BitNet Rust Implementation for Apple Silicon

Project TODO & Roadmap

Project Overview

Create a pure Rust implementation of BitNet (1.58-bit quantized neural networks) optimized for Apple M1/M2/M3 chips, leveraging Metal Performance Shaders and unified memory architecture.

Phase 1: Foundation & Research (Weeks 1-2)

✅ Research & Analysis

- [ ] **Study Reference Implementations**
- [ ] Analyze Microsoft BitNet repository structure and algorithms
- [ ] Review MLX framework architecture and Metal integration
- [ ] Examine mlx-bitnet implementation patterns
- [ ] Document key algorithms and data structures
- [ ] **Apple Silicon Architecture Study**
- [ ] Research M1/M2/M3 unified memory architecture
- [ ] Understand Metal Performance Shaders (MPS) capabilities
- [ ] Analyze Neural Engine integration possibilities
- [ ] Study Apple's ML Compute framework
- [ ] **Rust Ecosystem Evaluation**
- [ ] Evaluate candle-rs vs tch vs burn for tensor operations
- [ ] Research metal-rs bindings and capabilities
- [ ] Assess mlx-rs crate maturity and features
- [ ] Compare SIMD libraries (wide, packed_simd)

✅ Project Setup

- [ ] **Repository Structure**
- [ ] Initialize Cargo workspace
- [ ] Set up CI/CD with GitHub Actions
- [ ] Configure benchmarking with criterion
- [ ] Set up documentation with mdbook
- [ ] **Core Dependencies**
- [ ] Add tensor computation crate (candle-core)
- [ ] Integrate Metal bindings (metal-rs)
- [ ] Add MLX Rust bindings (mlx-rs)

- [ ] Set up tokenization (tokenizers)

Phase 2: Core Implementation (Weeks 3-6)

✅ Quantization Engine

- [ ] **1.58-bit Quantization**
- [ ] Implement BitNet quantization algorithm
- [ ] Create weight quantization functions (-1, 0, +1)
- [ ] Implement activation quantization
- [ ] Add dequantization for computation
- [ ] **Quantization Utilities**
- [ ] Weight packing/unpacking functions
- [ ] Quantization-aware training utilities
- [ ] Calibration dataset handling
- [ ] Quantization error analysis tools

✅ BitLinear Layer

- [ ] **Core BitLinear Implementation**
- [ ] Implement BitLinear layer as Module trait
- [ ] Add forward pass computation
- [ ] Implement gradient computation for training
- [ ] Add layer normalization integration
- [ ] **Optimization**
- [ ] Vectorized operations using SIMD
- [ ] Memory layout optimization
- [ ] Batch processing optimization
- [ ] Cache-friendly data structures

✅ Model Architecture

- [ ] **BitNet Model Structure**
- [ ] Implement transformer architecture with BitLinear
- [ ] Add attention mechanism with quantization
- [ ] Implement feed-forward networks
- [ ] Add positional encoding
- [ ] **Model Configuration**
- [ ] Create flexible model configuration system
- [ ] Add model serialization/deserialization

- [ ] Implement model loading from checkpoints
- [ ] Add model validation utilities

Phase 3: Apple Silicon Optimization (Weeks 7-10)

✅ Metal Integration

- [ ] **Metal Compute Shaders**
- [ ] Write Metal shaders for BitLinear operations
- [ ] Implement quantized matrix multiplication kernels
- [ ] Add activation function shaders
- [ ] Create memory-efficient data layouts
- [ ] **Metal Performance Optimization**
- [ ] Optimize threadgroup sizes for M1/M2/M3
- [ ] Implement async compute with command buffers
- [ ] Add memory bandwidth optimization
- [ ] Create GPU/CPU hybrid execution paths

✅ Unified Memory Architecture

- [ ] **Memory Management**
- [ ] Implement zero-copy tensor operations
- [ ] Add unified memory pool management
- [ ] Optimize memory allocation patterns
- [ ] Create memory usage profiling tools
- [ ] **Data Pipeline**
- [ ] Streaming data loading for large models
- [ ] Implement prefetching strategies
- [ ] Add memory-mapped model loading
- [ ] Create efficient batch processing

✅ Apple-Specific Features

- [ ] **Neural Engine Integration**
- [ ] Research ANE capabilities for BitNet
- [ ] Implement ANE fallback paths
- [ ] Add performance comparison tools
- [ ] Create hybrid execution strategies
- [ ] **Performance Monitoring**
- [ ] Add Metal GPU performance counters

- [ ] Implement power consumption monitoring
- [ ] Create thermal throttling detection
- [ ] Add performance profiling dashboard

Phase 4: Inference Engine (Weeks 11-14)

✅ Inference Pipeline

- [ ] **Core Inference Engine**
- [ ] Implement forward pass optimization
- [ ] Add batch inference support
- [ ] Create streaming inference for long sequences
- [ ] Implement KV-cache for transformer models
- [ ] **Generation Features**
- [ ] Add text generation with sampling strategies
- [ ] Implement beam search and nucleus sampling
- [ ] Add temperature and top-k/top-p controls
- [ ] Create generation stopping criteria

✅ Model Serving

- [ ] **Runtime Optimization**
- [ ] Implement model warming strategies
- [ ] Add dynamic batching
- [ ] Create request queuing system
- [ ] Implement load balancing for multi-core
- [ ] **API Interface**
- [ ] Create REST API for inference
- [ ] Add WebSocket support for streaming
- [ ] Implement authentication and rate limiting
- [ ] Create client SDKs

Phase 5: Training & Fine-tuning (Weeks 15-18)

✅ Training Infrastructure

- [ ] **Training Loop**
- [ ] Implement distributed training setup
- [ ] Add gradient accumulation
- [ ] Create checkpointing system

- [ ] Implement learning rate scheduling
- [ ] **Quantization-Aware Training**
- [ ] Add QAT loss functions
- [ ] Implement straight-through estimators
- [ ] Create quantization noise simulation
- [ ] Add quantization regularization

✅ Fine-tuning Capabilities

- [ ] **Parameter-Efficient Fine-tuning**
- [ ] Implement LoRA for BitNet
- [ ] Add adapter modules
- [ ] Create prefix tuning support
- [ ] Implement prompt tuning
- [ ] **Dataset Handling**
- [ ] Add common dataset loaders
- [ ] Implement data preprocessing pipelines
- [ ] Create data augmentation strategies
- [ ] Add validation and testing frameworks

Phase 6: Testing & Validation (Weeks 19-20)

✅ Comprehensive Testing

- [ ] **Unit Tests**
- [ ] Test all quantization functions
- [ ] Validate BitLinear layer correctness
- [ ] Test Metal kernel implementations
- [ ] Verify model loading/saving
- [ ] **Integration Tests**
- [ ] End-to-end inference testing
- [ ] Multi-device testing (M1/M2/M3)
- [ ] Performance regression tests
- [ ] Memory leak detection

✅ Benchmarking

- [ ] **Performance Benchmarks**
- [ ] Compare against reference implementations
- [ ] Benchmark across different Apple chips

- [ ] Measure memory usage patterns
- [ ] Profile inference latency and throughput
- [ ] **Accuracy Validation**
- [ ] Validate against original PyTorch models
- [ ] Test numerical precision
- [ ] Compare quantization quality
- [ ] Validate generation quality

Phase 7: Documentation & Release (Weeks 21-22)

✅ Documentation

- [ ] **Technical Documentation**
- [ ] API documentation with rustdoc
- [ ] Architecture overview
- [ ] Performance tuning guide
- [ ] Troubleshooting guide
- [ ] **User Guides**
- [ ] Quick start tutorial
- [ ] Model conversion guide
- [ ] Fine-tuning tutorial
- [ ] Deployment guide

✅ Release Preparation

- [ ] **Package Management**
- [ ] Prepare crates.io release
- [ ] Create installation scripts
- [ ] Add pre-built binaries
- [ ] Set up package distribution
- [ ] **Community**
- [ ] Create example projects
- [ ] Add contribution guidelines
- [ ] Set up issue templates
- [ ] Create community Discord/forum

Technical Specifications

Target Performance Goals

- **Inference Speed**: >100 tokens/second on M2 Pro
- **Memory Usage**: <4GB RAM for 7B parameter model
- **Quantization**: 1.58-bit weights, 8-bit activations
- **Model Size**: <2GB for 7B parameter BitNet model

Supported Features

- **Models**: BitNet 1.58, BitNet b1.58
- **Tasks**: Text generation, completion, chat
- **Chips**: M1, M1 Pro/Max, M2, M2 Pro/Max, M3 series
- **Formats**: Safetensors, GGUF, custom BitNet format

Key Dependencies

```toml
mlx-rs = "0.25"        # Apple MLX framework bindings
candle-core = "0.8"     # Tensor operations
metal-rs = "0.28"       # Metal GPU programming
tokenizers = "0.15"     # Text tokenization
serde = "1.0"           # Serialization
```

Risk Assessment & Mitigation

Technical Risks

- **MLX Rust Bindings Maturity**: Use candle-metal as fallback
- **Metal Shader Complexity**: Start with compute shaders, optimize iteratively
- **Quantization Accuracy**: Validate against reference implementations
- **Memory Constraints**: Implement streaming and model sharding

Timeline Risks

- **Dependency Issues**: Allocate extra time for toolchain setup
- **Performance Optimization**: Focus on correctness first, optimize second
- **Apple Silicon Variations**: Test on multiple chip generations early

---

Success Metrics

Performance Metrics

- [ ] Inference speed matches or exceeds MLX Python implementation
- [ ] Memory usage <50% of full precision model
- [ ] Model accuracy within 2% of original BitNet
- [ ] Cold start time <5 seconds for 7B model

Quality Metrics

- [ ] 95%+ test coverage
- [ ] Zero memory leaks in continuous operation
- [ ] Comprehensive benchmarking suite
- [ ] Production-ready documentation

Post-Release Roadmap

Short-term (Months 1-3)

- [ ] Community feedback integration
- [ ] Performance optimization based on real-world usage
- [ ] Additional model architecture support
- [ ] Integration with popular inference frameworks

Medium-term (Months 4-6)

- [ ] Multi-modal BitNet support
- [ ] Advanced quantization techniques
- [ ] Edge deployment optimizations
- [ ] Training acceleration features

Long-term (Months 7-12)

- [ ] Research integration with Apple Neural Engine
- [ ] Advanced model compression techniques
- [ ] Distributed inference capabilities
- [ ] Commercial deployment features

This roadmap is a living document and will be updated based on progress, feedback, and new developments in the BitNet and Apple Silicon ecosystems.