Agora: Real-time massive MIMO baseband processing in software

Jian Ding
Yale University

Rahman Doost-Mohammady
Rice University

Anuj Kalia
Microsoft

Lin Zhong
Yale University

ABSTRACT

Massive multiple-input multiple-output (MIMO) is a key technology in 5G New Radio (NR) to improve spectral efficiency. A major challenge in its realization is the huge amount of real-time computation required. All existing massive MIMO baseband processing solutions use dedicated and specialized hardware like FPGAs, which can efficiently process baseband data but are expensive, inflexible and difficult to program. In this paper, we show that a software-only system called Agora can handle the high computational demand of real-time massive MIMO baseband processing on a single many-core server. To achieve this goal, we identify the rich dimensions of parallelism in massive MIMO baseband processing, and exploit them across multiple CPU cores. We optimize Agora to best use CPU hardware and software features, including SIMD extensions to accelerate computation, cache optimizations to accelerate data movement, and kernel-bypass packet I/O. We evaluate Agora with up to 64 antennas and show that it meets the data rate and latency requirements of 5G NR.

CCS CONCEPTS

• **Networks** → **Wireless access points, base stations and infrastructure**.

ACM Reference Format:
Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. 2020. Agora: Real-time massive MIMO baseband processing in software. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386367.3431296

1 INTRODUCTION

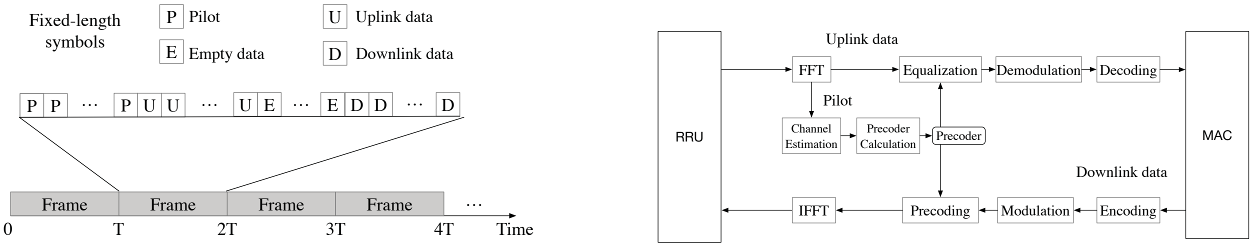
We report the design and implementation of real-time massive MIMO baseband processing in software, called Agora. To our knowledge, Agora is the first software-based realization of massive MIMO baseband processing that is publicly known. By *software*, we mean software running in general-purpose processors; this is different from "software" in software-defined radios (SDRs), which could include any programmable hardware such as FPGA. Agora is an evolutionary step toward a cloud-native mobile network that meets the computational need of mobile networks with a cloud-like infrastructure, instead of dedicated and specialized computing equipment [1, 2].

A mobile network consists of two parts: the core network and the radio-access network (RAN). Today, most of the core network functions have already moved into the cloud, using network function virtualization [3–5]. 5G NR allows flexibly splitting the RAN functionality between the remote radio unit (RRU) at each cell site, and a centralized unit shared by multiple RRUs. This makes it possible to gradually migrate RAN functions away from the RRU [6] and virtualize them in a cloud datacenter. Existing virtualized RAN (vRAN) solutions, typically built on top of Intel's FlexRAN [7] baseband processing software, leave the most computationally-intensive parts of baseband processing at the RRUs, or use specialized and dedicated hardware such as FPGA and ASIC for them.

Agora targets the most ambitious functionality split that implements all digital RAN functions in software at the centralized baseband unit shared by multiple cells. It leaves only the radio-frequency (RF) functions at the RRU, following 5G NR split option 8. While software realizations of baseband processing have been attempted before, e.g., Sora [8] and BigStation [9], Agora is the first to support massive MIMO at a scale required by modern mobile network standards like 5G NR. This includes supporting many more antennas and users, and more computationally-intensive bit error correction schemes like low-density parity check (LDPC) coding. Using 26 cores from a single multi-core server, Agora supports a 64-antenna massive MIMO RRU to serve 16 data streams (or layers), achieving 1.25 Gbps baseband throughput in the uplink with 64-QAM modulation and a 8/9 LDPC code rate.

In realizing Agora, we make the three contributions. First, Agora demonstrates for the first time that massive MIMO *à la* 5G NR with up to 64 RRU antennas is feasible with a single modern many-core server. Second, it contributes a key insight towards addressing the latency and data rate challenges in 5G NR: prioritizing data parallelism within the processing of one frame yields better performance than using pipeline parallelism across multiple frames. This insight distinguishes Agora from prior work such as Sora [8] and BigStation [9], which rely on pipeline parallelism to meet their performance goals. Our microbenchmarks show that this prioritization is highly profitable with modern servers with a large number of cores. Agora exploits the much lower communication cost between cores (than between servers in BigStation) and further conceals such cost behind computation. Third, Agora borrows a queue-based manager-worker threading model from web server design [10, 11] to flexibly control the CPU resource allocation in baseband processing. We identify and

CoNEXT '20, December 1–4, 2020, Barcelona, Spain
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431296



(a) Symbol-based frame structure

(b) Signal processing blocks in massive MIMO baseband

Figure 1: (a) Typical frame structure in time division duplex (TDD) massive MIMO. A frame consists of four types of symbols. (b) The baseband processing uses the *Pilot* symbols from users to estimate channels and compute the *Precoder*, which is then used to process both the *Uplink* and *Downlink* symbols. LDPC *Decoding* and *Encoding* are forward error correction mandated by 5G standards.

evaluate new performance optimizations for massive MIMO baseband processing software. The optimizations include cache-aware optimizations that lower the overhead of inter-core communication, code vectorization with SIMD (Single Instruction, Multiple Data) instructions, and fast ways to use matrix libraries.

We evaluate Agora with two complementary methods. First, we use a fast software-based workload generator that emulates different RRU configurations. We show that Agora successfully meets the data rate and latency requirements of 5G NR. Compared to a pipeline-parallel variant of Agora that we implement, our data parallel design achieves around 30% lower latency. Second, we experimentally analyze the performance and scalability bottlenecks of Agora. We find that, not surprisingly, LDPC decoding contributes to almost half the total processing time. This points to future work on accelerating LDPC decoding by parallelizing it or using specialized hardware such as FPGAs and GPUs. As the number of RRU antennas and the number of MIMO layers increases, overheads of inter-core data communication and synchronization contribute a growing fraction of Agora's processing time. This points to future work on designing a smart scheduler that automatically balances computation and data communication, and NUMA-aware concurrent data structures. We also present experimental results with Agora running in real-time with a 64-antenna massive-MIMO RRU.

Agora is open-source [12].

2 BACKGROUND

To improve spectral efficiency via higher spatial reuse, modern wireless standards employ a technology called multi-user MIMO (MU-MIMO). In MU-MIMO, an RRU with M antennas can concurrently serve K ($M \geq K$) users. We term this configuration $M \times K$ MIMO. The RRU uses *precoding* to realize this spatial multiplexing. Let $\mathbf{x}$, a $K \times 1$ vector, denotes the data streams intended for the K users. In linear precoding, the M RRU antennas send out an $M \times 1$ vector $\mathbf{y}$ derived from a linear transformation of $\mathbf{x}$: $\mathbf{y} = \mathbf{W}\mathbf{x}$, where $\mathbf{W}$ is an $M \times K$ matrix, called *precoder*.

We use linear precoding methods, the only ones considered practical for modern wireless standards (e.g., 5G, 4G/LTE and 802.11). Linear precoding methods, specifically the zero-forcing method adopted by Agora, are known to approach the capacity of non-linear methods when $M \gg K$. Others have considered non-linear precoding methods using specialized or unconventional hardware [13, 14].

The base station computes the precoder using the channel state information (CSI) between all pairs of base station and user antennas, represented by a $M \times K$ matrix, $\mathbf{H}$. Zero-forcing (ZF) is a widely-used precoding technique that aims to ensure that each user receives only its intended data stream, eliminating inter-user interference. The zero-forcing precoder is computed using a pseudo-inverse of H, $\mathbf{W}_{zf} = c \cdot \mathbf{H}^* \left(\mathbf{H}^{\mathrm{T}}\mathbf{H}^*\right)^{-1}$, where c is a constant factor to ensure no antenna exceeds the maximum allowable transmission power. Computing ZF requires matrix inversion and multiplication, so the computational complexity is $O(M \times K^2)$.

Because the spectral efficiency (and the cell capacity) gain improves according to $min(M, K)$, there is a strong incentive to scale up the RRU to tens or even hundreds of antennas, a technology known for 5G as *massive MIMO*. However, supporting large numbers of antennas or users in massive MIMO is challenging because of the increased computational demand. For example, the base station must compute the precoder within the channel coherence time (the time for which the channel is approximately constant), which can be as short as a few milliseconds for mobile users [15]. To address this challenge, real-time massive MIMO systems rely on specialized hardware such as FPGAs. The only reported software implementation, BigStation, supports only 12 RRU antennas [9].

Baseband processing, often known as the physical layer, exists between radios and the MAC; it converts time-domain IQ samples received from the radios to bits usable by the MAC and vice versa. Modern wireless systems employ Orthogonal Frequency-Division Multiplexing (OFDM) modulation, which divides a large frequency band (e.g., tens of MHz) into up to thousands of narrow subcarriers (e.g., tens of KHz). We denote the number of subcarriers by Q.

Baseband processing handles data structured in *frames* as shown in Figure 1(a). User mobility, and its resulting channel coherence time, determines the frame length, which can range from 100s down to a few milliseconds. A frame consists of tens of OFDM *symbols* of equal duration (~71 µs each). At the beginning of a frame, users send orthogonal pilots, interleaved either in time or frequency to allow the base station to estimate their channels represented by matrix $\mathbf{H}$. In each data symbol, all users concurrently transmit or receive one block of modulated data bits (e.g., 4 bits for 16-QAM) on each OFDM subcarrier allocated to them.

Figure 1(b) shows the signal processing for each symbol type. A frame usually starts with uplink pilot symbols, from which the base station obtains the CSI matrix and computes its precoder. We use

Table 1: Computational complexity of baseband processing blocks. M: # antennas, K: # users, Q: # subcarriers per OFDM symbol, L: code block length. A block consists of identical, independent tasks that process disjoint subsets of data in parallel.

Block	# of tasks	**Complexity per task**	**Data parallelism**
FFT	$O(M)$	$O(Q \log Q)$	Antenna
Channel estimation	$O(QM)$	$O(1)$	Subcarrier, antenna
Precoder calculation	$O(Q)$	$O(MK^2)$	Subcarrier
Equalization	$O(Q)$	$O(MK)$	Subcarrier
Demodulation	$O(QK)$	$O(1)$	Subcarrier, user
Decoding	$O(K)$	$O(L)$	User
Encoding	$O(K)$	$O(L)$	User
Modulation	$O(QK)$	$O(1)$	Subcarrier, user
Precoding	$O(Q)$	$O(MK)$	Subcarrier
IFFT	$O(M)$	$O(Q \log Q)$	Antenna

this precoder for both equalization, i.e., demultiplexing user data from uplink symbols received by M RRU antennas, and precoding, i.e., downlink beamforming.

It is challenging to realize massive MIMO baseband processing in software because of its high computation and data rate demands. Two more factors make the challenge even trickier. First, the signal processing blocks differ significantly in their compute time. Table 1 lists the computational complexity of each block. In our setup using 2.1 GHz Intel Xeon Gold 6130 processors, LDPC decoding is the dominant block. For example, decoding one code block with 8448 information bits, 1/3 code rate and 8 iterations takes ~300 µs. Precoder calculation is a distant second thanks to its use of matrix inversion that has a complexity of $O(M \cdot K^2)$; inverting one matrix for $M = 64$ and $K = 16$ takes ~20 µs. In comparison, the time taken for modulation, demodulation, FFT, and IFFT is almost negligible.

Second, each block is data-parallel in a different way, as shown in Table 1. For example, the *FFT* block is antenna-parallel because it processes data from each antenna independently. A block can be viewed as a collection of identical, independent *tasks*, each processing a disjoint set of data. A task in one block may need the output from all the parallel tasks in the previous block, creating a synchronization barrier. For example, the precoder calculation for a subcarrier needs data from channel estimation performed on all antennas. Agora effectively exploits these patterns toward its advantage as will be discussed in §3.

Pipeline parallelism and data parallelism: Prior software-based baseband processing, including BigStation [9], Sora [8] and Atomix [16], extensively exploit pipeline parallelism, in which processing for one baseband processing block overlaps with that for other blocks. Pipeline parallelism can improve throughput but not latency. The large numbers of antennas, subcarriers, and users in massive MIMO bring abundant data parallelism, which is key to reduce latency. For example, the RAN can perform precoding for different OFDM subcarriers, or decoding for different users in parallel. At the hardware level, Agora exploits data parallelism by using multiple cores, as well as multiple SIMD lanes within a core.

Performance metrics: 5G imposes demanding requirements on the latency and the data rate a base station must satisfy while serving its users. Here latency refers to the one-way transmit time between a user and the base station which includes the processing time at the base station. 5G requires latency to be less than 1 ms for ultra-reliable and low-latency communications (URLLC) and 4 ms for enhanced mobile broadband (eMBB) [17, 18]. Data rate, measured in information bits per second communicated between a base station and its users, has different targets for different frequency bands. For sub-6 GHz, which has a maximum bandwidth of 100 MHz, the minimum required downlink peak spectral efficiency of 30 bit/s/Hz [17, 18] corresponds to a peak data rate of 3 Gbps.

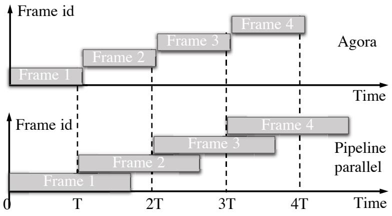


Figure 2: Frame processing schedule in Agora and the alternative pipeline-parallel design.

3 DESIGN

We design Agora to meet the challenging latency and data rate requirements imposed by 5G NR. Prior designs for software baseband processing are inadequate because they target either single-antenna wireless systems [8, 19], or small-scale MIMO systems in less demanding wireless standards like 4G [9].

3.1 Data parallelism and pipeline parallelism

The key design principle in Agora is to use all available CPU cores for the earliest available frame, thus minimizing its processing time. We call this approach "data parallel", since it favors data parallelism available within a frame's processing whenever possible.

In contrast, prior baseband processing systems such as BigStation [9] and Atomix [16] prioritize pipeline parallelism over data parallelism. Pipeline parallelism allows processing multiple frames at the same time. Our insight is that for single-machine systems like Agora, the data-parallel approach has fundamentally lower latency than a pipeline-parallel approach. This is because the latter approach shares CPU cores among the processing of multiple frames, and therefore has higher per-frame latency. Figure 2 shows the high-level difference between the two approaches. A mathematical explanation of this can be found in [20] (Chapter 3.4.2).

Pipeline parallelism was necessary in the past baseband processing systems such as BigStation and Atomix due to fundamental hardware architecture constraints that our target platform, i.e., today's many-core servers, does not have. BigStation requires multiple servers for baseband processing since it uses less powerful hardware from 2013. In such a distributed design, pipeline parallelism is crucial for hiding the high overhead of inter-server communication based on heavyweight OS network stacks. In contrast, Agora uses only intra-server communication for baseband processing, which has much lower overhead than inter-server communication. Atomix uses pipeline parallelism in part because their target hardware architecture (i.e., multi-processor DSPs) lacks coherent shared memory necessary for data parallelism. In contrast, commodity servers support high-speed shared memory.

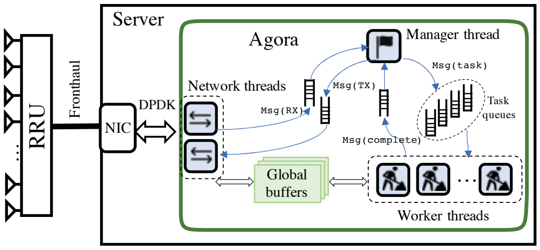


Figure 3: System overview: Agora implements massive MIMO baseband processing on a single many-core server receiving IQ samples from a remote radio unit (RRU) via a fronthaul link. Agora's threads synchronize through FIFO queues using 64-byte messages each containing two fields: task type and buffer location. Agora's threads exchange intermediate results via global shared memory buffers.

Another important drawback of a pipeline parallel design for a single-machine baseband processing system is the additional complexity in core allocation: the number of cores assigned to each block must be computed separately based on the block's computational demands. In Agora, however, we only need to determine the total number of cores required for all blocks.

3.2 Design overview

We now present a high-level overview of Agora's design, shown in Figure 3. Agora uses a manager-worker model with one manager thread and many worker threads, which communicate via message queues. We dedicate a configurable number of worker threads for network I/O. To reduce context switches, Agora pins each thread to a dedicated physical core. Therefore, we use the terms thread, worker and core interchangeably. Although the manager-worker model has been shown to be effective in other domains, e.g., web servers [10, 11], Agora is the first to apply it to software-based baseband processing.

The basic unit of work in Agora is a *task*, which roughly follows the task definition in § 2. A type of task implements a baseband processing block as shown in Figure 1(b). A worker thread serves one task at a time. At any given moment, Agora executes a large number of tasks of the same type, each operating on disjoint data in parallel. Each worker thread handles all types of tasks except network I/O, for which we use dedicated threads.

The manager thread communicates with the worker threads via lock-free shared memory queues using 64-byte messages that fit in one cache line to minimize inter-core communication. The manager sends a message when it creates a task, i.e., `Msg(task)` in Figure 3; a worker sends one when it completes a task, i.e., `Msg(complete)`. A message contains the task type, and an offset indicating the address of the task's input buffer in shared memory. The manager and network threads similarly exchange messages, i.e., `Msg(TX)` and `Msg(RX)`, via a pair of lock-free queues.

Worker threads exchange intermediate results using a set of shared memory buffers. Workers access these buffers without locking, using non-temporal stores to improve performance when possible. We provision sufficient shared memory buffer space for tens of frames to handle performance jitter.

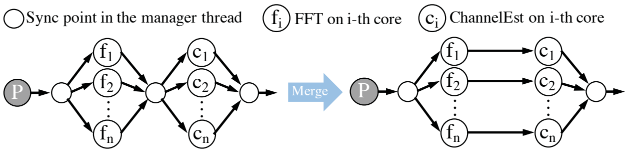

Sync point in the manager thread
f_i FFT on i-th core
c_i ChannelEst on i-th core
P
f_1
f_2
f_n
c_1
c_2
c_n
Merge

Figure 4: Block fusion in the pilot processing.

3.3 Lock-free message queues

The lock-free queues can be accessed by senders and receivers concurrently [21, 22]. There is a *task queue* for messages of each task type sent by the manager to workers, where the manager is the only sender and the workers are concurrent receivers. Having one queue for every task type makes it easier to control the order of tasks being processed. All workers send their complete messages via a single queue, for which the manager is the sole receiver.

The worker threads run an infinite loop busy polling (and dequeuing) messages from the task queues. Agora statically determines the order that workers should poll the task queues, i.e., the priorities of tasks based on their types. The order is determined based on an analysis of the baseband processing as illustrated in Figure 1. Notably, at any given time, all tasks in the queues belong to the same frame as the workers try to get the current frame processed as fast as possible.

3.4 Scheduling optimization

Agora's manager thread implements our scheduling policy of getting all (non-network) worker threads to process the earliest available symbol from the earliest available frame. Our implementation of this policy includes the following two optimizations.

Batching. Some blocks in massive MU-MIMO baseband processing such as FFT and demodulation require little computation. For such blocks, the manager assigns multiple tasks to a worker in one message to reduce the overhead of inter-core messaging. For example, consider the FFT block in which one task processes IQ samples from a single antenna. Agora's manager assigns a worker a batch of N FFT tasks per message, reducing manager-worker messages by a factor of N. We empirically determine the batch size N based on the FFT task's execution time and the cost of inter-core communication. In our setup, $N = 2$ provides the best performance.

Block fusion. Some consecutive baseband processing blocks are parallel in the same dimension. For example, both FFT and channel estimation are antenna-parallel; and demodulation and equalization are subcarrier-parallel. Agora fuses these pairs of blocks into larger blocks to reduce inter-core communication. Figure 4 shows an example of block fusion in pilot symbol processing. The uplink and downlink blocks after fusion are listed in Table 2.

3.4.1 *Leveraging pipeline parallelism.*

While Agora focuses on data parallelism and dedicates all available CPU cycles to the oldest frame when necessary, we do find two occasions where pipeline parallelism can be additionally exploited.

Intra-frame pipeline parallelism. For some blocks, their limited number of tasks does not allow their processing to be distributed to all available cores. When processing these blocks, we leverage pipeline parallelism *within* a frame to let other blocks that satisfy data dependency to be processed simultaneously, which helps improve

Table 2: Concurrency and block fusion in Agora.

Block	Parallelism dimension	Fused blocks
FFT (UL)	Antennas	Channel estimation
Precoder calculation (UL)	Subcarriers	
Demodulation (UL)	Subcarriers	Equalization
Decoding (UL)	Users	
Encoding (DL)	Users	
Precoding (DL)	Subcarrier	Modulation
IFFT (DL)	Antennas	

CPU utilization. For example, the number of LDPC decoding tasks is constrained by the number of users, which is typically up to 16 in the state-of-the-art massive MIMO systems. We therefore let the idle cores take any other available tasks, e.g., FFT, in the future symbols of the same frame.

Inter-frame pipeline parallelism. As mentioned in § 2, the last block in massive MIMO baseband processing, i.e., LDPC decoding, is the most computationally demanding one. When the decoding task queues have fewer tasks left than the number of workers, some of the workers will be idle. Agora allows these idle workers to start processing tasks from the next frame, before the current frame is completely finished.

3.4.2 Reducing RRU idle time in the downlink. A downlink symbol has to be sent to the RRU before it starts downlink transmission. In a TDD system, this means the RRU's transmission needs to wait until it finishes computing the downlink precoder from the current frame's pilots and processing a downlink data symbol, during which the RRU is wasting its air time. To bridge this idle time, Agora lets the current frame process downlink data symbols not only for the current frame, but also for the next frame, which means the first few data symbols in the next frame can be sent to the RRU before the next frame's precoder is ready, thus making it possible to fully utilize the air time of RRU. The negative impact of this approach is that the next frame's first few data symbols are using a slightly out-of-date precoder. However, we expect this negative impact to be small when estimating CSI frequently, e.g., every 1 ms, and user mobility is low, e.g., pedestrian mobility.

4 IMPLEMENTATION

We implemented Agora in C++ and C for Linux. It includes 20K source lines of code, and runs as a userspace application. Agora uses Intel FlexRAN's publicly available LDPC libraries [7] for encoding and decoding, which implement an offset min-sum belief propagation (BP) based decoding algorithm [23]. It uses Intel's Math Kernel Library (MKL) [24] for matrix operations, and uses AVX-512 SIMD instructions for optimizing data type conversions (e.g., for converting integer IQ samples to floats), demodulation, and matrix transposes. Agora supports both AVX2 and AVX-512 machines, and requires machines to have at least AVX2 support.

Agora supports configuring several cellular parameters, including the number of OFDM subcarriers, wireless bandwidth, modulation order, frame length, and LDPC code rate. Our current implementation supports only up to one code block per symbol.

4.1 Memory optimization

Agora employs a host of techniques to improve the performance of the memory subsystem, especially the effectiveness of CPU caches. These techniques are generally applicable to commodity x86 servers with 64-byte cache lines.

Reducing sharing. To avoid expensive locks for concurrency control, all tasks in a block operate on disjoint subsets of data. This allows workers to handle tasks without locking. We also pad buffers to cache line size to avoid false sharing.

Improving memory access efficiency. Recall that a block in Agora consists of many parallel tasks (Table 1). When a block is parallel in a different way from its downstream block, the output of a task in the upstream block is used by all the tasks in the downstream block. We optimize the communication of task outputs to task inputs to improve memory access efficiency.

Consider the following example. In the uplink, the FFT block is antenna-parallel, and the subsequent demodulation block is subcarrier-parallel. A worker that executes an FFT task takes data from one antenna as input, and generates a contiguous array of 8-byte frequency-domain samples, one per subcarrier. If a demodulation task handles only one subcarrier, the CPU core running this task must read an entire cache line (64 bytes) of samples to access the required subcarrier's sample, within which only 8 bytes are useful and the remaining 56 bytes are wasted. We eliminate this waste by implementing demodulation tasks to process eight consecutive subcarriers (§ 3.4), thereby using all 64 bytes of the retrieved cache line. Agora applies this idea to all blocks when possible.

Non-temporal stores. Consider two consecutive baseband processing blocks X and Y are parallel in different dimensions. A task for block Y uses the outputs of block X's tasks as inputs. These outputs were collectively generated by all cores in the system. Without any optimizations, this memory access pattern results in a high cache coherence traffic. We find that it is often beneficial to use non-temporal SIMD stores to write the outputs of block X's tasks directly to DRAM. Although this technique increases pressure on the memory bus, it is faster than incurring high cache coherence traffic. We show in § 6.3 that using non-temporal stores reduces Agora's frame-processing latency by 11%.

4.2 Matrix optimizations

Pesudo-inverse. MIMO equalization and precoding require computing channel matrix pseudo-inverses, $\mathbf{W} = c \cdot \mathbf{H}^* \left(\mathbf{H}^{\mathrm{T}}\mathbf{H}^*\right)^{-1}$. The high CPU cost of matrix inversion has been considered a key cause for the computationally-intensive nature of massive MIMO baseband processing. Interestingly, we find that matrix inversion is a relatively cheap component. In massive MIMO, the number of antennas M is much larger than the number of users K; typical deployments aim to support 16–32 users with 64–128 antennas. Therefore, we need to invert a relatively small $K \times K$ matrix ($\mathbf{H}^{\mathrm{T}}\mathbf{H}^*$). With one AVX-512–capable CPU core, we find that computing $\mathbf{W}$ takes only 15.8 µs for our target use case of $M = 64$ and $K = 16$.

Matrix libraries such as Intel MKL support computing a numerically-robust pseudo-inverse for high condition number channel matrices via a singular-value decomposition (SVD). We find that this approach is roughly an order of magnitude slower than computing $\mathbf{W}$ by directly inverting the inner square matrix, taking 135 µs for

$M = 64$ and $K = 16$. The added robustness is unneeded in practice because we need channel matrices with low condition numbers for effective beamforming with zero-forcing. In ill-conditioned channels, usually attributed to low SNR regime or highly correlated user channels, zero-forcing is not the best linear precoding method in terms of achievable rate and a lower overhead method such as conjugate beamforming may perform better [25].

Matrix multiplication. Equalization and precoding require matrix multiplication for a large amount of data (i.e., all data subcarriers and all data symbols), resulting in a high computation cost. We find that the performance of these multiplications improves drastically with just-in-time (JIT) optimization of the matrix multiplication kernel, which generates specially optimized code for the given problem size. In our experiments, enabling JIT code generation in Intel MKL [26] accelerates matrix multiplication by 3–5x for small matrix sizes, including our target use case of $M = 64$ and $K = 16$.

4.3 Server configuration

We run Agora as a real-time process to reduce expensive OS context switches. Real-time processes are not preempted by normal Linux processes because they have higher priorities. In addition, we isolate Agora's cores from OS interrupts. We disable Turbo Boost, Hyper Threading, and CPU idle states to reduce performance variance.

We use DPDK to bypass Linux's heavyweight network stack, with NIC steering rules to direct received packets to Agora's network threads. We dedicate two threads for network I/O, which is the minimum number required to handle the 44.5 Gbps of fronthaul traffic rate from 64 RRU antennas. We use 9000-byte jumbo Ethernet frames to avoid packet fragmentation.

5 EVALUATION SETUPS

We evaluate Agora on a many-core server with two RRU setups. The first setup uses another server to emulate the RRU of a massive MIMO base station in software. This allows us to stress Agora with various RRU configurations without being limited by real RRU hardware. The second setup employs an actual massive MIMO RRU with 64 antennas. In both setups, the RRU sends 24-bit IQ samples to Agora; Agora pads them to be 32-bit before performing computation. Our experiments consider a constant peak load, i.e., all users are always active and all symbols in a frame are in use to carry either pilot or data.

5.1 Server setup

We run Agora on a single many-core server connected to the RRU (emulated or otherwise). The server has four Intel Xeon Gold 6130 CPUs. Each CPU has 16 cores running at 2.1 GHz, and 22 MB of last-level cache. We use a dual-port 40 GbE NIC to connect to the RRU, although our hardware RRU is only capable of 10 Gbps. We use the RDTSC instruction to measure timestamps for performance profiling. For each experiment, we collect data from 8000 frames.

5.2 Emulated RRU

High performance IQ sample generator. We emulate the MIMO RRU with a fast and flexible software-based IQ sample generator running on a second server. We implement it with DPDK for kernel-bypass packet I/O. The generator follows the symbol-based frame structure outlined in §2 to produce or consume time-domain IQ samples. In each symbol duration, the generator uses a set of M UDP packets to send/receive the IQ samples of all M antennas to/from the Agora server (i.e., one UDP packet per antenna). A frame consists of multiple consecutive sets of UDP packets. Each packet consists of a 64-byte header specifying the frame, symbol and antenna indexes, and as many 24-bit IQ samples as the number of OFDM subcarriers. The IQ sample generator uses nanosecond-precision RDTSC timestamps to precisely control the idle time between sets of packets that determines the symbol duration. We can then control the frame length by changing the symbol duration and the number of symbols in a frame. Our measurements show that the deviation between measured frame length and desired frame length is less than 1 μs, e.g., for a 5 ms frame with 70 symbols, the average error is 0.2 μs with a standard deviation of 0.26 μs.

Cellular parameters. We use the following 5G NR configuration: 20 MHz bandwidth, 64-QAM modulation order, 2048 subcarriers, of which 1200 carry valid data and the rest are used as guard bands to prevent interference. We emulate channels with additive white Gaussian noise (AWGN) with 25 dB signal-to-noise ratio (SNR). We report Agora's performance with frame length between 1 ms and 5 ms, which allows measuring the channel every 1–5 ms. This is important because even with pedestrian mobility, the channel coherence time for large-scale MIMO is 7 ms according to recent measurement studies [15].

As shown in Figure 1(a), each frame consists of pilot symbols and data symbols, and each symbol has a fixed length. We vary the number of data symbols to change the frame length and use one pilot symbol for all frame lengths. We implement frequency-orthogonal pilots where different users occupy different subcarriers to send pilots. We consider two extreme cases where a frame has only uplink data symbols or has only downlink data symbols, to evaluate uplink and downlink performance. In practice, a frame can have both uplink and downlink symbols in the TDD mode that we operate in.

For LDPC, we use 1/3 code rate and base graph 1, the most computationally demanding configuration supported by Intel FlexRAN, for stress testing. We set our encoded code block size to 6864 bits (LDPC lifting size $Z = 104$), so that each symbol maps to one code block. We run up to five iterations for LDPC decoding.

5.3 Actual massive MIMO RRU

To verify that our implementation works with real cellular hardware, we replace the packet generator in the emulated RRU setup described above with a massive MIMO base station and eight mobile users as shown in Figure 5, which are commercially available from Skylark Wireless [27]. The base station has 64 MIMO antennas and operates at the 3.6 GHz CBRS band, serving as our massive MIMO RRU. Its 10 GbE fronthaul limits our testing to 5 MHz bandwidth, beyond which the traffic between the RRU and Agora's server exceeds 10 Gbps. We evaluate Agora with this hardware setup under indoor line-of-sight (LOS) channels and 17-26 dB SNR.

5.4 Pipeline-parallel comparison

To our knowledge, there is no prior work on software-based massive MIMO baseband processing in the public domain, thus we cannot directly compare Agora against prior designs. BigStation [9],

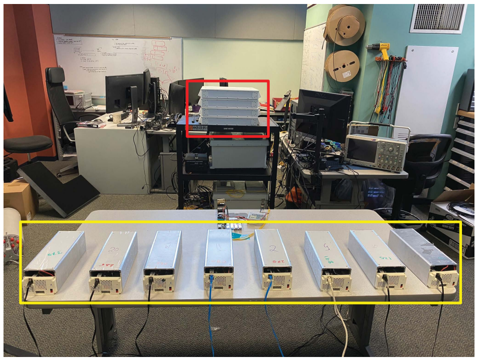


Figure 5: Our RRU hardware setup including a 64-antenna Faros base station, shown in a red box, and eight Iris users, shown in yellow box, both from Skylark Wireless.

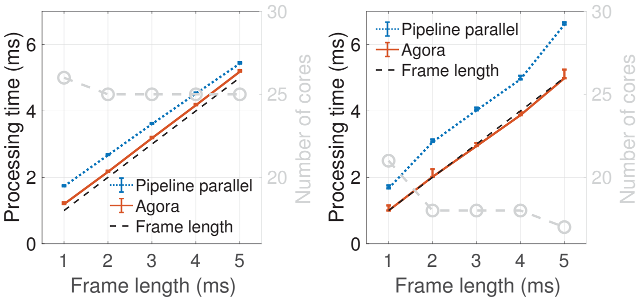
which implements 12-antenna MIMO systems for 802.11 in software running on multiple servers, is the most comparable prior work. However, we are not able to directly compare Agora against it since it is not open-source.

To highlight the benefits of Agora's data-parallel design, we compare it against a pipeline-parallel variant that we implement. The design of this variant is close in spirit to BigStation. BigStation favors pipeline parallelism because it targets a distributed system in which each machine only has only a few (~4) cores. In such a system, the high cost of synchronization over the network prevents intensive use of data parallelism.

We implement the pipeline-parallel variant of Agora with all optimizations that we perform for Agora and test it on the same many-core server. However, unlike Agora where a worker core can process any type of tasks (and blocks), the pipeline-parallel variant statically assigns a fixed, dedicated group of cores to each block in Table 2. Each group of cores exploits the data parallelism within the corresponding block. Non-overlapping groups of cores exploit pipeline parallelism by processing different blocks of baseband processing simultaneously. Given the number of cores, we use a combination of empirical data and mathematical analysis to find the allocation of cores to blocks that minimizes the frame latency. This is made easier by the fact that each block must get enough cores to finish within a frame's time budget and there are only a small number of blocks.

6 RESULTS

We now present our evaluation results, with the following three main takeaways. First, we show that it is feasible to run massive MIMO baseband processing on a single many-core server. Second, we show the advantages of Agora's data-parallel design over its pipeline-parallel variant, achieving lower uplink and downlink processing latency. Third, we show that Agora successfully handles over-the-air traffic from a real hardware RRU. For brevity, we sometimes present results only for the uplink because it is more computationally challenging than the downlink.



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

Figure 6: Median processing latency (with 99.9th percentile as errorbar) and number of cores of Agora and its pipeline-parallel variant over different frame lengths. Both uplink and downlink are based on 64×16 MIMO.

6.1 Feasibility of software massive MIMO

We evaluate Agora's end-to-end performance with our software-based IQ sample generator (§ 5.2) to show that massive MIMO baseband processing is feasible in software. Two metrics are crucial for feasibility. First, Agora must keep up with the frame rate. For example, with a 1 ms frame length, Agora's throughput must be at least one frame per millisecond to avoid dropping frames. Second, Agora must not add excessive latency, i.e., the time Agora takes to complete frame processing must not be much larger than the frame length. For example, to support the 5G NR use case of Enhanced Mobile Broadband (eMBB), the one-way frame processing latency in Agora—measured as the time from which the frame's first packet enters Agora to when we complete LDPC decoding for all users—must be lower than 4 ms [17].

6.1.1 Overall processing latency. We first show that Agora keeps up with the frame rate and latency requirements of eMBB. We measure uplink and downlink processing latency for different frame lengths, relevant to use cases with different channel coherence times. The symbol duration is constant at 71 μs, so there are 14 and 70 symbols per frame for 1 ms and 5 ms frames, respectively. For each frame length, we report the latency with the least number of cores that allows matching the incoming/outgoing IQ sample rate.

Figure 6 shows that Agora keeps up with the IQ sample rate for 64 × 16 MIMO with 26 worker cores for the uplink, and 21 worker cores for the downlink. Uplink baseband processing requires more cores than the downlink due to the high computational overhead of LDPC decoding. Considering that a frame with all uplink symbols is the most computational intensive case, we expect 26 cores to be sufficient to support frames with both uplink and downlink symbols. For both uplink and downlink, Agora's data rate is 454 Mbps and 482 Mbps for 1 ms and 5 ms frames, respectively. Note that these results are for 1/3 code rate, which is the most computationally demanding. With 8/9 code rate, Agora can achieve data rate of 1.25 Gbps and 1.33 Gbps for 1 ms and 5 ms frames, respectively.

Figure 6 also shows that Agora can achieve processing latencies close to frame length for both uplink and downlink. For the uplink, Agora's processing cannot finish before all the packets of a frame

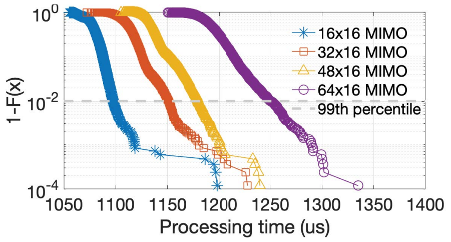


Figure 7: Complementary CDF of Agora's uplink processing time using 1 ms frame and 26 worker cores.

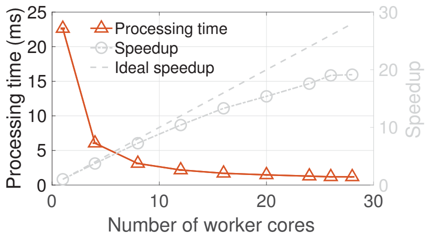


Figure 8: Agora's uplink processing time and speedup for 1 ms frame and 64×16 MIMO.

arrive, resulting in a latency longer than the frame length. For all frame lengths, Agora's average latency is about 180 µs longer than the frame length. For the downlink, the input data comes from the MAC, so its arrival time is not constrained by frame length. Therefore, Agora achieves latencies shorter than frame lengths, which can be further reduced by adding more worker cores.

We also observe that Agora significantly outperforms the pipeline-parallel variant (§ 5.4), especially for the downlink. Agora's superior performance comes from faster zero-forcing, which we discuss in detail in § 6.3.1.

Figure 7 shows the uplink latency distribution for four MIMO configurations, using 1 ms frames, measured from 8000 frames. For 64×16 MIMO, the most expensive configuration, Agora's 99.9th percentile and maximum latencies are 1.29 ms and 1.36 ms, respectively, which meet the 4 ms target of eMBB. Agora achieves low maximum latencies because we direct OS interrupts away from CPU cores used by Agora.

6.1.2 Scalability. Figure 8 shows that Agora is effective in using available cores to reduce its processing latency. As the number of cores increases, the processing latency of a 1 ms frame quickly decreases to below the 4 ms target. The latency stops improving beyond 26 cores, since it is eventually bound by the frame length.

6.1.3 Over-the-air evaluation. We evaluate Agora as the baseband processing system for the 64-antenna RRU described in § 5.3. We program eight clients to send 4 ms frames, with time-orthogonal full-band Zadoff-Chu sequence-based pilots and uplink random data symbols. Each symbol includes 512 OFDM subcarriers with 64-QAM (6-bit) modulation and 300 data subcarriers, corresponding to 1800 bits per symbol per user. The clients use -10 dBm RF transmission power with 6 dB digital power reduction to avoid clipping of data signals with high peak-to-average power ratio. This results in a pilot SNR of 17–26 dB among all 64 antennas at the RRU.

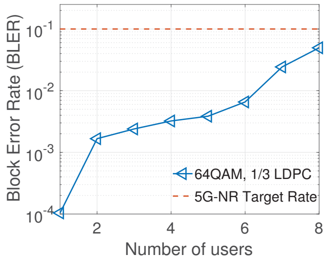


Figure 9: Worst-user block error rate (BLER) vs. number of client uplink streams with Agora and a 64-antenna Faros base station as RRU.

Table 3: Computation cost of uplink baseband processing blocks with 64×16 MIMO, 1 ms frame, and 26 cores. ZF refers to zero-forcing precoder calculation. "Demod" includes computation of equalization and demodulation.

	FFT	ZF	Demod	Decoding
Tasks per frame	896	75	15600	208
Time per task (µs)	2.7±0.09	21.1±0.51	0.19±0.002	46.5±0.25
Batching size	2	3	64	1
Total time across cores (ms)	2.45±0.08	1.59±0.04	2.92±0.03	9.67±0.05

Agora comfortably supports the 64×8 MIMO setup in real-time using *only 7 cores* while the block error rate (BLER) remains below the 10% target rate defined by the 5G NR standard. We measure the BLER as the fraction of uplink user data blocks (each with 1800 bits) for which LDPC decoding fails. We use BLER as an indicator of the achievable baseband processing throughput. Figure 9 shows the worst BLER across users for different numbers of users with 1/3 LDPC code rate. This maps to 16 bit/s/Hz spectral efficiency, i.e., 2 bit/s/Hz per user. Other possible avenues of improvement include handling more client streams, e.g., 16 clients instead of 8, and using higher modulation orders, e.g., 256-QAM.

6.2 Deconstructing performance

We next discuss how various components of Agora contribute to its latency, in order to better focus optimization efforts in future research. We break down Agora's processing time into three components: computation time, which measures the time spent in executing useful baseband processing blocks; data communication time, which measures the overhead of inter-core movement of baseband processing state (e.g., the channel inverse matrix data); and synchronization time, which measures the overhead of inter-core message passing.

6.2.1 Computation time. We first evaluate the computational cost of each fused block listed in Table 2. Table 3 shows these costs for 64×16 MIMO with 26 worker cores and 1 ms frame. For tasks with low per-task cost, Agora processes them in a batch. We set the batch sizes manually based on the task's execution time and inter-core synchronization cost. We use frequency-orthogonal pilots, and perform zero-forcing once for every 16 subcarriers, resulting in a total of 75 tasks ($= \frac{1200}{16}$).

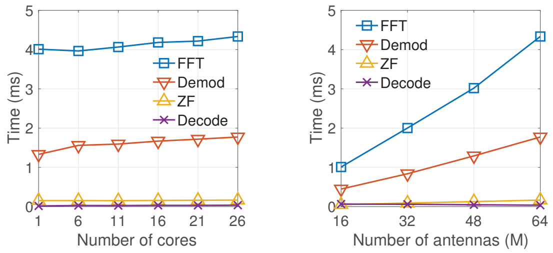


Figure 10: Cumulative data movement time across all cores in different blocks increases as (Left) the # of worker cores increases with 64×16 MIMO, and as (Right) the # of antennas increases, with 16 users and 26 worker CPU cores.

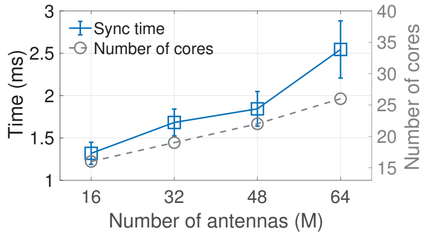


Figure 11: Agora's synchronization overhead for varying numbers of antennas and 16 users. For each antenna configuration, we use the fewest number of cores needed to meet the data rate in uplink processing (shown on the right vertical axis).

We observe that LDPC decoding takes the largest share of Agora's processing time budget. This makes LDPC decoding our primary target of baseband processing optimizations in the future. These optimizations can include software-level optimizations, or using accelerators like FPGAs and GPUs.

In addition, we observe that the computational overhead of individual tasks remains almost constant over the number of cores. However, it changes with the number of antennas (M) and users (K). For example, the number of FFT tasks increases linearly with the number of antennas, but the computation time per task remains largely unchanged. For zero-forcing and demodulation, computation time per task is affected by both M and K since they determine the problem size of matrix operations. For example, with 64×8 MIMO, the execution time per zero-forcing inversion decreases to 10.1 µs from 21.5 µs. The number of LDPC decoding tasks and total LDPC decoding time increase linearly with the number of users.

Finally, we observe that the cumulative time across all 26 cores spent in doing useful baseband processing work is $2.45 + 1.59 + 2.92 + 9.67 = 16.63$ milliseconds, which is less than the 26 ms (one millisecond per core) available budget. The remaining time is spent in inter-core data movement and synchronization. This is the tax we pay for data parallelism, evaluated next.

6.2.2 *Data communication overhead.*

Data parallelism in Agora requires frequent movement of the intermediate results in baseband processing from one core's cache to another core's cache. For example, a CSI matrix requires data from all antennas and all pilots, which is produced as output by multiple worker cores that perform antenna-parallel FFT and CSI estimation. Therefore, for a worker core to compute subcarrier-parallel precoder, it must first fetch the CSI matrix from other cores.

We use the following method to measure the overhead of data communication, with uplink processing as the running example. We run a variation of Agora where we replace baseband processing procedures such as FFT computation and matrix inversion with dummy versions that simply perform the corresponding read or write operations on the memory subsystem. This isolates the data movement overhead from overall processing.

We evaluate how the number of cores and the MIMO size affect Agora's data communication overhead. Figure 10 plots the time spent in data movement combined across all CPU cores for each of the four uplink blocks. In all cases, we observe that FFT and demodulation have a large data communication overhead, which is because these two blocks process the largest amount of data—almost all data received from the network. In contrast, the zero-forcing and decoding tasks have negligible data movement overhead because they process much less data. Zero-forcing runs for only pilot symbols in the frame. Decoding processes data post equalization and demodulation, which reduces the amount of data by ~8x for 64×16 MIMO and 64-QAM modulation.

Impact of number of cores. Figure 10 (Left) shows the data movement overhead for 64×16 MIMO and 1 ms frame as the number of cores increases. We observe that the data movement overhead only increases slightly with more CPU cores, which does not outweigh the benefits of parallelization.

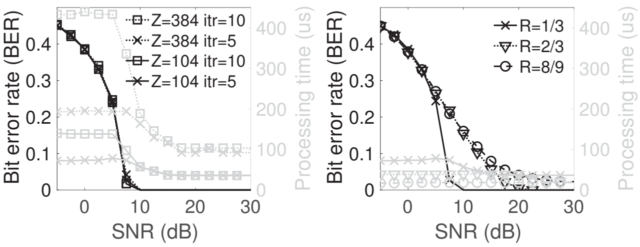
Impact of number of antennas. In Figure 10 (Right), we vary the number of RRU antennas M from 16 to 64, and use 16 users and 26 worker cores for all the experiments. We observe that the overhead of FFT grows almost linearly with M, which is because the number of FFT tasks grows linearly with M while each task accesses a fixed amount of data. The overhead of demodulation also grows linearly with M, but for a different reason: the amount of data each demodulation task accesses grows linearly with M due to the larger matrix size.

6.2.3 *Inter-core synchronization overhead.*

A key concern for the feasibility of massive MIMO baseband processing in software is the overhead of inter-core synchronization. To support data parallelism and our scheduling policy of prioritizing the earliest symbol first, Agora's manager thread must frequently synchronize with worker threads via the shared memory FIFO queues. For example, the manager thread must wait for all FFT tasks of a data symbol and all ZF tasks to complete before scheduling demodulation.

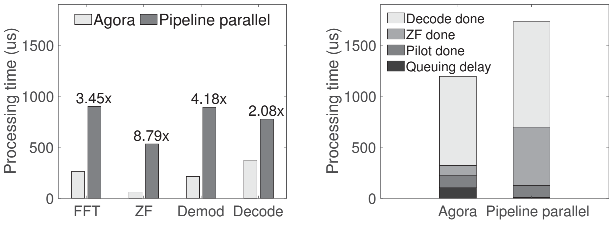
We compute the synchronization overhead by subtracting the time spent in useful computation and data movement from the total time budget, cumulated across all cores. Figure 11 shows that the synchronization overhead grows with more RRU antennas and correspondingly more worker cores. However, even with 64×16 MIMO and 26 worker cores, Agora spends only up to 2.5 ms of its 26 ms budget in synchronization, meaning that the cost of synchronization does not outweigh the benefit of increasing CPU cores.

Combined, data movement and inter-core synchronization account for nearly 8.9 ms of cumulative CPU core time with 26 cores, which is 34% of our 26 ms budget. Reducing this overhead is an



(a) R=1/3 **(b) Z=104, itr=5**

Figure 12: BER and processing time of LDPC decoding change with (a) lifting size (Z) and # of iterations, and (b) code rate (R).



(a) Latency in blocks **(b) Overall latency**

Figure 13: Processing time breakdown for 64×16 MIMO and 1 ms frame using 26 cores.

important avenue of our future research. For example, performing FFT in Agora's network threads can reduce the high data movement overhead of FFT. We are also experimenting with a NUMA-aware work scheduler in the manager to reduce expensive cross-socket data movement and synchronization.

6.2.4 LDPC decoding. Since LDPC decoding is the most expensive block in Agora, here we provide more insights about how LDPC configurations, such as SNR, lifting size (Z), number of iterations and code rate (R), impact bit error rate (BER) and processing time. BER and BLER are inter-dependent, i.e., a lower BER corresponds to a lower BLER. We observe that BLER has a coarse granularity and becomes all ones for 8/9 code rate, so we report BER in Figure 12 for finer granularity. We find that processing time increases linearly over the number of iterations and the lifting size (Z). A lower code rate (R) increases processing time but also reduces BER. A lower SNR leads to significantly higher processing time and BER.

In Figure 12(a), we use $Z = 384$, the maximum lifting size according to 5G NR, and $Z = 104$, the size used in the rest of evaluation, as examples to show the impacts of Z, number of iterations and SNR. Surprisingly, we observe that a smaller Z and fewer iterations do not worsen BER while lowering processing time. Smaller Z values also provide more data parallelism and therefore are favored by Agora. When the SNR is lower than 10 dB, BER drops significantly, which matches the results reported in the literature [28]. In Figure 12(b), we show the impact of code rate with $Z = 104$ and up to 5 iterations. The 1/3 code rate is the most computationally demanding, but also gives the lowest BER, especially in the SNR range of 10–20 dB. This result indicates that under high SNR, Agora can use a higher code rate to reduce LDPC decoding time.

6.3 Effectiveness of optimization

We next investigate the effectiveness of the optimizations described in § 3 and § 4. To demonstrate the importance of following our design principle (§ 3.1) to minimize frame processing latency, we compare Agora's data-parallel oriented design against its pipeline-parallel variant inspired by BigStation's design. We then show the importance of optimizations in minimizing CPU cycles spent on useless work.

6.3.1 Data parallelism vs. pipeline parallelism. Figure 6 already shows that Agora's prioritization of data parallelism over pipeline parallelism is important for its low latency and high efficiency. Figure 13(a) shows the break down of processing time in individual blocks for each design. For the pipeline-parallel variant, we allocate the 26 cores following the core allocation policy in § 5.4 to minimize its latency.

Agora's largest gain comes from precoder calculation (ZF): 8.8 times faster than that of the pipeline-parallel variant. This is because Agora allows any of its worker cores to process ZF while the pipeline-parallel variant only has three cores dedicated for ZF. These three cores, however, are already the maximum number of cores we can allocate to ZF while avoiding frame drops in other blocks. Agora's performance advantage over pipeline parallelism in other blocks is not as significant. The reason is that we dedicate more cores to those blocks to keep up with the frame rate.

To show how the speedup in individual blocks impacts overall latency, we examine important milestones within a frame's processing in Figure 13(b). These include the queuing delay and the milestones of completing pilot processing, ZF and decoding. Queuing delay is the time between when a frame's first packet arrives in Agora to when its processing starts. The other three milestones correspond to the time of finishing processing three blocks, i.e., FFT for pilot symbols, ZF, and LDPC decoding. Due to data dependencies, ZF can not start until all pilots have finished processing; demodulation and decoding can not start until ZF finishes. As expected, we observe Agora's major advantage to be that it finishes ZF much earlier than the pipeline-parallel variant. However, the time between ZF and LDPC decoding completion is not significantly different between the two designs. This is because between the two milestones, multiple blocks, i.e., FFT, demodulation, and LDPC decoding, are processed simultaneously in both designs. The time in pilot processing is about the same in both designs since it is lower bounded by pilot symbol duration. Agora's queuing delay is slightly longer than that of the pipeline-parallel variant because Agora (mostly) waits to start a new frame until it finishes the current one while the pipeline-parallel variant allows the new coming frame to be processed simultaneously with previous frames. However, the small additional queuing delay does not overwhelm Agora's overall advantage.

6.3.2 Importance of performance optimizations. Agora's ability to support real-time massive MIMO baseband processing relies on a combination of optimizations applied for scheduling (§ 3.4), matrix operations (§ 4.2), memory and cache performance (§ 4.1) and server configurations (§ 4.3). Table 4 shows the impact of disabling

Table 4: Effectiveness of optimizations shown by disabling them: median and 99.9th percentile latency for processing 1 ms frame and 64×16 MIMO with 26 worker cores, uplink.

Optimization disabled	Median (ms)	Increase	99.9th (ms)	Increase
Baseline (with all optimizations on)	1.19	-	1.29	-
Batching (§ 3.4)	1.96	1.64x	2.33	1.81x
Memory access optimization (§ 4.1)	1.67	1.40x	1.72	1.33x
Non-temporal store (§ 4.1)	1.34	1.12x	1.47	1.14x
Matrix inverse optimization (§ 4.2)	1.52	1.27x	1.63	1.26x
JIT matrix multiplication (§ 4.2)	1.41	1.18x	1.50	1.16x
Real-time process (§ 4.3)	1.16	0.98x	4.78	3.71x

Table 5: Median and 99.9th percentile latency of Agora for processing 1 ms frame and 64×16 MIMO, uplink.

Server	SIMD support	# worker cores	Median (ms)	99.9th (ms)
2 × Xeon E5-2697 v4, 2.3 GHz	AVX2	32	1.34	1.38
4 × Xeon Gold 6130, 2.1 GHz	AVX-512	26	1.19	1.29
4 × Xeon Gold 6252N, 2.3 GHz	AVX-512	23	1.13	1.19
2 × Xeon Gold 6240, 2.6 GHz	AVX-512	23	1.12	1.15

one of the optimizations on the median and 99.9th percentile latencies. We can see that even disabling only one of the optimizations can significantly increase processing latency, which demonstrates the necessity of applying the combination of all optimizations. It is worth noting that a proper server configuration is also crucial. For example, as shown in Table 4, without running Agora as a real-time process, its tail latency can suffer significantly due to OS context switches.

6.3.3 Impact of hardware. We find Agora's optimizations are effective on servers with recent x86 Xeon processors with AVX2/AVX-512 support, as shown in Table 5, with small adjustments to batch sizes due to difference in memory speed and cache size. The server in the second row is used in the rest of evaluation (see § 5.1). On the servers with AVX-512 support, Agora can achieve similar median and tail latencies using similar number of cores. On the oldest server with only AVX2 support (first row), Agora achieves a slightly longer median latency of 1.34 ms in processing a frame, meeting the latency requirement of 5G NR. However, more cores than available to workers, i.e., 32, are needed to process arriving frames fast enough, i.e., one frame per 1 ms. The results in Table 5 also highlight the effectiveness of AVX-512 for massive MIMO baseband processing.

7 RELATED WORK

Software-based baseband processing. Existing frameworks such as Sora and Ziria [8, 19] demonstrate that it is possible to achieve hardware-comparable baseband processing performance in software. These frameworks target earlier Wi-Fi standards like 802.11a, which do not involve beamforming techniques and therefore only require a small number of CPU cores for data processing. BigStation [9] considers newer LTE and Wi-Fi standards that adopt beamforming and presents a distributed pipeline-parallel architecture for baseband processing of large MU-MIMO. In contrast, Agora uses a data-parallel design running on a single multi-core server, which avoids having to go over the datacenter network for synchronization and data transfers. Open-source projects such as OpenAirInterface [29] and srsLTE [30] also implement baseband processing in software. However, these systems do not support massive MIMO yet. Intel's FlexRAN [31] is a reference design for virtualized RAN (vRAN) and has been used in some vRAN offerings. However, FlexRAN is closed-source, and due to the lack of publicly available information, we are unable to compare it with Agora. FlexRAN's LDPC libraries are available publicly [7]; we use them for LDPC encoding and decoding in Agora.

Some recent projects implement more computationally challenging non-linear precoding for massive MIMO in programmable or novel hardware, e.g., FlexCore with GPUs [13] and QuAMax with the D-Wave quantum annealing computer [14]. Different to these works, Agora targets commodity general-purpose processors and focuses on precoding methods that are already adopted by standards. An early version of Agora is described in [20] as MILLIPEDE.

Real-time massive MIMO systems. Existing massive MIMO base stations rely on specialized hardware to achieve real-time functionality. LuMaMi[32] is an FPGA-based massive MIMO testbed that supports up to 100 antennas and 12 users. LuMaMi uses intensive computation acceleration and PCIe-based data movement to reduce the delay between receiving pilots and starting downlink transmission to only 143 µs (shorter than two symbols). Right now baseband processing in software can be 2× slower. However, there is a large space for further improvements with the programability and flexibility in software, which can reduce the performance gap between software and hardware. For example, Agora uses a stale precoder for part of the downlink symbols (§ 3.4.2) to bridge the processing gap that causes idle time at the RRU.

Latency and throughput optimization. Web and cloud application servers share similar performance goals of high throughput and low latency as Agora, as well as a similar optimization space, e.g., choosing a threading model, and optimizing the network stack. (*i*) The choice of threading model plays a critical role in optimizing latency and throughput for web and cloud application servers. Prior work [10, 11, 33] has extensively studied performance tradeoffs of threading models in different application scenarios. Agora's choice of threading model was informed by these studies, but also took into consideration the unique workload pattern and data dependencies in 5G baseband processing. (*ii*) The authors of [34] summarize the inefficiencies from hardware, OS and application that add to tail latency. Since Agora runs on a similar server environment, these inefficiencies also affect Agora's performance, e.g., interference from background processes can introduce context switch overhead. To eliminate them, Agora adopts common methods as those in prior work, detailed in § 4.3. (*iii*) There have been extensive efforts on reducing latency contributed by the network stack, including kernel bypass such as DPDK [35] and OS optimization such as IX [36], ZygOS [37] and Shinjuku [38]. Agora is orthogonal to them since it mainly focuses on optimizing the part after packets are received in the user space.

8 LIMITATIONS

Software or not? Agora demonstrates the feasibility of implementing modern baseband processing completely in software on a commodity server. However, our work says little about the profitability of a software approach. That is, should a mobile network operator adopt this approach? By running a software implementation in a pay-by-use cloud infrastructure, the operator may reduce its capital expenditure upfront, with cheaper and simpler base stations. However, the impact on its operational expenditure remains uncertain. On the one hand, software implementations with general-purpose processors are known to be less efficient than specialized hardware, by orders of magnitude compared to ASICs [39]. On the other hand, by using general-purpose processors in a shared infrastructure, the operator no longer needs to pay for the time that it is not using, which is not true for implementations with dedicated, specialized hardware seen in traditional base stations. Future work is necessary to settle the question about operational cost. This also highlights the importance of improving the efficiency of Agora via the optimization avenues identified in this paper.

Path towards scaling up. Scaling up to even wider bandwidth (Q), more RRU antennas (M), and more MIMO layers (K) is desirable to improve the cell capacity. This will put pressure on Agora in different ways. For example, more bandwidth and antennas proportionally increase the fronthaul capacity requirement. The fronthaul demand of $M = 64$ and $Q = 100MHz$ (the maximum bandwidth of 5G NR at sub-6 GHz) exceeds even the capacity of 100 GbE NIC. This suggests that FFT/IFFT should be left at the RRU, which can reduce the fronthaul traffic by 45% for 5G NR parameters described in § 5.2, without substantially increasing the base station cost.

Because of the rich parallelism in massive MIMO, adding more cores could be a straightforward path for scaling to larger Q, M and K. For a more challenging case such as $M = 128$ and $K = 64$, we observe that the computation time of zero-forcing grows by ~16x, which would require us to parallelize inversion of a single matrix across cores. The computation time of LDPC decoding grows by 4x due to the increased K. It is promising that a single server can be enough for this more challenging case considering that a single server nowadays can already have over 200 cores, which is ~8x of the 26 cores we currently use for $M = 64$ and $K = 16$. In the next 5–10 years, we expect this number to grow further. We also expect Agora to benefit from new features added into future servers, e.g., the new `bfloat16` support in Intel's Cooper Lake microarchitecture [40] can speed up both computation and data movement in Agora.

However, one roadblock of adding more cores is the increasing inter-core communication cost. As shown in Figure 10 and 11, the inter-core data movement and synchronization cost in Agora grow with M; this can become problematic when M grows further, e.g., 128 or even 256. One way to control this cost is to improve Agora's threading model design. For example, currently, the manager thread places tasks in a task queue that is shared by all worker threads. Moving to per-worker task queues, with the manager making scheduling decisions explicitly and statically can reduce the overhead of parallelization. For larger Q, M and K, we expect both Agora and the pipeline-parallel variant to have higher data movement cost since they both require inter-core data movement when the processing proceeds from one block to another. However, we expect Agora's inter-core synchronization cost to grow faster than its pipeline-parallel variant, which may lead to favoring pipeline parallelism over data parallelism when the number of cores increases to a point that the inter-core synchronization cost overwhelms the benefit of data parallelism.

Separating design from implementation. Massive MIMO systems have many configurable parameters, such as M, K, Q, and the LDPC configurations. Currently, Agora can be considered as manually optimized for a limited subspace of the large configuration space. There is a strong need for Agora to automatically find an optimal or good configuration out of the large space. Achieving this goal requires separating the description or design of a massive MIMO system configuration from its actual implementation. We will need a high-level language that allows the user to specify massive MU-MIMO parameters, and a "compiler" that can automatically find an optimized implementation. Existing studies, such as Ziria [19], Halide [41] and TVM [42], use domain specific languages (DSLs) to separate design from implementation. TVM's compiler can generate optimized code from a large search space for deep learning applications. Ziria's compiler can also produce optimized code for baseband processing. It currently has open-source implementation for 802.11a/g and 4G-LTE release 8. However, it relies on compiler annotations to break the pipeline into threads, hence generating pipeline-parallel code. We will draw inspirations from this literature to design a DSL and an optimizing compiler for Agora.

9 CONCLUSION

This work presents Agora, a software-based framework for real-time baseband processing of massive MIMO on a single many-core server. Evaluation of Agora shows that it achieves significantly lower latency and higher data rate than the state of the art and can scale up to use all available cores effectively. Agora achieves this by maximizing the use of data parallelism for massive MIMO baseband while eschewing pipeline parallelism in the baseband processing. Agora employs a carefully designed threading model to scale to many cores and a series of non-trivial cache-aware optimizations to cope with the memory bottleneck. Our experiments show that Agora is able to support real-time baseband processing for 64 × 16 MIMO with a single many-core server.

Agora was designed and optimized for massive MIMO baseband processing. The primary challenge it addresses is latency; the primary opportunity it exploits is massive data parallelism within a data stream. And it assumes a single many-core machine. We observe the same challenge and opportunity are true for many applications in data analytics and computer perception such as autonomous driving and natural user interfaces. The tight latency requirement can also limit these applications to use a local machine. Therefore, it is our hope that their developers may also find the design, implementation, and optimization ideas from Agora useful.

Acknowledgements

This work was supported in part by NSF Grant CNS 1518916 and NSF/PAWR. Songtao He, Peiyao Zhao, and Caihua Li contributed to a very early version of Agora. Xintong Liu contributed to Agora's AVX2 encoder. Brandon Liu helped with Intel MKL's JIT benchmarks. Douglas Moore helped improve the code quality of Agora. The authors are grateful to the anonymous reviewers and paper shepherd whose input made the final paper better.

REFERENCES

[1] China Mobile Research Institute. C-RAN: the road towards green RAN. *White Paper, ver. 2.5*, 2011.

[2] Mike Wolfe. CommScope definitions: What is C-RAN? https://www.commscope.com/Blog/CommScope-Definitions-What-is-C-RAN/, 2016.

[3] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. A high performance packet core for next generation cellular networks. In *Proc. ACM SigComm*, 2017.

[4] Mehrdad Moradi, Yikai Lin, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. SoftBox: A customizable, low-latency, and scalable 5G core network architecture. *IEEE Journal on Selected Areas in Communications*, 36(3):438–456, 2018.

[5] Mehrdad Moradi, Karthikeyan Sundaresan, Eugene Chai, Sampath Rangarajan, and Z Morley Mao. Skycore: Moving core to the edge for untethered and reliable uav-based lte networks. In *Proc. ACM Int. Conf. Mobile Computing & Networking (MobiCom)*, 2018.

[6] Parallel Wireless. 5G NR logical architecture and its functional splits. https://www.parallelwireless.com/wp-content/uploads/5GFunctionalSplits.pdf.

[7] Intel. FlexRAN LTE and 5G NR FEC software development kit modules. https://software.intel.com/en-us/articles/flexran-lte-and-5g-nr-fec-software-development-kit-modules, 2019.

[8] 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 *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 2009.

[9] Qing Yang, Xiaoxiao Li, Hongyi Yao, Ji Fang, Kun Tan, Wenjun Hu, Jiansong Zhang, and Yongguang Zhang. BigStation: enabling scalable real-time signal processing in large MU-MIMO systems. In *Proc. ACM SigComm*, 2013.

[10] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2001.

[11] Akshitha Sriraman and Thomas F Wenisch. μTune: Auto-tuned threading for OLDI microservices. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2018.

[12] jianding17/Agora. https://github.com/jianding17/Agora.

[13] Christopher Husmann, Georgios Georgis, Konstantinos Nikitopoulos, and Kyle Jamieson. FlexCore: Massively parallel and flexible processing for large {MIMO} access points. In *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 2017.

[14] Minsung Kim, Davide Venturelli, and Kyle Jamieson. Leveraging quantum annealing for large mimo processing in centralized radio access networks. In *Proc. ACM SigComm*, 2019.

[15] Clayton Shepard, Jian Ding, Ryan E Guerra, and Lin Zhong. Understanding real many-antenna mu-mimo channels. In *Proc. IEEE Asilomar Conf. on Signals, Systems and Computers (Asilomar)*, pages 461–467, 2016.

[16] Manu Bansal, Aaron Schulman, and Sachin Katti. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 2015.

[17] 3GPP TR 38.913 v14.3.0. 5G: Study on scenarios and requirements for next generation access technologies (Release 14), October 2017.

[18] ITU-R M.2410-0. Minimum requirements related to technical performance for IMT-2020 radio interface(s), November 2017.

[19] Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agullo. Ziria: A DSL for wireless systems programming. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2015.

[20] Jian Ding. Software-based baseband processing for massive MIMO. Master's thesis, Rice University, August 2019. Available at: https://scholarship.rice.edu/handle/1911/107406.

[21] cameron314/concurrentqueue. ConcurrentQueue. https://github.com/cameron314/concurrentqueue, 2020.

[22] Cameron. A fast general purpose lock-free queue for c++. https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++, 2014.

[23] Jinghu Chen and Marc PC Fossorier. Density evolution for two improved BP-based decoding algorithms of LDPC codes. *IEEE communications letters*, 6, 2002.

[24] Intel. Math kernel library (MKL). https://software.intel.com/en-us/mkl, 2019.

[25] Hong Yang and T. L. Marzetta. Performance of conjugate and zero-forcing beamforming in large-scale antenna systems. *IEEE Journal on Selected Areas in Communications*, 31(2):172–179, February 2013.

[26] Intel. Intel math kernel library improved small matrix performance using just-in-time (JIT) code generation for matrix multiplication (GEMM). https://software.intel.com/content/www/us/en/develop/articles/intel-math-kernel-library-improved-small-matrix-performance-using-just-in-time-jit-code.html, 2018.

[27] Skylark Wireless. https://www.skylarkwireless.com.

[28] Jung Hyun Bae, Ahmed Abotabl, Hsien-Ping Lin, Kee-Bong Song, and Jungwon Lee. An overview of channel coding for 5g nr cellular communications. *APSIPA Transactions on Signal and Information Processing*, 8, 2019.

[29] Navid Nikaein, Mahesh K. Marina, Saravana Manickam, Alex Dawson, Raymond Knopp, and Christian Bonnet. OpenAirInterface: A flexible platform for 5G research. *ACM SIGCOMM Comput. Commun. Rev.*, 44(5):33–38, 2014.

[30] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. srsLTE: An open-source platform for LTE evolution and experimentation. In *Proc. ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization (WiNTECH)*, 2016.

[31] Sujata Tibrewala. The 5G network transformation. https://software.intel.com/en-us/articles/the-5g-network-transformation, 2018.

[32] 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*, 2017.

[33] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proc. USENIX Annual Technical Conf. (ATC)*, 1999.

[34] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proc. ACM Symp. Cloud Computing (SOCC)*, 2014.

[35] Intel. Data plane development kit (DPDK). https://www.dpdk.org/, 2019.

[36] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2014.

[37] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2017.

[38] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 2019.

[39] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proc. Int. Symp. Computer Architecture (ISCA)*, 2010.

[40] Cliff Robinson. 2020 cooper lake socketed with 56 cores and bfloat16. https://www.servethehome.com/2020-cooper-lake-socketed-with-56-cores-and-bfloat16/, 2020.

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN NoticeS*, 48(6), 2013.

[42] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2018.