1. PROJECT OVERVIEW

P2P-RAGCache is an advanced research project that integrates peer-to-peer (P2P) distributed systems with Retrieval-Augmented Generation (RAG) to accelerate Large Language Model (LLM) inference. The project aims to reduce Time-to-First-Token (TTFT) by transitioning Key-Value (KV) cache management from centralized storage to a high-speed, decentralized multi-point parallel loading system using Distributed Hash Tables (DHT).

Research Objectives

- Build a decentralized DHT-based overlay network for distributed KV cache management
- Implement parallel P2P fetching mechanisms to aggregate network bandwidth
- Develop queue-aware latency masking to overlap network I/O with GPU processing
- Implement dynamic RoPE adjustment for position-aware cache stitching
- Optimize system resource allocation across heterogeneous hardware

Expected Impact

This research will contribute to reducing inference latency in RAG systems by up to 10x, enabling real-time long-context retrieval for production LLM deployments. The decentralized architecture also improves fault tolerance and scalability compared to centralized cache storage solutions.

---

2. RESEARCH TIMELINE & MILESTONES

The project is structured into five major milestones, each building upon the previous one. Estimated timeline: 16-20 weeks for a full-time undergraduate researcher.

MILESTONE 1: DHT Ring Overlay Construction (Weeks 1-4)

Objective: Organize distributed inference nodes into a self-organizing DHT ring to enable decentralized chunk-level KV cache lookup.

Tasks:

1. Literature Review: Study DHT protocols (Chord, Pastry, Kademlia) and select the most appropriate for low-latency lookup
2. Design the DHT Architecture: Define node ID assignment, routing table structure, and stabilization protocols
3. Implement Node Join/Leave: Create mechanisms for nodes to join the ring, update finger tables, and handle graceful departures

4. Test Basic Routing: Verify O(log N) lookup performance with simulated network delays
5. Document Network Protocol: Create specifications for message formats and communication patterns

Deliverables:

- Working DHT implementation with at least 10 nodes
- Routing table visualization tool
- Technical report on DHT design decisions and performance benchmarks

Technical Skills Required:

- Distributed systems fundamentals
- Network programming (sockets, TCP/UDP)
- Python or Go for implementation

Weekly Breakdown:

- Week 1: Literature review and protocol selection
- Week 2: Architecture design and basic node implementation
- Week 3: Routing table and lookup logic implementation
- Week 4: Testing, debugging, and documentation

MILESTONE 2: Distributed KV Cache Distribution (Weeks 5-7)

Objective: Develop a mechanism to store precomputed document-chunk KV caches across the DHT, ensuring each node holds specific fragments or replicas in its Leaf Set.

Tasks:

1. Design Cache Partitioning Strategy: Determine how to split KV caches by layer, attention head, or sequence position
2. Implement Leaf Set Replication: Create logic to replicate cache fragments to neighboring nodes for redundancy
3. Develop Cache Serialization: Design efficient binary formats for storing KV tensors (consider compression)
4. Build Cache Storage Layer: Integrate with local storage (SSD/memory) with eviction policies for hot/cold data
5. Test Cache Placement: Verify that caches are distributed evenly and can be located within O(log N) hops

Deliverables:

- Cache distribution protocol implementation

- Storage benchmarks showing read/write latency
- Replication factor analysis (2x, 3x, 4x replicas)

Technical Skills Required:

- Understanding of LLM attention mechanisms and KV cache structure
- Binary serialization (Protocol Buffers, MessagePack, or custom formats)
- Storage system design and optimization

Weekly Breakdown:

- Week 5: Cache partitioning strategy and serialization format design
- Week 6: Leaf Set replication and storage integration
- Week 7: Testing, benchmarking, and optimization

MILESTONE 3: Parallel P2P Prefetching Engine (Weeks 8-11)

Objective: Implement the core logic for multi-point parallel fetching, allowing a node to pull segments of a required KV cache from multiple neighbors simultaneously to maximize aggregate network bandwidth.

Tasks:

1. Design Parallel Fetch Protocol: Define how to request specific cache segments from multiple nodes
2. Implement Segmentation Logic: Split large KV caches into transferable chunks (e.g., by layer or attention head)
3. Build Connection Pool Manager: Maintain concurrent connections to multiple peers with flow control
4. Develop Bandwidth Aggregation: Coordinate parallel transfers to maximize throughput without overwhelming receivers
5. Add Error Handling: Implement retry logic, timeout handling, and fallback to alternative nodes
6. Benchmark Performance: Compare parallel vs. sequential fetching under various network conditions

Deliverables:

- Multi-threaded parallel fetching system
- Performance graphs showing speedup vs. number of parallel sources
- Network utilization analysis under different load conditions

Technical Skills Required:

- Concurrent programming (async/await, threading)
- Network protocols and TCP optimization
- Performance profiling and bottleneck analysis

Weekly Breakdown:

- Week 8: Parallel fetch protocol design and basic implementation
- Week 9: Connection pool manager and flow control
- Week 10: Bandwidth aggregation and error handling
- Week 11: Performance benchmarking and optimization

MILESTONE 4: Queue-Aware Latency Masking (Weeks 12-14)

Objective: Integrate a predictive generator that triggers P2P prefetching during the queue wait time of an inference request, overlapping network I/O with current GPU processing.

Tasks:

1. Implement Request Queue Monitor: Track pending inference requests and estimate wait times
2. Build Prefetch Predictor: Analyze query patterns to predict which chunks will be retrieved
3. Create Asynchronous Prefetch Trigger: Initiate P2P fetching when queue wait time exceeds threshold
4. Integrate with RAG Retriever: Hook into the retrieval pipeline to identify top-k chunks early
5. Optimize Timing: Balance prefetch aggressiveness vs. cache pollution for wrong predictions
6. Measure TTFT Reduction: Compare end-to-end latency with and without queue-aware prefetching

Deliverables:

- Queue-aware prefetch scheduler
- TTFT improvement metrics (target: 50-70% reduction)
- Analysis of prefetch accuracy and cache hit rates

Technical Skills Required:

- Understanding of LLM serving systems (batching, scheduling)
- Asynchronous programming and event-driven architectures
- Performance modeling and queueing theory basics

Weekly Breakdown:

- Week 12: Queue monitor and prefetch predictor implementation
- Week 13: Integration with RAG pipeline and timing optimization
- Week 14: TTFT measurement and performance analysis

MILESTONE 5: Position-Aware Cache Stitching (Weeks 15-18)

Objective: Finalize the logic for dynamically adjusting Rotary Positional Embeddings (RoPE) when fusing retrieved P2P caches to maintain generation quality.

Tasks:

1. Study RoPE Mechanics: Understand how positional information is encoded in KV caches
2. Design RoPE Adjustment Algorithm: Create logic to add relative position offsets when merging caches
3. Implement CUDA Kernel: Write optimized GPU code for on-the-fly RoPE adjustment during cache fusion
4. Build Independent Attention Mask: Implement attention masking to simulate standard prefill behavior
5. Validate Output Quality: Compare generated text quality with and without position adjustment
6. Optimize Performance: Minimize overhead of RoPE adjustment to preserve TTFT gains

Deliverables:

- RoPE adjustment CUDA kernel with benchmarks
- Quality evaluation metrics (perplexity, ROUGE scores)
- End-to-end system integration demonstrating full pipeline

Technical Skills Required:

- Deep learning fundamentals (attention mechanisms, transformers)
- CUDA programming and GPU optimization
- PyTorch or similar deep learning framework

Weekly Breakdown:

- Week 15: RoPE mechanics study and algorithm design
- Week 16: CUDA kernel implementation
- Week 17: Quality validation and testing
- Week 18: End-to-end integration and final optimization

3. RESEARCH METHODOLOGY

3.1 Development Environment Setup

Hardware Requirements:

- At least 3-5 physical or virtual machines for DHT testing
- One GPU node (NVIDIA A100/H100 recommended) for LLM inference
- High-speed networking (10 Gbps minimum, 40 Gbps ideal)
- SSD storage on each node (500GB+ for cache storage)

Software Stack:

- Python 3.10+ with PyTorch 2.0+
- CUDA Toolkit 12.0+ for GPU programming
- Docker or Kubernetes for container orchestration
- Ray or Dask for distributed computing primitives
- Git for version control and collaboration

3.2 Experimental Design

Baseline Systems for Comparison:

- Centralized KV cache on disk (baseline)
- Centralized KV cache in memory
- Standard RAG without KV cache reuse

Evaluation Metrics:

- Time-to-First-Token (TTFT) - primary metric
- End-to-end inference latency
- Network bandwidth utilization
- Cache hit rate and retrieval accuracy
- Generation quality (perplexity, BLEU, ROUGE scores)
- System throughput (queries per second)

Benchmark Datasets:

- Natural Questions (Google) for open-domain QA
- MS MARCO for document retrieval
- Custom long-context datasets (50k+ tokens)

3.3 Testing Strategy

Unit Testing:

- Test each component in isolation (DHT routing, cache serialization, parallel fetching)
- Use mock objects and simulated network conditions

Integration Testing:

- Verify end-to-end pipeline from query to response
- Test fault tolerance (node failures, network partitions)

Performance Testing:

- Conduct stress tests with high query loads
- Measure scalability from 5 to 50 nodes
- Profile CPU, GPU, and network bottlenecks

6. EVALUATION & SUCCESS CRITERIA

6.1 Minimum Viable Product (MVP)

At the end of the research period, the MVP should include:

- A working DHT network with at least 10 nodes demonstrating O(log N) lookup
- Distributed KV cache storage and retrieval functionality
- Parallel P2P fetching showing measurable speedup over sequential fetching
- Basic queue-aware prefetching demonstrating TTFT reduction
- RoPE adjustment implementation maintaining generation quality

6.2 Target Performance Metrics

Success will be evaluated based on these quantitative targets:

Metric	Target
TTFT Reduction	50-70% compared to disk-based baseline
Parallel Fetch Speedup	3-5x with 5 parallel sources
Cache Hit Rate	>85% for frequently accessed chunks
Generation Quality	<2% perplexity increase vs. baseline
System Scalability	Linear throughput scaling up to 20 nodes

6.3 Final Deliverables

1. Complete source code repository with documentation and installation guide

2. Technical report (15-20 pages) describing design, implementation, and experimental results
3. Final presentation (20-30 minutes) for research group or symposium
4. Performance benchmarks and comparison graphs
5. Optional: Conference paper draft for submission to systems or ML conference

---

10. EXPERIMENTAL DESIGN DETAILS

10.1 Baseline Implementations

Centralized Disk-Based (Baseline 1):

- Store all precomputed KV caches on a single server with SSD storage
- Measure TTFT and throughput as baseline

Centralized Memory-Based (Baseline 2):

- Store KV caches in RAM of a single server
- Represents best-case centralized performance

No Cache Reuse (Baseline 3):

- Standard RAG that recomputes KV caches for each query
- Highest quality but worst latency

10.2 Controlled Variables

Keep these constant across experiments:

- Model architecture (e.g., Llama-2-7B)
- Number of retrieved chunks (k=5)
- Network latency simulation (if testing on local cluster)
- Dataset and query distribution

10.3 Independent Variables to Test

- Number of DHT nodes (5, 10, 20, 50)
- Replication factor (2x, 3x, 4x)
- Number of parallel fetch sources (1, 3, 5, 10)
- Prefetch threshold (queue wait time: 10ms, 50ms, 100ms)
- Cache compression level (none, moderate, aggressive)

10.4 Measurement Protocol

For each configuration:

1. Warm up system with 100 queries
2. Run 1000 test queries from benchmark dataset
3. Record TTFT, end-to-end latency, throughput for each query
4. Calculate mean, median, P95, P99 latencies
5. Monitor network bandwidth, CPU, GPU utilization
6. Record cache hit rates and prefetch accuracy
