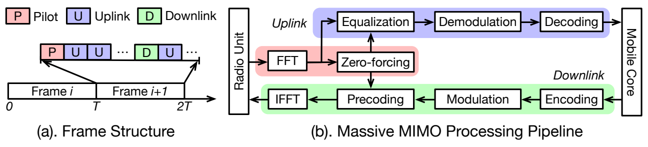


Figure 1: Frame structure and baseband processing pipeline.

We built MegaStation atop Agora [65] and Nvidia Aerial SDK [30] and evaluated it over the GigaIO FabreX platform. Compared with three state-of-the-art solutions [65, 71, 100, 121], MegaStation achieves up to 66.2% lower tail frame processing latency for the 256×128 MIMO setting. MegaStation effectively harnesses any available computing power of the GPU pool and schedules suitable baseband processing jobs without overloading the GPU. Further, we empirically demonstrate the scalability and adaptiveness of MegaStation under different vRAN settings. For example, our system can support 8 RUs under the 128×64 MIMO setting on 6 GPUs with up to $4\times$ higher throughput than other approaches.

2 Background and Motivation

We provide the necessary background about massive MIMO baseband processing, introduce the single-node supercomputer, and show its potential for vRAN acceleration.

2.1 Baseband Processing of Massive MIMO

In massive MIMO, the multi-antenna radio unit (RU) receives wireless signals combining multiple users' transmissions, where each antenna digitizes these signals into per-subcarrier (i.e., frequency) in-band and quadrature (IQ) samples and encapsulates these IQ samples as packets. The baseband processing unit (or BBU)–exchanging IQ sampled packets with an RU via a fronthaul link–performs a series of pipelined tasks to recover the bits from radios and then delivers them to the mobile core, and vice versa.

Take a typical $M \times K$ MIMO configuration as an example that serves K users concurrently via M antennas. Massive MIMO uses frequency canceling to support different users on the same frequency. We represent the signal streams transmitted by K user antennas for a given subcarrier as a $X_{K\times 1}$ vector. An RU then receives $Y_{M\times 1}$ signal, modeled as $Y_{M\times 1} = W_{M\times K} \times X_{K\times 1}$, where W is the computed precoder, i.e., $W_{i,j}$ capturing the wireless channel status between an antenna i and a user j. The baseband processing is responsible for converting between the signals Y from all RU antennas and the users' signals X. The processing granularity is a time frame (Figure 1-a), whose length depends on user and cell configuration, lasting up to a few milliseconds. Each frame is further divided into a few to tens of symbol durations (14 in a typical case). The beginning of a frame is a *Pilot* symbol that encodes the channel state information (CSI) and allows the BBU to estimate the channel matrix $W_{M\times K}$. The rest are *non-Pilot* data symbols, carrying one block of transmitted or received modulated data bits for all users on each subcarrier.

Processing Pipeline. Massive MIMO generally comprises an uplink and a downlink processing pipeline inside its BBU (Figure 1-b), each consisting of multiple stages. Take the uplink one as an example. First, it transforms the time-domain IQ samples into frequency-domain samples for each antenna through FFT. Second, the pipeline obtains the channel matrix (W) from the pilot symbol, and then uses the zero-forcing (ZF) technique [56, 123] to compute the precoder via the pseudo-inverse of the matrix W, i.e., $H = (W^*W)^{-1}W^*$. Third, for non-pilot symbols, the BBU performs equalization to recover user signals (X) by computing $H \times Y$. Fourth, it applies demodulation to extract information bits from complex numbers. Finally, the BBU performs decoding using a forward error correction (FEC) scheme to generate and verify the user bits. Akin to prior studies [65, 71], we use the low-density parity-check (LDPC) algorithm [70]. The downlink pipeline works in the opposite way, where each task performs the reverse functionality as the one in the uplink.

2.2 High Computing Demand under vRAN

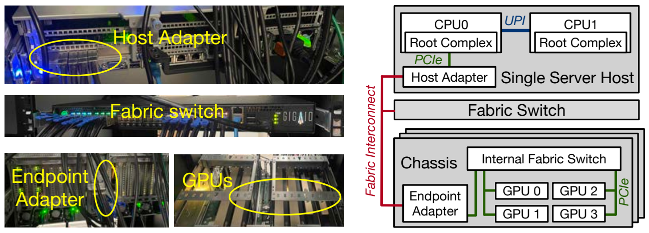
RAN virtualization realizes the BBU's functionalities using commodity servers or programmable engines instead of proprietary and specialized boxes [42, 46, 47, 49]. Even though this entails many benefits (e.g., cost efficiency, management agility, and operational visibility), vRAN drastically increases the computing demand, especially under massive MIMO.

There are three reasons. First, the baseband processing includes a pipeline of computing-intensive and highly parallel tasks (§2.1). For example, FFT and IFFT are parallelized at the antenna level, whose computing complexities depend on the number of subcarriers. Equalization, precoding, demodulation, and modulation tasks are subcarrier-parallel. Each requires non-trivial matrix calculations, and their dimensions depend on the number of subcarriers and users. The parallelism of en(de)coding hinges on the number of users. Such non-consistent parallel degrees indicate that data shuffling is required when crossing different execution stages [71].

Second, the MIMO is scaling up with an increasing number of antennas and users to satisfy future communication demands, such as XL-MIMO [63, 64, 117]. Third, cellular networks impose stringent requirements from data rate and latency perspectives. For example, only considering a bandwidth of 100MHz [18], the peak downlink/uplink spectral efficiency is 30/15 bps/Hz, translating to a maximum data rate of 3/1.5 Gbps. Further, 5G requires single-digit millisecond transmission latency between a user and a base station [17, 18] for two types of communication: ultra-reliable and low-latency communications (URLLC) and enhanced mobile broadband (eMBB). In the worst case (if an irrecoverable bit error happens), the BBU should send a downlink NACK to the user within four time slots (a few to 10s of milliseconds).

2.3 Prior Solutions

The de facto hardware substrate for deploying vRAN is commodity servers, starting from a pioneering effort–Sora [114].



(a). Hardward Testbed (b). System Architecture

Figure 2: The hardware testbed and system architecture of a single-node supercomputer based on the GigaIO's platform.

It processes the 802.11 a/b/g protocol using an Intel Core 2 processor (with SIMD extensions). BigStation [121], built atop Sora, realizes a distributed processing pipeline for multi-user MIMO (MU-MIMO) by parallelizing frames over multiple PCs. Regarding massive MIMO, Agora [65] is the first system that implements the baseband processing pipeline over an Intel Skylake processor by exploring data parallelism at the task level, supporting small-scaled massive MIMO (64×16). Hydra [71] scales up the massive MIMO capacity by distributing uplink/downlink pipelines to multiple X86 servers. It follows the BigStation architecture but designs efficient synchronization to minimize inter-stage data shuffling. Server-based solutions offer general-purpose computing and simplify software updates, but suffer from high TCO (total cost of ownership) and physical space constraints, especially when scaling the massive MIMO because more servers are required.

People also develop hardware-accelerated solutions to satisfy computing demands. Industry vendors build specialized BBUs (baseband processing units)–such as Nokia AirScale Baseband Module [47], Huawei BBU5900 [46], H3C BBU5200 [45]–and use ASIC DSP blocks to process the uplink and downlink pipelines. Atomix [58] takes programmable DSPs and builds a modular development framework for implementing wireless protocols and applications. FlexCore [79] parallelizes the detection of large numbers of mutually interfering information streams over GPUs. Intel's FlexRAN [28] provides a development SDK to build FPGA-accelerated vRAN stacks. LuMaMi [100] develops a specialized distributed FPGA testbed (connected via PCIe switches) and can handle 20 MHz bandwidth under a 100×10 sized MIMO. Albeit providing better energy/cost efficiency, such systems generally fall short in programmability, operability, and maintenance. For example, LuMaMi necessitates reconfiguring the FPGA array and partitioning the symbol handing path to accommodate MIMO and traffic changes.

2.4 Single-Node Supercomputer

Empowering by emerging load/store cluster interconnects [2, 40], infrastructure composability has become a new hardware architecture property [7–10, 13]. It transparently expands the server's computing capacity and improves the overall system cost efficiency. This yields a new and powerful computing platform called **Single-Node Supercomputer (SNC)**.

Take the GigaIO's FabreX and SuperNODE testbed [8] as

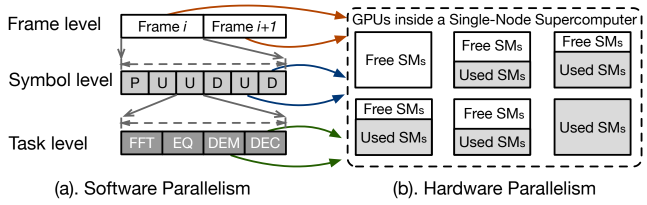


Figure 3: Three levels of parallel processing across the pipeline.

an example, which is our prototype and evaluation target. As shown in Figure 2-b, the platform consists of (1) host adapters (i.e., FA400X [23, 24]) that extend the host server's PCIe subtree; (2) standalone chassis boxes (i.e., RB4082 [33]), where each encloses endpoint adapters with up to 8 GPUs per appliance, connected through a PCIe backplane; (3) an external fabric switch (i.e., RS4024 [25, 34]), which connects host servers and chassis via SFF-8644 [37] copper/optical cables. In the current generation, the SuperNODE can attach up to 32 GPUs to a single server host. FabreX and SuperNODE are built atop routable PCIe 3.0/4.0 [2], and are under upgrading to other interconnects [40, 54]. Thus, a single-node supercomputer is one commodity server equipped with tens of GPUs, all of which stay under the server's PCIe root complexes.

The platform has several features. First, each remote GPU appears as a local PCIe device, which is directly accessed from host applications through existing driver and system stacks. This differs from today's distributed GPU clusters in the HPC and data centers, where a GPU access should traverse the message passing layer and networking stack [48, 68, 72, 76, 81, 99, 119], entailing communication overheads. Second, all GPUs are located under a multi-rooted tree topology inherited from PCIe, where host-GPU and GPU-GPU communications are provisioned with adequate bandwidth (e.g., $\times$16 lanes). Third, the platform can be scaled incrementally, where one can add/remove GPUs on demand without considering the available PCIe slots and power of the host server, simplifying infrastructure planning and management. Besides, the system is physically compacted (4U in our lab setup) with air cooling in the remote chassis. A single-node supercomputer resembles an NVIDIA DGX platform [31, 32]. However, it is loosely coupled physically with less GPU-GPU bandwidth (compared with NVLink 3.0/4.0) but has a higher GPU consolidation ratio and better infrastructure flexibility.

We believe that SNC is a promising vRAN substrate, especially when handling the dynamic high computing demand of massive MIMO. This is because (a) it encloses high computing density (GPU #) without bandwidth oversubscription in a single server, alleviating the need to handle the cross-node communication complexity in distributed systems; (b) it is a physically compact setup and has better space efficiency when scaling than today's rack-scale vRAN infrastructures; (c) GPUs can be added/removed on-demand flexibly via the PCIe hotplug without changing the underlying computing setting, whose management cost is less than dealing with the addition and removal of standalone GPU servers.

3 Understanding the Execution Parallelism

To effectively use SNC for baseband processing, this section characterizes the impact of different execution parallelism.

3.1 Types of Software and Hardware Parallelism

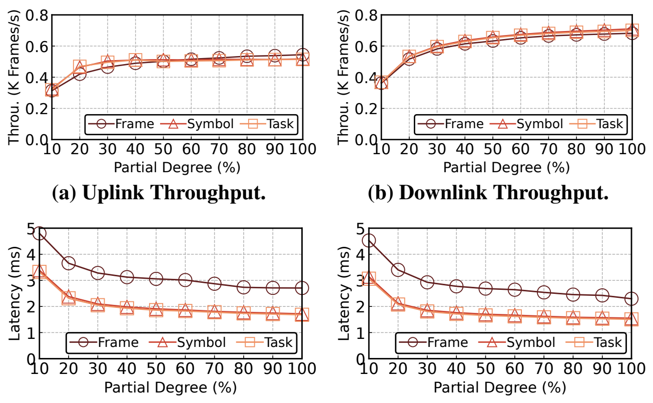
SW Parallelism. As described in §2.1, the baseband processing pipeline encloses a sequence of stages that exhibit different execution granularities and incongruent parallel degrees. There are three ways to parallelize it (Figure 3).

- *Frame Level.* The first is a coarse-grained approach that concurrently processes a window of consecutive frames, which has been explored in LuMaMi [100]. Since the computing intensity is inconsistent across different stages, it is challenging to do resource provisioning;
- *Symbol Level.* The second one is to parallelize symbol processing within a frame, such as Hydra [71]. Except for the pilot symbol (§2.1), other symbol handling can happen concomitantly. It is finer-grained than the first case but requires tracking execution dependency at runtime;
- *Task Level.* The third strategy is to schedule computations at the finest granularity (like BigStation [121]) that harnesses the available parallelism at best. We define a task as a MIMO processing stage as in Figure 1. However, this approach complicates the pipeline design because one should not only track dependent tasks, but also introduce the synchronization barrier (through shuffle) when the parallel degree across different phases changes, e.g., FFT (antenna#) $\rightarrow$ EQ/DEM (subcarrier#) $\rightarrow$ DEC (user#).

HW Parallelism. We quantify the available parallelism as the number of GPU's streaming processors (SM), denoted as P. These SMs could come from one or several GPUs. We then represent the adequate parallelism (SM #) to run a job in our baseband processing as Q. Thus, we categorize the hardware parallelism scenarios as follows:

- *Full Parallelism*, where $P \geq Q$ and $P\ SMs \in 1\ GPU$. The job can be placed in one GPU with no performance loss;
- *Fragmented Parallelism*, where $P \geq Q$ and $P\ SMs \in x\ GPUs$ and $x > 1$. Even though there are enough parallel engines, the job needs to be partitioned across several GPUs, and data shuffle would be required for synchronization when crossing different stages, hurting performance;
- *Partial Parallelism*, where $0 < P < Q$ and $P\ SMs \in x\ GPUs$ and $x \geq 1$. There is inadequate parallelism to run the job and one would see performance interference due to GPU sharing. The problem becomes even worse when the available SMs are from different GPUs;
- *Delayed Parallelism*, where $P = 0$. This indicates all SMs are busy. A job has to be queued first and be awakened when one of the above three types of parallelism emerges.

The Dilemma. Ideally, we should devise an execution strategy that always makes the baseband processing job run in full parallelism mode. To satisfy the stringent performance



(a) **Uplink Throughput.** (b) **Downlink Throughput.**

(c) **Uplink Latency.** (d) **Downlink Latency.**

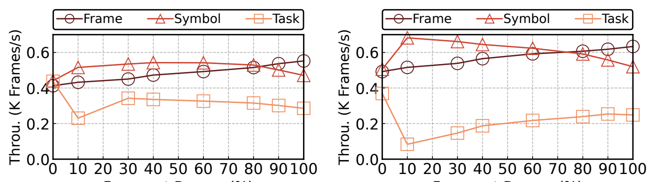
Figure 4: Baseband processing performance at three execution granularities varying the partial degree. We report the frame processing latency and throughput in all three cases.

requirements and high computing demand of massive MIMO, one should fully exploit the parallelism across three levels, i.e., parallelizing frame, symbol, and task processing as much as possible. As a result, the irregular software parallelism will gradually make fragmented, partial, and delayed hardware parallelism appear frequently, adversely impacting pipeline performance. We aim to characterize this dilemma and understand how different parallel executions perform under the above hardware parallel scenarios. Our experimental setup is described in §5.1 and the MIMO configuration is 128×32.

3.2 Partial Parallelism

Setup. We configure the partial parallelism by co-locating the target baseband processing task with a persistently running background job (which spawns a number of busy polling threads to consume the SM resources). We vary the number of occupied SMs and explore how it impacts three execution granularities. We define the partial degree as $\frac{P}{Q}$.

Results. The three software parallelisms can tolerate a modest to high partial degree. As shown in Figure 4, when half of the SMs are occupied, we observe up to 7.3% performance drop in all three cases. This is because the parallelism of the baseband processing pipeline is determined by its most computing-intensive phase. Other tasks requiring fewer SMs can still run in full parallelism when the partial degree is high. Thus, the performance-affected parts due to inadequate SMs would only contribute to a small portion of the total execution according to Amdahl's Law. Symbol/Task-level parallelism outperforms the frame-level one by 56.9%/60.4% in terms of uplink/downlink frame processing latency on average across all cases. This is because finer-grained execution parallelizes different symbols and tasks within a frame over multiple SMs. In contrast, frame-level execution cannot fully utilize the GPU for a single frame, resulting in a higher per-frame latency but allowing more frames to be scheduled concomitantly. As a result, all three software parallelism methods achieve similar throughput. We further explore an extreme scenario that only configures 1 SM (1.0% partial degree). Regardless of running



(a) **Uplink.** (b) **Downlink.**

Figure 5: Baseband processing perf. at three granularities varying the fragment degree. We measure the per-(frame | symbol | task) throughput (K frames per second).

at which granularity, the performance is reduced by more than $10\times$, indicating that a heavily-loaded GPU can easily violate the performance requirements of baseband processing.

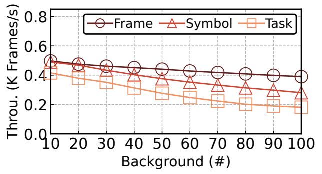
Takeaways. Partial parallelism with modest/high partial degree exhibits marginal performance degradation. However, as shown in Figure 4, the baseband processing performance drops significantly if a GPU has less than 20.0% free SMs. Therefore, one should carefully estimate the partial degree before allocating tasks to GPUs. Even though all three software parallelism options deliver similar throughput, frame-level parallelism presents the worst processing latency, making it ill-suited for an execution scenario with stringent deadlines.

3.3 Fragmented Parallelism

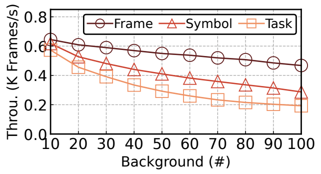
Setup. We provision required parallelism (Q SMs) from two composable GPUs and define the ratio of two GPUs' SMs as the *fragment degree*, i.e., $\frac{GPU1\ SM\#}{GPU2\ SM\#}$. The number of SMs is controlled via the same synthetic background job used in the partial parallelism section. We then examine how different execution granularities of the baseband processing behave when gradually varying the fragment degree.

Results. At the frame level, the pipeline proportionally schedules frames to two GPUs based on the number of SMs in a round-robin fashion. As shown in Figure 5, frame parallelism can tolerate high fragment degree. Compared to zero fragment degree, the throughput of the uplink/downlink increases by 34.1%/30.6% when fragment degree reaches 100.0%. The reason is that a single GPU exposes the FIFO queue structure [55, 105], limiting the number of concurrent scheduled frames. With one more GPU, the number of frames that can be scheduled simultaneously increases, causing overall throughput improvement. Although resources are split between two GPUs, each individual GPU exhibits partial parallelism. As shown in the partial parallelism, a slight reduction in the number of SMs can still be accommodated. This indicates that partitioning would not lead to a significant performance drop.

The execution strategy at the symbol level is similar; however, symbols (other than pilot) cannot run before the completion of the pilot symbol in each frame. If symbols do not stay within the same GPU as the pilot, extra data shuffling is required for another GPU to synchronize results. Also, symbols show up to 25.1%/65.1% increase under low fragment degrees (from 0.0% to 44.0%/16.0% fragment degree for uplink/downlink) thanks to the increasing hardware request



(a) Uplink.



(b) Downlink.

Figure 6: Baseband processing perf. at three granularities varying background jobs #. We measure the per-(frame | symbol | task) throughput (K frames per second).

queue. It even outperforms the frame-level parallelism (up to 17.8%/33.3% higher) due to the fine-grained execution. This provides a GPU with a wider range of options for selecting smaller jobs to run when there are limited SMs, yielding higher utilization and performance improvement. Conversely, coarse-grained large kernels offer less flexibility, resulting in an inefficient use of fragmented resources. Nevertheless, symbol parallelism underperforms the frame case when the fragment degree exceeds 70.0%. At 100.0% fragment degree, we observe a 14.5%/17.4% uplink/downlink performance drop. The reason is that extra data shuffling required for symbol-level scheduling cannot be hidden by computation under a high fragment degree (discussed in Appendix A.1).

When parallelizing at the task granularity, we also partition tasks proportionally to two GPUs. A key challenge is to minimize data shuffling overheads. We follow the Hydra [71] approach that only synchronizes when the parallelism changes. This indicates that equalization and demodulation (precoding and modulation) tasks across the uplink (downlink) pipeline run in the same GPU. Task level experiment performs worst according to Figure 5, presenting a 48.2%/52.2% uplink/downlink throughput drop. In contrast to the symbol case that only synchronizes pilot computation once per frame, tasks synchronize data between every two stages for non-pilot symbol processing, resulting in 13 times of synchronization (excluding pilot) per frame in a typical 14-symbol frame.

Takeaways. Fragmented parallelism, albeit providing enough SMs, should carefully divide jobs based on the underlying GPU's computing capability to mitigate jeopardy. Finer-grained execution at the symbol level helps utilize the available SMs in each GPU. But one would see diminishing returns when synchronization overheads dominate. There is a trade-off between execution granularity and data shuffle frequency.

3.4 Delayed Parallelism

Setup. We design the delayed parallelism experiment by interleaving the target baseband processing job with special background kernels. They run large-scale GEMM (General Matrix Multiplication) tasks, where $M \times N \times K = 1280 \times 1024 \times 512$, consuming as many available SMs as possible. A bunch of such kernels is being pushed to the GPU consistently one after another. We investigate how the GPU internal scheduler affects different-sized baseband processing jobs.

Results. We find that the frame-based execution adapts to delayed parallelism the best, followed by symbol and then task granularity. For example, when there are 100 interleaved kernels, the uplink/downlink frame pipeline observes 23.5%/30.9% degradation while uplink/downlink symbol is dropped by 47.2%/60.8%, and uplink/downlink task is 60.0%/74.7% lower. The key reason is that GPUs employ a strict FCFS scheduling policy with a limited amount of hardware execution queues [55, 105, 106], providing little fairness guarantees. When execution contention happens, coarse-grained scheduling submits a sequence of jobs in batches and thus would receive more GPU computing cycles. Besides, frequent kernel launches and completions at a finer granularity cause more GPU interruptions and jeopardize performance.

Takeaways. The impact of delayed parallelism stems from how the computing resource of a GPU is shared among competing kernels. The inherent FCFS policy and kernel-agnostic partition make inefficient multi-tenancy support on the GPU. Thus, when determining the execution granularity under GPU contention, one should collaboratively consider the kernel size distribution of co-located baseband processing jobs.

3.5 Summary

There is no *one-size-fits-all* execution granularity that allows the baseband processing pipeline to accommodate different irregular hardware parallelism. Therefore, an ideal baseband processing solution over an SNC should be execution adaptive, aware of the underlying target status at runtime, and collectively consider co-located kernels. None of the prior solutions [28, 30, 58, 65, 71, 100, 114, 121] can achieve this.

4 MegaStation: Design and Implementation

This section describes MegaStation and shows how we address the above dilemma. Our system design goals are:

- **Delay deadline.** MegaStation should satisfy the processing latency requirements of each time frame. It should ensure the average/tail delay stays within the constraint;
- **High utilization.** MegaStation should fully harness the available hardware parallelism (no matter which type) and minimize stalled or idle GPU cycles. It should strive to consolidate baseband processing tasks on just enough GPUs without over- or under-subscriptions;
- **Adaptability.** MegaStation makes no assumption about the frame structure and can handle different uplink/downlink frames. It should automatically adapt its processing capability to the target spectrum, and client rate.

4.1 Key Idea and System Overview

Key Idea. MegaStation effectively leverages the massive parallelism of a single-node supercomputer to process data frames. *Our key insight is that one can adjust the execution granularity and reconstruct the baseband processing pipeline on the fly based on the hardware parallelism status.* We are inspired by the dynamic instruction scheduling [115, 116] from computer architecture field. Dynamic instruction scheduling reorders instructions to prevent stalls by scheduling indepen-

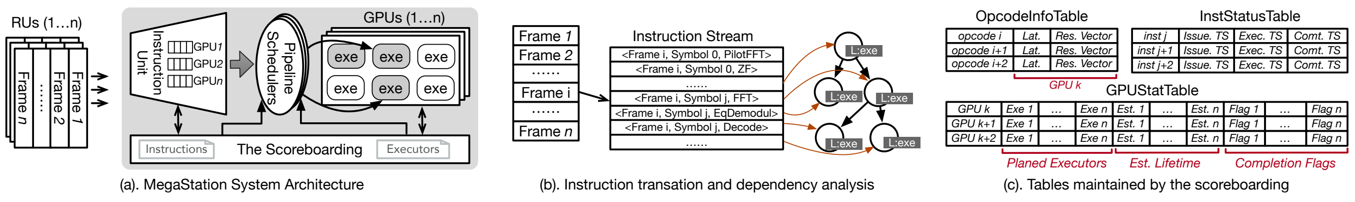


Figure 7: (a) provides an overview of MegaStation. (b)/(c) depict the details of the instruction unit and scoreboarding.

dent instructions when the execution engine is free. We view the baseband processing task on GPUs as instructions and apply dynamic scheduling. This flexible approach allows us to make real-time adjustments to the software parallelism based on the status of the GPUs. Hence, *we model the single-node supercomputer as a tightly coupled microprocessor and employ a scoreboarding-like algorithm* to schedule "baseband processing" instruction streams over the composable GPU pool. MegaStation continuously bookkeeps the occupancy of the GPU's computing resources (e.g., SMs, registers, and shared memory), and issues an instruction when it is ready. Within the frame deadline, MegaStation coalesces one or several instructions with data locality to maximize the probability of full parallelism, mitigates the fragmented/partial parallelism via instruction reordering, and tackles the delayed parallelism via managed over-commitment.

System Overview. MegaStation comprises four components (Figure 7-a): (a). *Processing Functional Units or Executors (§4.3)*, running baseband processing tasks over the GPU SMs; (b). *Instruction Unit (§4.4)*, which takes data frame sequences as inputs, translates them into instructions, and places them into an instruction queue; (c). *The Scoreboarding (§4.5)*, a centralized bookkeeper that continuously tracks the execution status of inflight instructions and performs resource accounting of the GPU pool, serving as the basis for other system components; (d). *Pipeline Scheduler (§4.6)*, which orchestrates instruction execution, determines what and how to run the next tasks based on committed status. MegaStation can support one or multiple radio units concomitantly [62, 71].

4.2 Baseband Processing Tasks as Instructions

MegaStation models a baseband processing task as an instruction, represented as $\langle$`Opcode`, `Operand1`$_{src}$, `Operand2`$_{src}$ (optional), `Operand`$_{dst}\rangle$. Our current design uses 4 bits to denote an opcode and supports 11 different opcodes (Table 1 in Appendix A.2), i.e., Data load & store, PilotFFT, UpFFT, iFFT, zero-forcing (ZF), equalization + demodulation (EqDemodul), modulation (Modul), precode, decode, and encode. Among them, equalization and precoding take three operands, the second source operand being a beamforming matrix, while the rest require two. An operand is mostly a pointer to the data input/output location. Thanks to the local view of GPUs on a single-node supercomputer, multiple GPUs share a unified memory address space when presenting a src/dst operand.

The lifecycle of an instruction comprises three basic execution stages: **ISSUE**, **EXEC**, and **COMMIT**. First, the instruction unit conducts dependency analyses of each enqueued instruction and marks it ready for issuing when data and executor are in place. Next, an instruction is scheduled and inserted into the request queue of a GPU executor. A functional unit processes requests following its prescribed order. Finally, upon completion, commit notification is broadcast to other architectural components (like the scoreboard), pushing the pipeline forward. At the beginning of each stage, We collect the timestamps TS_{issue}, TS_{exec}, TS_{commit}.

4.3 Baseband Processing Executors over GPUs

MegaStation instantiates GPU-based task executors on demand and uses them to model instruction running on GPUs. Since a GPU does not expose any internal resource usage primitives at run-time, an executor is not a physical computing entity but a logical representation that describes its required resources. We use a tuple $\langle$CTA #, Warp #, Regs #, SMem Bytes$\rangle$ to represent the executor. Table 1 presents the executor information in our design for a 128×32 MIMO configuration. Warp #, Regs #, SMem Bytes information can be obtained from the compiler or profiler; while CTA # is specified at the kernel launch based on the MIMO settings. MegaStation uses this resource to provision executors when planning to make the computation placement decision, while the actual allocation happens after instructions are loaded.

Executor Planning. MegaStation preallocates executors when the instruction unit has generated instructions for an incoming frame. We provision executors for all the enqueued instructions. This is a typical bin-packing problem [82, 122], whose goal is to find the minimal amount of GPUs to hold a list of executors without exceeding the capacity of any computing resources of a GPU. Since (a) the GPU pool keeps fluctuating with instructions issued and retired; (b) this process happens pre-execution; (c) resource allocation and partitioning inside GPU is a blackbox, finding an optimal allocation using an ILP (Integer Linear Programming) like solution is unnecessary. MegaStation resorts to an approximate approach following the first-fit strategy [67]. For each instruction, we first search the in-use GPUs and then the unoccupied ones to hold the executor such that the total amount of CTAs (or SMs), warps, registers, and shared memory of the candidate GPU stays within the device limit. This step updates the `GPUStatTable` of the scoreboarding (Figure 7-c).

Different GPU architectures exhibit diverse computing capacities [41]. Newer GPUs could hold more executors simultaneously with more CTAs and global memory. An executor's

Algorithm 1 Structural Analysis.

```
1: Inst_DAG: the DAG within the instruction unit
2: function PLAN_INST_EXECUTOR(cur_inst)
3:     preced_insts ← preced_nodes(Inst_DAG, cur_inst)
4:     est_time ← ESTIM_TIME(cur_inst, preced_insts)
5:     if $est\_time.start - TS_{current} > 2T_{frame}$ then return Resched.
6:     pred_GPU, pred_SM ← MAX_SM(preced_GPUs, est_time)
7:     if $pred\_SM > \alpha\, pred\_GPU.total\_SM$ then
8:         return pred_GPU                          ▷ Full/Partial Parallelism
9:     max_GPU, max_SM ← MAX_SM(all_GPUs, est_time)
10:    if $max\_SM < \alpha\, max\_GPU.total\_SM$ then
11:        return delayedQ.push(cur_inst)  ▷ Delayed Parallelism
12:    if $pred\_GPU.wait < data.copy$ then      ▷ Fragmented Para.
13:        return pred_GPU
14:    else /* including $pred\_GPU = NULL$ */
15:        return max_GPU
16: function ESTIM_TIME(cur_inst, preced_insts)
17:    if $cur\_insts = frame\_start$ then
18:        return ESTIM_FRAME($T_{frame}$)
19:    else if $cur\_insts = symbol\_start$ then
20:        return ESTIM_SYMBOL($T_{symbol}$)
21:    else
22:        return ESTIM_INST($T_{inst}$)
```

resource tuple may vary depending on the GPU architecture it is compiled for. These all require us to keep track of the instructions' behavior in different GPUs (`OpcodeInfoTable`) and monitor the device limit and planned executors of a GPU (`GPUStatTable`), used by our scoreboarding scheme (§4.5). The scheduler (§4.6) then consults the scoreboarding to decide which instructions run next on which GPUs at runtime.

4.4 Instruction Unit

MegaStation receives IQ samples from the fronthaul link and delivers them to the instruction unit. It performs three tasks: instruction translation, dependency graph construction, and structural analysis. As shown in Figure 7-b, we first decode the frame structure, parse each pilot/uplink/downlink symbol, and then generate instructions for the pipelined tasks. MegaStation attaches a prefix to each instruction to facilitate later analysis: $\langle$Frame i, Symbol $j\rangle$+*Inst*.

Next, the instruction unit builds the directed acyclic graph (DAG) among pending instructions to capture the execution dependency. MegaStation explores this from both data and structural perspectives. Regarding data dependency, we construct the DAG graph by analyzing the RAW (read-after-write) relation, where the source operand of instruction $I2$ depends on the destination operand of instruction $I1$. For example, an *Inst*[equalization] can be executed when it receives the beamforming matrix from the preceding *Inst*[zero-forcing] and the output of *Inst*[FFT]. In our case, WAR (write-after-read) and WAW (write-after-write) rarely happen because the destination operands of subsequent tasks along the baseband processing pipeline differ from the source or destination operands of precedent tasks. Since time frames are independent, the DAG is per-frame, whose size is bounded.

For structural analysis, MegaStation analyzes each GPU status to determine its hardware parallelism. According to this analysis, MegaStation decides the GPU on which the executor of instruction should run. Unlike microprocessors, our functional units are the SMs of each GPU and are not fixed. This indicates that an instruction could have multiple executor candidates (§4.3) based on the runtime hardware occupancy. As runtime information fluctuates frequently with instructions being submitted and committed, merely obtaining the latest GPU status at runtime is insufficient. There exists a time gap between when an instruction is issued and when it actually runs on GPUs. In order to predict the GPU status at the time when the instruction is running, we maintain a `GPUStatTable` with the status of all the planned executors (§4.5), which is used to estimate the lifetime of an instruction.

ALG1 dispatches instruction executor to different GPUs based on hardware parallelism within instruction's running time range. To enhance estimation accuracy, MegaStation only schedules instructions that are set to begin within a two-frame time window. MegaStation employs a top-down approach, initially finding a fit for the entire frame before moving on to symbol and then instruction (ALG1: ESTIM_TIME). We prioritize GPUs with input locality, coalescing instructions with dependency. If the minimum number of available SMs during the instruction's running time range is less than threshold partial degree α times the total number of SMs of the GPU (indicating full or partial parallelism), we could safely schedule the instruction to that GPU (ALG1:L7-8). α is GPU architecture-dependent and profiled offline. When adding a new GPU architecture (discussed in §3.2), we vary the partial degree under three execution granularities, measure the uplink/downlink performance (Figure 4), and empirically determine α which yields marginal performance drops (10%).

When there are insufficient SMs in the GPU of precedent nodes or there are no precedent nodes (i.e., a new frame is received), we search for a GPU with the maximum number of free SMs within the instruction's time range. If the newly searched GPU exhibits an adequate number of SMs, it indicates fragmented parallelism. Based on the findings in §3.3, the algorithm must consider the wait time due to the SM scarcity and data transfer time between the GPUs (ALG1: L12-L13). For new incoming frames (no precedent GPUs), we always choose one GPU with the lowest occupancy.

When the newly searched GPU still lacks SM (ALG1: L10-L11), suggesting delayed parallelism, we enqueue the instruction to a delay priority queue where frames are sorted based on their deadlines. All incoming and unscheduled frames are redirected to the delayed queue until there are sufficient available SMs and the delayed queue is empty. Instructions in the delayed queue are assigned to the GPU with minimal occupancy at the frame level granularity and labeled as "delayed". When the frame's processing deadline is violated, it indicates that the incoming baseband processing load exceeds the MegaStation's capacity. All the obsolete frames are discarded and MegaStation sends back a customized NACK signal that triggers load balancing (4.7). If a consistently heavy load is observed, we would add more GPUs from the current SNC (if there are free ones) or redirect traffic to another SNC.

4.5 The Scoreboarding

MegaStation bookkeeps execution statistics for instructions and GPU hardware using a scoreboarding mechanism. In terms of instruction tracking, as depicted in Figure 7-c, we introduce two tables. One is the `OpcodeInfoTable` that stores executor information regarding each instruction opcode. Each entry in this table contains the resource vector of its corresponding executor for each type of GPU and the average execution latency within a calibrating window. The resource vector of each opcode's executor is obtained from the compiler or profiler prior to MegaStation initialization and remains static unless a new type of GPU is added. In such cases, the resource vector is profiled for that type of GPU before executors can be scheduled to the GPU. Upon completion of an instruction, the execution time is calculated as $T_{inst} = TS_{commit} - \max(TS_{preced\_commit}, TS_{exec})$, where TS_{preced_commit} denotes the commit timestamp of instruction's precedent instruction. The execution time (T_{inst}) is averaged within a calibrating window to mitigate fluctuations. Along with the execution time of each instruction, MegaStation also maintains the average execution time of each category of frame (T_{frame}) and symbol (T_{symbol}), and the average scheduling time in pipeline scheduler $T_{sched} = TS_{exec} - TS_{issue}$ in `OpcodeInfoTable`. The other is the `InstStatusTable` that keeps track of timestamps in the **ISSUE**, **EXEC**, and **COMMIT** stages for each instruction. When the instruction unit plans the executor, it records TS_{issue}. Upon submission of the executor by the pipeline scheduler, TS_{issue} is measured. Finally, when the GPU signals completion, TS_{commit} is stored.

Regarding monitoring the utilization of GPU hardware, as shown in Figure 7-c, we maintain a data structure called `GPUStatTable` for each GPU to track the status of each instruction executor deployed to that GPU. This table comprises (a) the number of SMs used by each executor; (b) the estimated start and end timestamp for each executor; and (c) a completion flag indicating the retirement of the executor. The SM usage of an executor is determined by CUDA's occupancy calculator [19], which considers warp, register, and shared memory information to calculate the number of CTAs that an SM can hold. This information helps obtain the number of SMs needed to execute a given instruction. The planned executor is inserted by the instruction unit with the estimated timestamp. The scoreboarding actively queries GPUs about the status of each executor and updates the completion flag.

4.6 Pipeline Scheduler

The pipeline scheduler instructs what to run next in each GPU with a finer granularity than ALGO1. It consults the scoreboarding (§4.5), fetches instructions (baseband processing tasks) from the instruction unit, and deploys them over its GPU, which adjusts the software execution granularity to the dynamic hardware parallelism adaptively at runtime. Each scheduler traverses its instruction queue (§4.4) (Figure 7-a) and fires out instructions using a new algorithm, called LROC (ALG2, Appendix A.2)–a careful synthesis of Least Slack Time First (LSTF) scheduling along with instruction Reordering, Over-commitment and Coalescing techniques.

LSTF Scheduling. As described before, the baseband processing of massive MIMO has stringent performance requirements, especially per-frame processing delay. Thus, we apply the LSTF scheduling discipline to prioritize task execution. MegaStation implements the instruction queue using a red-black tree with a list of keys. The first key is slack timestamp, which is computed with two factors: frame-induced hard deadline (TS_{ddl}), reserved executing time of the instruction and its succedents in the frame along the depth of the DAG (dubbed as $\sum^{Depth} T_{inst}$), where T_{inst} is acquired from the scoreboarding (§4.5) and calibrated online. $TS_{slack} = TS_{ddl} - \sum^{Depth} T_{inst}$.

Reordering. As discussed before (§3), the baseband processing performance is affected by the amount of execution parallelism, parallelism distribution, and data movement overheads. When planning (§4.4), MegaStation predetermines the executor of an instruction based on overall available resources across GPUs. When executing, it adjusts the instruction order with runtime status of the local GPU for performance maximization. The second key of the instruction queue is SM usage of each instruction. Our algorithm prioritizes scheduling instructions with the least SM usage to greedily occupy the remaining GPU resources after handling urgent instructions. In this way, it opportunistically reorders instructions with no dependency to improve GPU utilization.

Over-commitment. MegaStation strives to keep the GPU busy (i.e., work-conserving) and reduce the idle cycles when waiting for the host to launch CUDA kernels if the GPU's SMs become available. Our scheduler thus allows computation overloading, inevitably introduces delayed parallelism, and resolves it through the stream priority mechanism provided by CUDA. After the GPU is full, MegaStation schedules an instruction labeled as "delayed" and submits it to a low-priority CUDA stream. This allows the instruction to run whenever the GPU has available resources without interfering with higher-priority normal instructions.

Coalescing. To avoid the synchronization gap between GPU completion and CPU being signaled, MegaStation provision instruction scheduling with the help of the sequential property within a single CUDA stream. If an instruction has precedent instructions on the local GPU that require a larger resource than the current instruction's executor, we preemptively submit the current instruction to the same CUDA stream with the precedent instructions without waiting for the completion signal. This preemptive submission has minimal impact on future scheduling, as the current instruction can inherit resources reserved by the precedent executor on the GPU.

4.7 Deployment and Failure Handling

Fault tolerance is crucial in practice. We apply Slingshot's approach [89] to handle failures, i.e., treating short-term loss of frames as bad signal quality effects in the wireless environment and recovering them through inherent cellular network impairments. All discarded frames in MegaStation are handled similarly to Slingshot [89]. Since frames are independent, some frame loss would not affect other frames' processing.

There are three failure domains in MegaStation: server host, interconnect fabric, and GPU pool. When the host fails, MegaStation enters fail-stop mode. All pending frames are discarded and incoming frames should be reissued to another backup vRAN server. For the interconnect failure, MegaStation loses connection with all the GPUs. One option is to use the same solution as host failure, or alternatively fall back to a CPU-only mode with a slower frame processing rate.

MegaStation depends on GPU drivers to detect GPU failure. If a GPU malfunctions, device exceptions are raised when trying to submit instructions or query instruction status, causing it to be excluded from the scoreboard. Any running and pending (uncommitted) instructions on the GPU are moved to the instruction queues for rescheduling. In the meantime, a daemon continuously monitors PCIe device lists for new GPUs. Once a new functional GPU is detected, it is installed by MegaStation to the scoreboard for instruction scheduling.

4.8 Implementation Details

We implement MegaStation atop Agora [65] and NVIDIA Aerial SDK [30] in C++ with 8372 LOC and CUDA with 2147 LOC. Regarding the baseband processing instruction, MegaStation uses CUTLASS [22] and cuSOLVER [21] when computing the CSI matrices, and cuFFT [26] for FFTs. For LDPC encoding and decoding, we use the NVIDIA Aerial SDK [30] and enhance it with inline data copy. We reimplement precoding, equalization, modulation, and demodulation functions in CUDA following Agora's algorithms. Our executors leverage the GPU hardware features. For example, we use tile-based iterators [120] to fuse data transformation into the data loading and storing stage. We develop a lightweight mechanism for instruction synchronization, where each instruction allocates a flag on GPUs. These flags are manipulated via CUDA stream operation [20] on GPU side and are synchronized via GDRCopy [27] to CPU side. MegaStation employs a master-slave architecture. A master thread realizes the instruction unit, where per-GPU slave threads run the pipeline scheduler. Each slave thread also manages a pool of memory blocks, which translates the operand label to the actual memory pointer. Appendix A.3 presents more details.

5 Evaluation

5.1 Experimental Methodology

Hardware testbed. We use the GigaIO FabreX-based SuperNODE [8, 44] as the single-node supercomputer target.

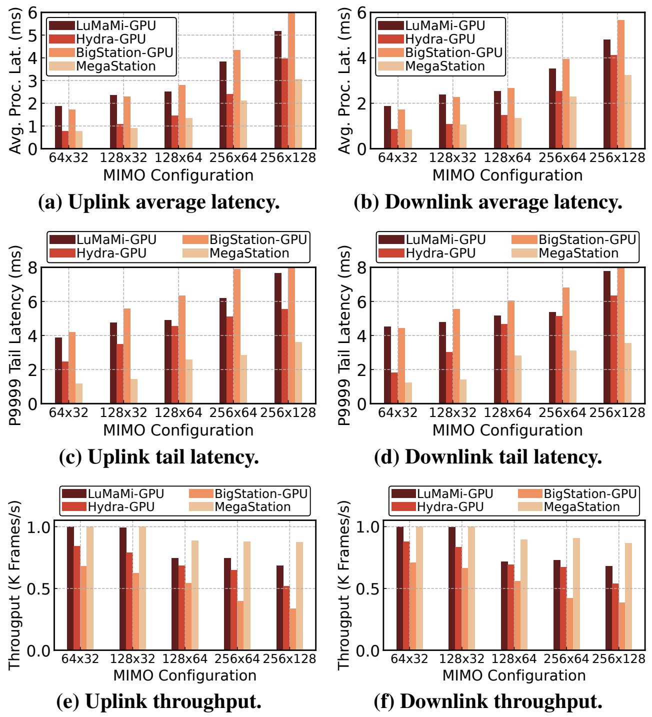


Figure 8: Average/tail uplink/downlink per-frame processing latency and throughput comparing five MIMO settings.

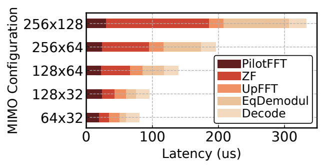
The host server is a 2U Dell R740 box, consisting of two 32-core Intel Xeon Gold 6248 processors (running at 2.5GHz), 192GB DRAM, and 1.92TB HDD. The load/store fabric uses routable PCIe [2], whose details are described in §2.4. We use NVIDIA V100 and A100 GPUs in the composable pool.

Comparison baselines. We compare MegaStation with three baselines: LuMaMi [100], Hydra [71], BigStation [121]. Since none of these prior systems target the GPU pooling, we develop the baseline over our target following their proposed design. The first one (*LuMaMi-GPU*) performs frame-level parallel execution and distributes frames to available GPUs in a work-conserving manner. The second one (*Hydra-GPU*) parallelizes the pipeline at the symbol level. Akin to Hydra [71], we place dependent stages in the same GPU to minimize the cross-GPU communication overheads. The third one (*BigStation-GPU*) implements task-level parallelism, where tasks are distributed to different executors.

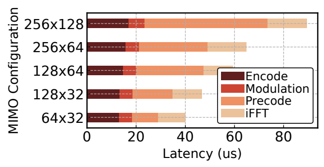
Workload. We use the Agora [65] traffic generator to emulate an RU (radio unit). It applies the Rayleigh fading channel and transmits time-domain IQ samples. Our experiments use 2048 subcarriers and consider different MIMO configurations.

5.2 End-to-end Performance

We set up 1 RU under 5 massive MIMO configurations (i.e., 64×32, 128×32, 128×64, 256×64, 256×128) and launched the experiment over just one chassis box with A100 GPUs. We measure the average and P9999 (tail) processing latency, and overall throughput for both 1ms uplink and downlink frames. As shown in Figure 8, MegaStation meets



(a) Breakdown on uplink path.



(b) Breakdown on downlink path.

Figure 9: Latency breakdown for five MIMO settings.

the deadline requirements, with P9999 latency ranging from 1.2 to 3.6 ms for uplink frames, and 1.2 to 3.5 ms for downlink frames across different MIMO settings. We satisfy the stipulated 4m deadline for 1ms frames.

Across 5 MIMO cases, MegaStation achieves 50.6/58.9%, 12.2/46.9%, 53.7/66.2% lower average/tail latency on average for the uplink frames compared with *LuMaMi-GPU*, *Hydra-GPU*, and *BigStation-GPU* approaches, and 45.2/57.1%, 9.5/41.6%, and 47.9/61.9% for the downlink ones, respectively. For throughput, MegaStation performs 10.7/12.2%, 25.3/22.7%, and 44.9/41.8% better than three baselines on average for the uplink/downlink frames. MegaStation outperforms *BigStation-GPU* the most because the latter incurs too much cross-task synchronization. *LuMaMi-GPU* and *Hydra-GPU* perform suboptimally mainly because (a) their coarse-grained execution cannot fully leverage all available SMs promptly, especially when the GPU parallelism becomes irregular; (b) the scheduling mechanism is workload-agnostic, ignoring data locality and deadlines. Both are captured in our structural analysis, which places executors on suitable GPUs, and pipeline scheduler, which reorders instructions based on slack time and coalesces them with data locality.

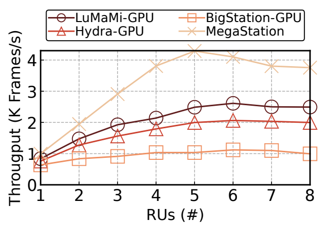
5.3 Uplink/Downlink Pipeline Latency Breakdown

We break down the uplink and downlink processing latency of MegaStation (Figure 9). Overall, zero-forcing is the most expensive stage, consuming 36.3% of the whole pipeline on average across five MIMO cases. However, it only happens for the pilot symbols which could be amortized by uplink/downlink symbols. The next time-consuming stage is EqDemodul of the uplink path and Precoding of the downlink path, which spends 44.2% and 42.7%. We also compare the GPU executor with the CPU one (Agora [65]). Our experiment chooses the 64×16 MIMO, the largest one supported by Agora [65]. The CPU one uses 60 cores, while we use a single V100 GPU in MegaStation. The results (Figure 13 in Appendix A.4) demonstrate that the GPU can achieve an average speedup of $10\times$ compared to the CPU due to its massive parallelism.

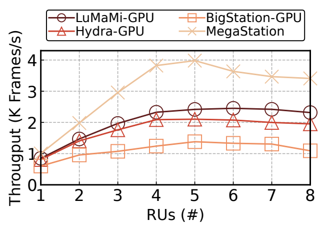
5.4 Scalability

We increase the number of A00 GPUs on the SNC and analyze scalability in two dimensions: (a) how MegaStation performs as the number of RUs increases, and (b) how MegaStation scales with an increasing number of GPUs.

RU Scalability. We gradually increase the number of RUs from 1 to 8 under 128×64 MIMO for both uplink and downlink frames. Figure 10 presents the throughput per-

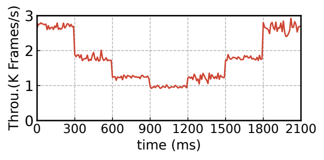


(a) Uplink throughput.

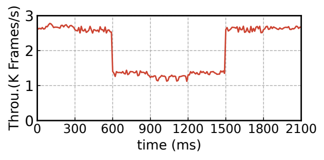


(b) Downlink throughput.

Figure 10: We report the frame throughput as increasing the number of consolidated RUs for the 128×64 MIMO.



(a) 3 RUs 128×64.



(b) 3 RUs 128×32.

Figure 11: We adjust the number of GPUs from 5 to 2 and back to 5 every 300 ms while measuring the throughput every 10 ms.

formance of four approaches. MegaStation scales the best and can harvest all the available SMs for frame processing. Over uplink/downlink frames, MegaStation achieves 1.5/1.4×, 1.8/1.6×, 3.2/2.6× higher performance compared with *LuMaMi-GPU*, *Hydra-GPU*, and *BigStation-GPU*. They scale poorly due to the emergence of irregular hardware parallelism when more RUs send frames concurrently, yielding inefficient parallel execution and lower performance. MegaStation scales nearly linearly with the number of RUs regarding throughput. This is because MegaStation uses scoreboarding to track the real-time status of each GPU, allowing our scheduler to allocate tasks based on GPU availability and effectively manage congestion. Also, the over-commitment technique further helps exploit GPU computing power in the delayed parallelism and minimizes GPU idle cycles.

The number of RUs cannot be scaled infinitely. It is bottlenecked by the fabric interconnect between the host and chassis. The maximum number of RUs is theoretically bounded by $\frac{F \times S \times RU\#}{B} < T$, where F is frame rate (frames per second), S is frame size (bytes per frame), B is the bandwidth of interconnect (bytes per second), and T is frame duration (seconds). When the interconnect saturates, adding more GPUs would not help. One could load balance traffic among multiple SNCs via a fronthaul switch to solve this problem [89].

GPU Scalability. We add the number of GPUs from 1 to 5 under 3 RRUs under a 128×64 MIMO (Figure 14 in Appendix A.4). MegaStation achieves 1.9/1.8× speedup for uplink/downlink frames. However, *LuMaMi-GPU*, *Hydra-GPU*, and *BigStation-GPU* only gain 1.3/1.2×, 0.9/1.0×, and 0.6/0.7× improvement, respectively. The performance drop of *LuMaMi-GPU* is attributed to its coarse-grained execution granularity, which fails to fully utilize the increasing SM resources. *Hydra-GPU* and *BigStation-GPU* exhibit limited speedups due to higher communication overhead introduced by distributing symbols or tasks across more GPUs. MegaStation addresses these problems by (a) scheduling at task-level granularity, which is the finest granularity in baseband processing; (b) in structural analysis, taking data transfer time into consideration when distributing instructions with dependency to different GPUs; (c) implementing coalescing technique to avoid GPU waiting for instructions.

5.5 MegaStation Drill-Down

We evaluate each system component in MegaStation via instruments and report their CDF. As shown in Figure 15 (Appendix A.4), we find that 90% of the instruction unit (IU) execution takes less than 12 μs, indicating that it is not the system bottleneck even if it is single-threaded. The per-GPU pipeline scheduler (PS) is 1.3$\times$ more costly than IU because it interacts with the CUDA driver on submitting instructions.

The pilot symbol processing is the most consuming part (Figure 15-b/c), where 90% of the execution takes 0.3ms to compute zero forcing and 0.12ms to transfer the zero-forcing (ZF) results to other GPUs. To accelerate ZF under large MIMO, we split the ZF instruction into multiple sub-instructions based on frequency bins and distribute them to multiple GPUs. Subsequent instructions would then have a dependency on the sub-instructions, captured in the DAG. To amortize large GPU-GPU communication, we balance irregular parallelism-induced waiting time and data shuffling overheads when copying tasks to idle GPUs.

5.6 Discussion

Reliability. There are three failure domains in MegaStation (§4.7). We manually toggle GPU availability in the GigaIO FabreX control panel to observe system response to the GPU fluctuation. We alternate between enabling and disabling GPUs, transitioning from 5 to 2 and back to 5 every 300 milliseconds. We configure 3 RRUs under the 128×64 and 128×32 MIMO and measure the throughput every 10 ms. The system throughput decreases with the number of GPUs, but returns to its original level when GPUs are restored to the initial setting (Figure 11). The scoreboarding scheme allows monitoring the status of each GPU in real time, preventing further scheduling instructions on unavailable GPUs.

Heterogeneous GPUs. We examine how performance is affected when utilizing heterogeneous GPUs. We compare A100+V100 GPUs vs. A100-Only. We show that *LuMaMi-GPU*, *Hydra-GPU*, and *BigStation-GPU* experience a 23.1%, 30.7%, and 32.8% average latency increase (Appendix A.4), respectively, while MegaStation only observes a 16.1% increase. Our method could handle heterogeneity better for two reasons. First, MegaStation takes different GPU architectures into account and dynamically adjusts its scheduling policy based on the available SMs. Second, MegaStation could capture the increased data movement overhead during profiling and rebalance the waiting and communication overhead.

In Appendix A.4, we show how MegaStation executes under small MIMOs, operates under other GPUs, and tackles the fragmented parallelism issue.

6 Related Work

Quantum Acceleration. Researchers have explored using quantum computing to tackle the high computing demand of massive MIMO. QuAMax [86] leverages quantum annealing to build the Sphere Decoder-based maximum likelihood (ML) MIMO decoders/detectors. HyPD [83] develops a hybrid classical-quantum decoder for polar error correction codes, where it employs CMOS processing for the decoder's binary tree traversal and quantum annealing processing for the polar decoder. Minsung Kim *et al.* [85] investigate a spin-level preprocessing approach for fine-grained decomposition that enables flexible parallel quantum signal processing.

Infrastructure Composability. Emerging high-bandwidth networks [1,16] and cluster interconnects [2,40,54,78] enable physical resource pooling. For example, [91,94,101,113,125] develop object/paging-based remote memory over CXL memory expanders. [57,74,80,87,88,92,96,102–104,126] build disaggregated storage. Researchers also develop composable accelerators [29,59,60] and explore efficient host-accelerator collaborated computation. The single-node supercomputer is another composable platform and we use it for accelerating the baseband processing of massive MIMO.

Network Function. Our work benefits from prior pioneering efforts of accelerating networking functions. RouteBricks [66] parallelizes router functionalities across multiple servers and multiple cores within a single server. PacketShader [75] examines using a commodity GPU to build a software router. NBA [84] introduces system techniques to optimize NF workloads on a CPU-GPU heterogeneous system. [61,69,73,77,90,93,95,97,98,107–112,124] offload networking functions onto reconfigurable switches and SmartNICs. MegaStation follows the past strategy and dynamically exploits the GPU's hardware parallelism at a fine granularity.

7 Conclusion

This paper makes a case for using an emerging single-node supercomputer for the massive MIMO baseband processing. We build MegaStation, an application-platform co-designed system that dynamically adjusts the execution granularity and reconstructs the baseband processing pipeline on the fly to accommodate the irregular underlying hardware parallelism. We model the single-node supercomputer as a tightly coupled microprocessor and employ a scoreboarding-like scheme to orchestrate baseband processing computations. Real system-based evaluations show that MegaStation outperforms comparable state-of-the-art approaches considerably and can fully leverage the capabilities of this new computing platform.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Wenjun Hu, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199, CNS-2212192, and CAREER-2339755.

References

[1] 800G Specification. https://ethernettechnologyconsortium.org/wp-content/uploads/2021/10/Ethernet-Technology-Consortium_800G-Specification_r1.1.pdf, 2021.

[2] FabreX PCIe network fabric: a primer. https://gigaio.com/wp-content/uploads/2021/02/FabreX-PCIe-network-fabric-a-primer-1.pdf, 2021.

[3] Huawei's 5G Indoor Distributed Massive MIMO. https://www.huawei.com/en/news/2021/9/ubiquitous-gigabit-5g-indoor-dmm-wins-award, 2021.

[4] Mavenir and NEC deploy massive MIMO in France. https://www.lightreading.com/open-ran/mavenir-and-nec-deploy-massive-mimo-in-france, 2022.

[5] Nokia and AT&T collaborating to improve 5G uplink with distributed massive MIMO. https://www.nokia.com/about-us/news/releases/2022/02/28/nokia-and-att-collaborating-to-improve-5g-uplink-with-distributed-massive-mimo-mwc22/, 2022.

[6] The 5G Network Backbone: A Guide to Small Cell Technology. https://www.telit.com/blog/5g-networks-guide-to-small-cell-technology/, 2022.

[7] Enfabrica's Accelerated Compute Fabric. https://blog.enfabrica.net/press-release-enfabrica-raises-125-million-series-b-to-fuel-ramp-of-ai-infrastructure-networking-a8a0b21653d2, 2023.

[8] GigaIO's FabreX System. https://gigaio.com/products/fabrex-system-overview/, 2023.

[9] GroqRack Compute Cluster. https://groq.com/wp-content/uploads/2022/10/GroqRackâĎć-Compute-Cluster-Product-Brief-v1.0.pdf, 2023.

[10] H3's Falcon System. https://www.h3platform.com/solution/composable-ai, 2023.

[11] Introducing Azure Operator Nexus. https://techcommunity.microsoft.com/t5/azure-for-operators-blog/introducing-azure-operator-nexus/ba-p/3753393, 2023.

[12] KDDI kicks off deployment of 5G Open vRAN sites in Japan. https://www.datacenterdynamics.com/en/news/kddi-kicks-off-deployment-of-5g-open-vran-sites-in-japan/, 2023.

[13] Liqid's SmartStack System. https://www.liqid.com/products/gpu-on-demand, 2023.

[14] T-Mobile's 5G network gets capacity boost from MU-MIMO: report. https://www.fiercewireless.com/tech/t-mobiles-5g-network-gets-capacity-boost-mu-mimo-report, 2023.

[15] The 5G vRAN solution from Fujitsu. https://www.fujitsu.com/global/about/resources/news/press-releases/2023/0220-01.html, 2023.

[16] Ultra Ethernet. https://ultraethernet.org/wp-content/uploads/sites/20/2023/10/23.07.12-UEC-1.0-Overview-FINAL-WITH-LOGO.pdf, 2023.

[17] 3GPP Release 17. https://www.3gpp.org/specifications-technologies/releases/release-17, 2024.

[18] 5G UE Data Rate. https://devopedia.org/5g-ue-data-rate, 2024.

[19] CUDA Occupancy Calculator. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator, 2024.

[20] CUDA Stream Memory Operations. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__MEMOP.html, 2024.

[21] cuSOLVER. https://docs.nvidia.com/cuda/cusolver, 2024.

[22] CUTLASS 3.5. https://github.com/NVIDIA/cutlass, 2024.

[23] FA4003–FabreX PCIe Gen3 Adapter. https://gigaio.com/wp-content/uploads/2022/02/GIO_NetworkAdapterCard_G3-V4-2.pdf, 2024.

[24] FA4004–FabreX PCIe Gen4 Adapter. https://gigaio.com/products/fabrex-network-adapter-card/, 2024.

[25] FabreX Gen3 Top of Rack PCIe Switch Hyper-Performance Network. https://gigaio.com/wp-content/uploads/2022/02/GIO_FabreX_TOR-Gen3-V4-1.pdf, 2024.

[26] Fast Fourier Transform for NVIDIA GPUs. https://developer.nvidia.com/cufft, 2024.

[27] GDRCopy. https://github.com/NVIDIA/gdrcopy, 2024.

[28] Intel FlexRAN Reference Architecture for Wireless Access. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html, 2024.

[29] Leo CXL Smart Memory Controllers. https://www.asteralabs.com/products/leo/leo-cxl-memory-connectivity-controllers/, 2024.

[30] NVIDIA Aerial SDK. https://developer.nvidia.com/aerial-sdk, 2024.

[31] NVIDIA DGX Platform. https://www.nvidia.com/en-us/data-center/dgx-platform/, 2024.

[32] NVIDIA HGX AI Supercomputer. https://www.nvidia.com/en-us/data-center/hgx/, 2024.

[33] RB4082–Accelerator Pooling Appliance. https://gigaio.com/products/accelerator-pooling-appliance/, 2024.

[34] RS4024–Top of Rack PCIe Switch (Gen 4). https://gigaio.com/products/top-of-rack-pcie-switch/, 2024.

[35] SoftBank Corp., NEC and Broadcom Jointly Validate RAN Modernization with Virtualization by Unifying O-RAN Architecture and Telco Cloud. https://www.broadcom.com/company/news/product-releases/61881, 2024.

[36] The O-RAN Alliance. https://www.o-ran.org, 2024.

[37] The SFF-8644 connector. https://cs-electronics.com/sff-8644/, 2024.

[38] Verizon advances O-RAN technology by deploying 130,000 new O-RAN capable radios in its network. https://www.verizon.com/about/news/verizon-advances-o-ran-technology, 2024.

[39] 5G Sites infrastructure. https://www.ericsson.com/en/ran/5g-sites-infrastructure, 2025.

[40] Computer Express Link Specification. https://computeexpresslink.org/cxl-specification/, 2025.

[41] CUDA Compute Capabilities. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities, 2025.

[42] Ericsson Antenna System. https://www.ericsson.com/en/portfolio/networks/ericsson-radio-system/antenna-system, 2025.

[43] Ericsson Cloud RAN. https://www.ericsson.com/en/ran/cloud, 2025.

[44] GigaIO SuperNODE. https://gigaio.com/supernode/, 2025.

[45] H3C BBU5200 Baseband Unit. https://www.h3c.com/en/Products_and_Solutions/InterConnect/Moblie_Communication/Products/Access/BBU/BBU5200/, 2025.

[46] Huawei DBS5900 Distributed Base Stations. https://e.huawei.com/en/products/wireless/base-station/dbs5900, 2025.

[47] Nokia AirScale Baseband Solutions. https://www.nokia.com/networks/mobile-networks/airscale-radio-access/baseband/, 2025.

[48] NVIDIA GPUDirect: Enhancing Data Movement and Access for GPUs. https://developer.nvidia.com/gpudirect, 2025.

[49] Samsung Baseband Unit. https://www.samsung.com/global/business/networks/products/radio-access/baseband/, 2025.

[50] The Intel vRAN Solution. https://www.intel.com/content/www/us/en/wireless-network/5g-network/radio-access-network.html, 2025.

[51] The Open Virtualized RAN from Mavenir. https://www.mavenir.com/portfolio/mavair/radio-access/vran/, 2025.

[52] The Samsung vRAN Solution. https://www.samsung.com/global/business/networks/products/radio-access/virtualized-ran/, 2025.

[53] The Software-Defined Virtualized RAN (vRAN) from Parallel Wireless. https://www.parallelwireless.com/products/software-defined-virtualized-ran/, 2025.

[54] Ultra Accelerator Link. https://www.ualinkconsortium.org, 2025.

[55] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.

[56] Aslan, Yanki and Roederer, Antoine and Fonseca, Nelson JG and Angeletti, Piero and Yarovoy, Alexander.

Orthogonal versus zero-forced beamforming in multibeam antenna systems: Review and challenges for future wireless networks. *IEEE Journal of Microwaves*, 1(4):879–901, 2021.

[57] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 49–67, 2023.

[58] Manu Bansal, Aaron Schulman, and Sachin Katti. Atomix: A Framework for Deploying Signal Processing Applications on Wireless Infrastructure. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, 2015.

[59] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*, page 37–44, 2022.

[60] Boles, David and Waddington, Daniel and Roberts, David A. Cxl-enabled enhanced memory functions. *IEEE Micro*, 43(2):58–65, 2023.

[61] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*, pages 1–12, 2024.

[62] Yongce Chen, Y. Thomas Hou, Wenjing Lou, Jeffrey H. Reed, and Sastry Kompella. O-M3: Real-Time Multi-Cell MIMO Scheduling in 5G O-RAN. *IEEE Journal on Selected Areas in Communications*, 42(2):339–355, 2024.

[63] Özlem Tuğfe Demir, Emil Björnson, and Luca Sanguinetti. Channel modeling and channel estimation for holographic massive MIMO with planar arrays. *IEEE Wireless Communications Letters*, 11(5):997–1001, 2022.

[64] Ruoqi Deng, Boya Di, Hongliang Zhang, H Vincent Poor, and Lingyang Song. Holographic MIMO for LEO satellite communications aided by reconfigurable holographic surfaces. *IEEE Journal on Selected Areas in Communications*, 40(10):3071–3085, 2022.

[65] Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. Agora: Real-time massive MIMO baseband processing in software. In *Proceedings of the 16th international conference on emerging networking experiments and technologies (CoNext'20)*, pages 232–244, 2020.

[66] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, page 15–28, 2009.

[67] György Dósa and Jirí Sgall. First Fit bin packing: A tight analysis. In *30th International symposium on theoretical aspects of computer science (STACS'2013)*, 2013.

[68] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231, 2010.

[69] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 51–66, 2018.

[70] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[71] Junzhi Gong, Anuj Kalia, and Minlan Yu. Scalable Distributed Massive {MIMO} Baseband Processing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 405–417, 2023.

[72] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 485–500, 2019.

[73] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, page 916–929, 2023.

[74] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 1012–1027, 2023.

[75] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM'10)*, page 195–206, 2010.

[76] Bowen He, Xiao Zheng, Yuan Chen, Weinan Li, Yajin Zhou, Xin Long, Pengcheng Zhang, Xiaowei Lu, Linquan Jiang, Qiang Liu, Dennis Cai, and Xiantao Zhang. DxPU: Large-scale Disaggregated GPU Pools in the Datacenter. *ACM Trans. Archit. Code Optim.*, 20(4), dec 2023.

[77] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, page 33–47, 2023.

[78] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 297–312, 2024.

[79] Christopher Husmann, Georgios Georgis, Konstantinos Nikitopoulos, and Kyle Jamieson. FlexCore: Massively Parallel and Flexible Processing for Large MIMO Access Points. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, 2017.

[80] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, 2020.

[81] Yuki Iida, Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, and Shinpei Kato. GPUrpc: Exploring Transparent Access to Remote GPUs. *ACM Trans. Embed. Comput. Syst.*, 16(1), oct 2016.

[82] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[83] Srikar Kasi, John Kaewell, and Kyle Jamieson. A Quantum Annealer-Enabled Decoder and Hardware Topology for NextG Wireless Polar Codes. *IEEE Transactions on Wireless Communications*, pages 1–1, 2023.

[84] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems (Eurosys'15)*, 2015.

[85] Minsung Kim and Kyle Jamieson. Finer-Grained Decomposition for Parallel Quantum Mimo Processing. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, 2023.

[86] Minsung Kim, Davide Venturelli, and Kyle Jamieson. Leveraging quantum annealing for large MIMO processing in centralized radio access networks. In *Proceedings of the ACM Special Interest Group on Data Communication (Sigcomm'19)*, page 241–255, 2019.

[87] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, 2016.

[88] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 345–359, 2017.

[89] Nikita Lazarev, Tao Ji, Anuj Kalia, Daehyeok Kim, Ilias Marinos, Francis Y. Yan, Christina Delimitrou, Zhiru Zhang, and Aditya Akella. Resilient Baseband

Processing in Virtualized RANs with Slingshot. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, 2023.

[90] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, page 1–14, 2016.

[91] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, pages 574–587, 2023.

[92] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, page 591–605, 2020.

[93] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[94] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[95] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, page 318–333, 2019.

[96] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 305–323, 2020.

[97] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 795–809, 2017.

[98] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 363–378, 2019.

[99] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 289–304, 2020.

[100] Steffen Malkowsky, Joao Vieira, Liang Liu, Paul Harris, Karl Nieman, Nikhil Kundargi, Ian C Wong, Fredrik Tufvesson, Viktor Öwall, and Ove Edfors. The world's first real-time testbed for massive MIMO: Design, implementation, and validation. *IEEE Access*, 5:9073–9088, 2017.

[101] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 3*, pages 742–755, 2023.

[102] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, page 106–122, 2021.

[103] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

[104] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage*, 20(3), June 2024.

[105] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, page 595–610, 2023.

[106] Sreepathi Pai. How the Fermi Thread Block Scheduler Works (Illustrated). https://www.cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html, 2014.

[107] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 663–679, 2018.

[108] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, page 16–22, 2020.

[109] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 772–787, 2021.

[110] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 740–755, 2021.

[111] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 1–16, 2018.

[112] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 685–699, 2020.

[113] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, pages 105–121, 2023.

[114] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. Sora: High Performance Software Radio Using General Purpose Multi-core Processors. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, 2009.

[115] James E Thornton. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, pages 33–40, 1964.

[116] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[117] Li Wei, Chongwen Huang, George C Alexandropoulos, EI Wei, Zhaoyang Zhang, Mérouane Debbah, and Chau Yuen. Multi-user holographic MIMO surfaces: Channel modeling and spectral efficiency analysis. *IEEE Journal of Selected Topics in Signal Processing*, 16(5):1112–1124, 2022.

[118] Jun Wu, Zhifeng Zhang, Yu Hong, and Yonggang Wen. Cloud radio access network (c-ran): a primer. *IEEE network*, 29(1):35–41, 2015.

[119] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 533–548, 2020.

[120] Xincheng Xie, Junyoung Kim, and Kenneth Ross. Gamut: Matrix multiplication-like tasks on gpus. In *14th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS'23)*, 2023.

[121] Qing Yang, Xiaoxiao Li, Hongyi Yao, Ji Fang, Kun Tan, Wenjun Hu, Jiansong Zhang, and Yongguang Zhang. BigStation: enabling scalable real-time signal processingin large mu-mimo systems. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (Sigcomm'13)*, page 399–410, 2013.

[122] Andrew Chi-Chih Yao. New algorithms for bin packing. *Journal of the ACM (JACM)*, 27(2):207–227, 1980.

[123] Taesang Yoo and Andrea Goldsmith. Optimality of zero-forcing beamforming with multiuser diversity. In *IEEE International Conference on Communications (ICC)*, volume 1, pages 542–546, 2005.

[124] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM'20)*, pages 283–295, 2020.

[125] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, pages 658–674, 2023.

[126] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

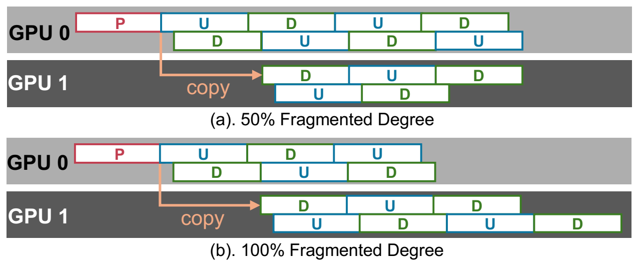


Figure 12: Symbol-level parallelism under fragmented parallelism with different fragment degrees. P/U/D=Pilot/Uplink/Downlink.

Executors	CTAs (#)	Warps (#)	Regs (#)	SMem (Byte)
PilotFFT	128	4	56	8448
UpFFT	128	4	56	8448
iFFT	128	4	56	8448
ZF	4864	32	30	8448
EqDemodul	1216	1	30	0
Modul	1216	1	18	0
Precode	4864	1	30	0
Decode	32	4	168	14144
Encode	32	8	56	843

Table 1: Resource tuples of MegaStation executors in a 128×32 MIMO configuration. We use NVIDIA terminology. CTAs=The number of cooperative thread array (i.e., thread block) per kernel. Warps=The number of warps per CTA. Regs=The number of registers per thread. SMem=Shared memory size per CTA.

A Appendix

A.1 More Fragmented Parallelism Discussion

Under a low fragment degree (Figure 12-a), GPU0 (with the pilot symbol) has more symbols than GPU1 due to the round-robin strategy. GPU0's computation time exceeds the aggregated execution cost of data copying and GPU1's computation. As the fragment degree increases (Figure 12-b), symbol distribution becomes more balanced. GPU0 becomes idle during non-overlapping time intervals when its computing phase interleaves with the data copy and execution phase on GPU1.

A.2 MegaStation Details

Table 1 displays the logical representation of executors. The values for Wraps #, Regs #, and SMem are compiler-specific and remain constant regardless of kernel runtime. However, CTAs # is determined at runtime based on MIMO configuration. MegaStation calculates CTAs # and pass it to the kernel when kernel is launching.

The LROC algorithm described in ALG 1 illustrates the scheduling process in our pipeline scheduler. It starts by prioritizing the most urgent instructions. When there is available space left in the GPU, it continues by reordering instructions whose precedents have been completed. Once the GPU is fully utilized, the algorithm assigns instructions marked as "delayed" to a lower priority stream. Additionally, the algorithm takes advantage of opportunities to combine instructions with dependencies if precedent instructions have already allocated larger GPU resources.

Algorithm 2 LROC Algorithm.

```
1:  procedure LROC_SCHEDULER
2:      min_inst = instQ.min();
3:      while $min\_inst.TS_{slack} - TS_{cur} < T_{thr}$ do                ▷ LSTF
4:          SUBMIT(min_inst, preced_stream(min_inst))
5:          min_inst = instQ.next_min()
6:      for $inst \in instQ$ do                                          ▷ Reorder
7:          if $GPU.avail\_SM \leq 0$ then break
8:          if $pred\_insts.finish$ & $inst.SM \leq GPU.avail\_SM$ then
9:              SUBMIT(inst, pred_insts.stream)
10:     for $inst \in instQ$ do                                          ▷ Over-commit
11:         if $inst.label = delayed$ then
12:             SUBMIT(inst, low_prior_stream)
13:             break
14:     for $inst \in instQ$ do                                          ▷ Coalesce
15:         if $pred\_insts.running$ & $inst.SM \leq pred\_insts.SM$ then
16:             SUBMIT(inst, pred_insts.stream)
```

A.3 More MegaStation Implementation

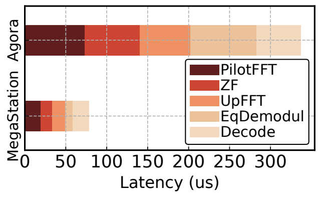
To take advantage of GPU's shared memory and reduce the number of CUDA kernels within instructions, we use tile-based iterators [120] to fuse data transformation into the data loading and storing stage of a kernel computation. In this way, intermediate results are not written back to the global memory and thus data read/write round trips are reduced.

We develop a lightweight mechanism for instruction synchronization. We register and manage a 64KB buffer inside each GPU and map it to host memory. Upon an incoming frame, MegaStation allocates a free array ($sizeof(array) = instruction\#_in_frame \times 4$B) from the buffer, where the CPU actively fetches the array via GDRCopy [27] during the frame execution. Each element in the array is a completion flag of the instruction. When an instruction is finished, we use the CUDA stream operation [20] (`cuStreamWriteValue`) to signal the completion by writing the completion status word to the corresponding place. To enforce the execution order of two dependent instructions in two different CUDA streams, we insert `cuStreamWaitValue` before the second instruction to wait for the completion of the first one without blocking the GPU execution engine.

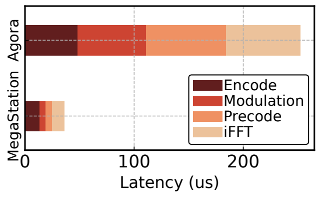
MegaStation employs a master-slave architecture. A master thread realizes the instruction unit, where per-GPU slave threads run pipeline scheduler and track the status of instructions. To minimize allocation overhead, MegaStation reserves memory for every type of operand in each GPU and manages them in a free list on the host side. Similar to the idea of register renaming, our memory manager translates the operand labels to the actual memory pointer before submitting the instruction to GPU. An exception will be triggered when the free list is empty, where we either allocate a new chunk or signal the scheduler to block.

A.4 More MegaStation Evaluation

GPU v.s. CPU Executor. Figure 13 compares the performance between GPU executors (MegaStation) and CPU ex-

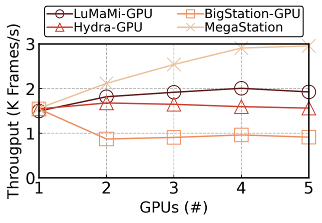


(a) Breakdown on uplink path.

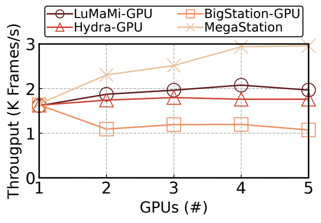


(b) Breakdown on downlink path.

Figure 13: Latency breakdown comparing MegaStation (GPU) and Agora [65] (CPU).

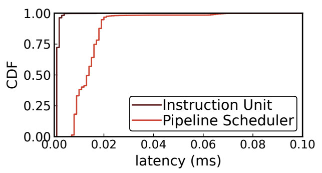


(a) Uplink throughput.

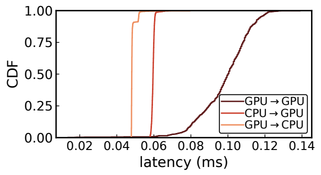


(b) Downlink throughput.

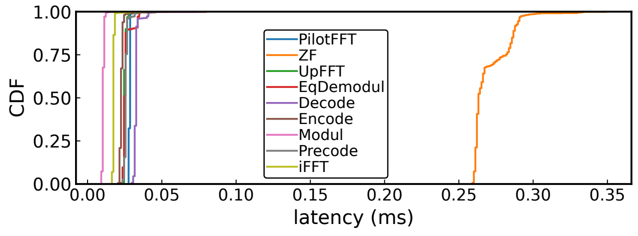
Figure 14: We report the frame throughput as increasing the number of GPUs for 3 RUs (128×64 MIMO).



(a) IU and PS.

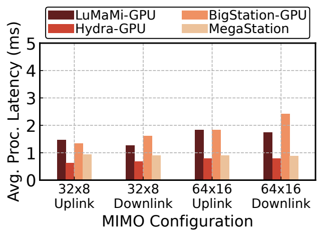


(b) Communications.

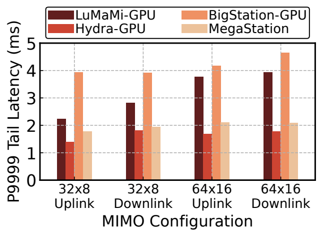


(c) Executors.

Figure 15: Performance of different components in MegaStation under the 128×32 MIMO setting. IU=Instruction Unit. PS=Pipeline Scheduler.



(a) Average Latency.



(b) P9999 Tail Latency.

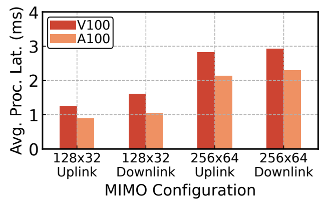
Figure 16: Average/Tail Latency for small MIMOs.

ecutors (Agora [65]).

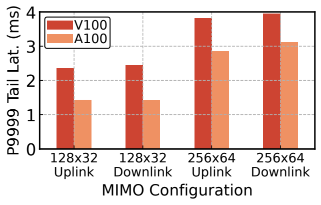
GPU Scaling. Figure 14 explores how MegaStation scales with an increasing number of GPUs.

MegaStation Drill-down. Figure 15 reports the CDF of different system components in MegaStation.

Small MIMOs. We investigate the performance of MegaStation under small MIMO configurations (32×8, 64×16 MIMO). Using the same hardware setup described in §5.2, we depict the average and tail latency of five different methods in Figure 16. Our results show that in contrast to large MIMO scenarios, *Hydra-GPU* performs the best. Compared with MegaStation, it achieves a 20.1%/15.6% improvement on average in average/tail latency due to the lower scheduling overhead of static methods. *Hydra-GPU* outperforms *LuMaMi-GPU*/*BigStation-GPU* because of the finer-grained execution and fewer data synchronizations, respectively.



(a) Average Latency.



(b) P9999 Tail Latency.

Figure 17: Average/Tail Latency on V100/A100.

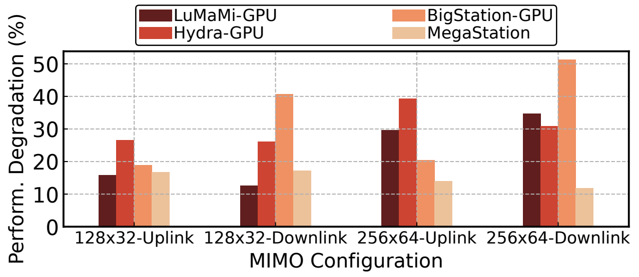
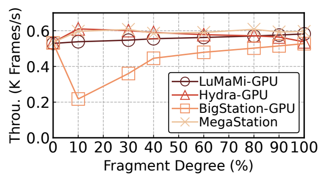
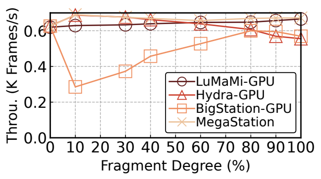


Figure 18: Performance drop on V100+A100 with respect to results on two A100s



(a) Uplink throughput.



(b) Downlink throughput.

Figure 19: Throughput varying the fragment degree.

V100 GPUs. We examine the performance when running MegaStation on different types of GPUs. Specifically, we compare 2 V100 GPUs in a single chassis with two A100 GPUs and report overall average and tail latency. As shown in Figure 17, we observe that the A100 GPUs outperform the V100 GPUs by 27.2%. This is due to higher FLOPS and a greater number of SMs in the A100 GPUs (108 SMs compared to 80 SMs in the V100, which is 25.9% higher).

Tackling Fragmented Parallelism. While developing MegaStation, we find that fragmented parallelism happens frequently. Accurate computing resource accounting and careful scheduling are essential to ensure minimal frame processing deadline violation. We then revisit how MegaStation addresses the fragmented parallelism, where resources are partitioned among multiple GPUs and we should trade off communication overhead with executor distribution.

Similar to the characterization setup (§3.3), we evaluate the performance of these four approaches as the fragment degree increases under 128×64 MIMO with half uplink and half downlink symbols in one frame. As previously discussed, symbol-level execution granularity (*Hydra-GPU*) outperforms frame-level (*LuMaMi-GPU*) at low fragment degrees because data copying is hidden by computation. However, at high fragment degrees, frame-level execution performs better than symbol-level, as data copying cannot be overlapped with computation. Taking the data shuffling overhead into consideration in structural analysis, MegaStation leverages the benefits of both execution granularities, as illustrated in Figure 19. MegaStation employs a top-down approach that first selects a GPU for one frame and then assigns its symbols to that GPU. If the GPU reaches its capacity, MegaStation trade-offs between finding a new GPU (i.e., introducing communication overhead) or waiting for the current one to become capable. By dynamically adapting the execution granularity based on the fragment degree, MegaStation is able to achieve high performance across all fragment degrees.