Understanding 1

**🧠 Project: On-Device LLM Integration**

**📌 Problem Statement:**

**"[SSNeuralcore-infer] inference pipeline enablement for SWA and token eviction"**

**🧩 1. Understanding the Problem**

This project involves building and optimizing an **LLM inference pipeline** that runs **on-device** using a specific inference engine — likely **SSNeuralcore** (could be Samsung Semiconductor Neural Core).

**✳️ Core Requirements:**

* Enable **Sliding Window Attention (SWA)** in the pipeline → to support **long-context input efficiently**
* Enable **token eviction** → to manage **KV cache** size and reduce memory usage
* Optimize for **low-latency**, **low-memory**, **real-time** inference on constrained devices (e.g., mobile, edge)

**🛠️ 2. Key Concepts to Understand**

**🔹 A. On-Device LLM Inference:**

* Running transformer models on mobile, edge, or embedded devices
* Limitations: CPU/GPU cores, RAM, power consumption

**🔹 B. Sliding Window Attention (SWA):**

* An efficient attention method where tokens only attend to their **local window** (e.g., last 512 tokens)
* Reduces quadratic complexity of attention to **linear time**
* Common in models like **Longformer, BigBird**

**🔹 C. Token Eviction:**

* LLMs store **key-value (KV) caches** during inference
* In long interactions, cache grows → too large for mobile
* Eviction = intelligently removing tokens from cache (e.g., LRU, sliding, distance-based)

**🎯 3. What All To Be Done (Tasks)**

| **Phase** | **Task** |
| --- | --- |
| ✅ Understanding | - Study Transformer inference - Study attention mechanisms (full vs. sliding) - Understand SSNeuralcore-infer API/docs |
| ✅ Design | - Architect a modular inference pipeline - Identify cache management points - Plan window attention mechanism |
| ✅ Implementation | - Implement sliding window attention - Add token eviction policy - Integrate both in SSNeuralcore-infer |
| ✅ Optimization | - Profile memory/latency - Add quantization (optional) - Tune for small models (e.g., TinyLLaMA) |
| ✅ Evaluation | - Compare full attention vs SWA - Benchmark token eviction (FIFO vs. LRU etc.) - Log inference speed, RAM usage, quality |
| ✅ Documentation | - Flow diagrams, memory profiles, logs - Trade-offs and innovation explanations |

**🧑‍🏫 4. What All To Learn**

**🔸 A. LLM Foundations**

* Transformer architecture (attention, MHA, decoder-only vs encoder-decoder)
* Prompt handling and tokenization
* KV caching and autoregressive decoding

**🔸 B. Efficient Attention Mechanisms**

* Sliding Window Attention (SWA)
* Dilated attention, Local Attention
* FlashAttention (optional)

**🔸 C. Cache & Memory Management**

* KV cache structure
* Token eviction strategies (FIFO, LRU, Score-based)
* Memory-constrained inference design

**🔸 D. On-Device Inference Frameworks**

* SSNeuralcore-infer (your target engine)
* Alternatives: ONNX Runtime, Llama.cpp, GGML, TFLite

**🔸 E. Optimization**

* Quantization (INT4, INT8)
* Pruning and distillation
* Static vs. dynamic shape inference

**🌐 5. Existing Solutions (Benchmarks & Open Source)**

| **Tool / Library** | **Description** | **What You Can Learn** |
| --- | --- | --- |
| 🔹 [Llama.cpp](https://github.com/ggerganov/llama.cpp) | CPU/GPU on-device LLM | Token streaming, quantization |
| 🔹 [GGML](https://github.com/ggerganov/ggml) | Quantized CPU inference | Efficient cache handling |
| 🔹 [TensorRT-LLM](https://github.com/NVIDIA/TensorRT-LLM) | GPU-optimized inference engine | Sliding window attention, token streaming |
| 🔹 [Longformer](https://github.com/allenai/longformer) | Local attention transformer | Implementation of SWA |
| 🔹 [FlashAttention](https://github.com/HazyResearch/flash-attention) | Faster softmax attention | Compare with SWA for performance |

**📈 6. How to Improve Existing Solutions**

| **Existing Gap** | **Proposed Improvement** |
| --- | --- |
| 🧠 FIFO eviction is dumb | Use **LRU or importance-based eviction** |
| 💾 KV cache still grows too large | Use **compressed KV cache** (e.g., low-rank, quantized) |
| 🐢 Sliding Window loses global context | Use **hybrid SWA + global token memory** |
| 🧮 Attention is still quadratic in long windows | Use **dilated or clustered attention** |
| 📶 Token latency is linear | Implement **asynchronous prefetching** or **pipeline parallelism** |

**💡 7. Proposing Novel Improvements**

**Step-by-step:**

| **Step** | **Example** |
| --- | --- |
| 1. Identify Limitation | "Sliding window drops early tokens — harms long context memory." |
| 2. Propose Solution | "Introduce memory bank of summary tokens retained beyond sliding window." |
| 3. Justify | "Minimal memory growth, retains long-term context." |
| 4. Implement | Update attention mask to mix sliding + memory |
| 5. Benchmark | Test vs. baseline on inference time and response quality |

**🔧 8. Implementation Blueprint**

**Tools:**

* Language: Python + C++ (or C if required)
* Engine: SSNeuralcore-infer API
* Test Models: TinyLLaMA, DistilGPT, Falcon-RW

**Implementation Steps:**

1. **Load model with tokenizer**
2. **Integrate sliding window attention mask** into transformer decoder
3. **Design KV cache manager**:
   * Max length = N
   * When exceeded, evict tokens using policy (FIFO, LRU, etc.)
4. **Connect to inference loop**:
   * Prompt → tokenize → model → next token
   * Update and manage KV cache
5. **Monitor memory + latency**
6. **Compare with full attention baseline**

**🧪 9. Evaluation Metrics**

| **Metric** | **Why** |
| --- | --- |
| ⏱️ Latency per token | Measure inference speed |
| 🧠 Peak memory usage | Key for mobile deployment |
| 📏 Output quality | Compare token predictions |
| 🔁 Context retention | Does it "forget" early parts? |
| 🔄 Cache hits/misses | Efficiency of eviction policy |

**🗂️ 10. Resources**

**Official Docs:**

* [SSNeuralcore SDK Docs] *(if provided to you)*
* HuggingFace: Transformers Basics
* ONNX Runtime Mobile: <https://onnxruntime.ai/>

**Tutorials:**

* Efficient Attention Overview – Lilian Weng
* How KV Cache works in LLMs
* Sliding Window Attention Explained

**✅ Summary**

You are tasked with **enabling LLM inference on constrained devices** using:

* **Sliding Window Attention** → For efficient context handling
* **Token Eviction** → For cache/memory optimization  
  Using a **custom inference engine (SSNeuralcore-infer)**

This is a **real-world, cutting-edge project** that mirrors how companies like Samsung, Qualcomm, Meta, and Apple are deploying LLMs on phones and wearables.

Would you like me to help you with:

* ✅ A project timeline?
* ✅ Sample code to start with?
* ✅ A slide deck or report draft?
* ✅ LRU token eviction code?

# On-Device LLM Inference Pipeline with Sliding Window Attention and Token Eviction

# This is a simplified reference implementation of a pipeline for integrating an LLM with:

# - Sliding Window Attention (SWA)

# - Token Eviction for KV cache management

#

# This code assumes a pseudo framework similar to SSNeuralcore-infer, and is adaptable

# to custom C++/CUDA backends or lighter frameworks like llama.cpp or ggml.

import torch

import numpy as np

from transformers import AutoTokenizer, AutoModelForCausalLM

from collections import deque

# ------------------------- CONFIG ----------------------------

MODEL\_NAME = "TinyLlama/TinyLlama-1.1B-Chat-v1.0" # Small on-device model

WINDOW\_SIZE = 128 # Sliding window size for attention

CACHE\_LIMIT = 256 # Max tokens to keep in KV cache

DEVICE = "cuda" if torch.cuda.is\_available() else "cpu"

# ------------------------- SETUP -----------------------------

tokenizer = AutoTokenizer.from\_pretrained(MODEL\_NAME)

model = AutoModelForCausalLM.from\_pretrained(MODEL\_NAME)

model.to(DEVICE)

model.eval()

# ------------------------- CACHE MANAGER ----------------------

class TokenEvictionCache:

def \_\_init\_\_(self, limit):

self.cache = deque()

self.limit = limit

def add(self, token\_id):

if len(self.cache) >= self.limit:

self.cache.popleft() # FIFO Eviction

self.cache.append(token\_id)

def get(self):

return list(self.cache)

kv\_cache = TokenEvictionCache(CACHE\_LIMIT)

# ------------------------- SWA MASK CREATION ------------------

def create\_attention\_mask(token\_ids):

# Create attention mask with sliding window

seq\_len = len(token\_ids)

mask = torch.zeros(seq\_len, seq\_len)

for i in range(seq\_len):

start = max(0, i - WINDOW\_SIZE)

mask[i, start:i+1] = 1

return mask

# ------------------------- INFERENCE PIPELINE -----------------

def generate\_response(prompt, max\_new\_tokens=50):

input\_ids = tokenizer.encode(prompt, return\_tensors="pt").to(DEVICE)

for token\_id in input\_ids[0]:

kv\_cache.add(token\_id.item())

generated\_ids = input\_ids

for \_ in range(max\_new\_tokens):

cached\_input = torch.tensor(kv\_cache.get(), dtype=torch.long).unsqueeze(0).to(DEVICE)

attention\_mask = create\_attention\_mask(kv\_cache.get()).to(DEVICE)

with torch.no\_grad():

outputs = model(input\_ids=cached\_input, attention\_mask=attention\_mask)

next\_token\_logits = outputs.logits[:, -1, :]

next\_token\_id = torch.argmax(next\_token\_logits, dim=-1)

kv\_cache.add(next\_token\_id.item())

generated\_ids = torch.cat((generated\_ids, next\_token\_id.unsqueeze(0)), dim=1)

if next\_token\_id.item() == tokenizer.eos\_token\_id:

break

return tokenizer.decode(generated\_ids[0], skip\_special\_tokens=True)

# ------------------------- RUN EXAMPLE ------------------------

if \_\_name\_\_ == "\_\_main\_\_":

user\_prompt = "Explain the benefits of sliding window attention in transformers."

output = generate\_response(user\_prompt)

print("\nResponse:\n", output)

Understanding 2

# On-Device LLM Integration: SSNeuralcore-infer SWA & Token Eviction Project

## Project Overview

**Group**: On-device LLM Integration  
**Problem Statement**: SSNeuralcore-infer inference pipeline enablement for SWA (Sliding Window Attention) and token eviction

This project focuses on implementing memory-efficient attention mechanisms for large language models running on resource-constrained edge devices, specifically targeting the SSNeuralcore inference engine.

## Problem Statement Deep Dive

### Core Challenge

The project addresses the fundamental memory bottleneck in transformer-based LLM inference on edge devices:

1. **Quadratic Memory Growth**: Standard self-attention requires O(n²) memory for sequence length n
2. **KV-Cache Explosion**: Key-Value caches grow linearly with sequence length, quickly exhausting device memory
3. **Real-time Constraints**: Need to maintain low latency while managing limited memory resources

### Technical Context

* **SSNeuralcore-infer**: Samsung's specialized neural inference engine optimized for mobile/edge deployment
* **Sliding Window Attention**: Attention mechanism that only considers a fixed window of recent tokens
* **Token Eviction**: Intelligent removal of tokens from memory when limits are reached
* **Edge Deployment**: Mobile devices, IoT devices, embedded systems with <8GB RAM

### Why This Matters

* **Cost Reduction**: Enables powerful LLM capabilities without cloud dependency
* **Privacy**: On-device processing keeps sensitive data local
* **Latency**: Eliminates network round-trips for real-time applications
* **Reliability**: Works offline without internet connectivity

## What Needs to Be Done

### 1. Sliding Window Attention Implementation

* **Window Size Optimization**: Determine optimal window sizes for different model architectures
* **Attention Pattern Design**: Implement efficient sliding patterns (fixed, exponential decay, learned)
* **Memory Management**: Optimize KV-cache storage and retrieval within windows
* **Quality Preservation**: Minimize information loss from limited attention span

### 2. Intelligent Token Eviction System

* **Eviction Policies**: Develop algorithms beyond simple FIFO/LRU
* **Importance Scoring**: Score tokens based on attention weights, semantic relevance, positional importance
* **Context Preservation**: Maintain critical tokens for task coherence
* **Dynamic Thresholds**: Adaptive memory limits based on available resources

### 3. Pipeline Integration

* **Seamless Integration**: Embed SWA and eviction into existing SSNeuralcore-infer pipeline
* **Configuration System**: Runtime parameters for window sizes, eviction policies
* **Performance Monitoring**: Real-time metrics for memory usage, attention quality
* **Fallback Mechanisms**: Graceful degradation when memory limits are exceeded

### 4. Hardware Optimization

* **NPU Utilization**: Leverage Samsung's neural processing units for attention computation
* **Memory Hierarchy**: Optimize for different memory types (DRAM, cache, registers)
* **Parallel Processing**: Multi-core attention computation with proper synchronization
* **Energy Efficiency**: Minimize power consumption during attention operations

## Key Learning Areas

### 1. Transformer Architecture Deep Dive

* **Self-Attention Mechanisms**: Mathematical foundations, computational complexity
* **Multi-Head Attention**: Parallel attention streams, head pruning strategies
* **Positional Encoding**: Relative vs absolute position, rotary embeddings
* **KV-Cache Management**: Storage formats, compression techniques, access patterns

### 2. Memory-Efficient Attention Variants

* **Sliding Window Attention**: Fixed windows, overlapping windows, hierarchical windows
* **Sparse Attention**: Random, local, global attention patterns
* **Linear Attention**: Approximations that reduce quadratic complexity
* **Flash Attention**: Memory-efficient exact attention computation

### 3. Systems Programming for AI

* **Memory Management**: Custom allocators, memory pools, garbage collection
* **CUDA/OpenCL Programming**: GPU acceleration for attention kernels
* **ARM Optimization**: NEON intrinsics, cache optimization, memory alignment
* **Profiling and Debugging**: Performance analysis, memory leak detection

### 4. Mobile AI Optimization

* **Quantization**: INT8/INT4 attention computation
* **Model Compression**: Attention head pruning, layer distillation
* **Batch Processing**: Dynamic batching for improved throughput
* **Temperature Scaling**: Attention sharpening/smoothing for quality control

## Essential Resources

### Academic Papers

#### Sliding Window Attention

1. **"Longformer: The Long-Document Transformer"** - Sliding window + global attention
2. **"BigBird: Transformers for Longer Sequences"** - Sparse attention patterns including sliding windows
3. **"GPT-Cache: Efficient Context Window Extension"** - Dynamic context management
4. **"Streaming LLM: Efficient Streaming Language Models"** - Infinite sequence handling with sliding windows

#### Memory-Efficient Attention

1. **"FlashAttention: Fast and Memory-Efficient Exact Attention"** - Tiled attention computation
2. **"FlashAttention-2: Faster Attention with Better Parallelism"** - Improved memory efficiency
3. **"Self-attention Does Not Need O(n²) Memory"** - Memory complexity reduction techniques
4. **"Linformer: Self-Attention with Linear Complexity"** - Linear attention approximations

#### Token Management

1. **"Efficient Transformers: A Survey"** - Comprehensive overview of efficiency techniques
2. **"Token Dropping for Efficient BERT Pretraining"** - Dynamic token removal strategies
3. **"AdaDropout: An Adaptive Dropout Framework"** - Adaptive attention token selection
4. **"Scissorhands: Exploiting the Persistence of Importance Hypothesis"** - Structured attention pruning

### Technical Documentation

* **Samsung Neural SDK**: SSNeuralcore programming interfaces
* **ARM Compute Library**: Optimized neural network functions
* **ONNX Runtime**: Cross-platform inference optimization
* **PyTorch Mobile**: Mobile deployment framework
* **TensorFlow Lite**: Lightweight inference engine

### Open Source Implementations

* **Transformers Library (Hugging Face)**: Reference implementations
* **FlashAttention GitHub**: Official FlashAttention implementation
* **xFormers (Meta)**: Memory-efficient attention variants
* **FasterTransformer (NVIDIA)**: Optimized transformer inference
* **LightSeq**: High-performance sequence modeling

### Benchmarking Tools

* **MLPerf Mobile**: Standardized mobile AI benchmarks
* **ONNX Model Zoo**: Pre-trained models for testing
* **Attention Visualization Tools**: Understanding attention patterns
* **Memory Profilers**: Valgrind, AddressSanitizer, custom profilers

## Existing Solutions Analysis

### 1. Current Sliding Window Implementations

#### Longformer Approach

**Strengths:**

* Combines sliding window with global attention tokens
* Linear complexity in sequence length
* Maintains long-range dependencies through global tokens

**Limitations:**

* Still requires significant memory for global tokens
* Complex implementation with multiple attention patterns
* Not optimized for extreme resource constraints

#### BigBird Pattern

**Strengths:**

* Multiple sparse attention patterns (sliding, random, global)
* Theoretical guarantees on expressiveness
* Good performance on long sequences

**Limitations:**

* Complex attention pattern management
* Higher implementation complexity
* Limited mobile optimization

### 2. Token Eviction Strategies

#### Static Approaches

* **FIFO (First In, First Out)**: Simple but loses important context
* **LRU (Least Recently Used)**: Better than FIFO but ignores semantic importance
* **Fixed Window**: Maintains recent context but may lose critical information

#### Dynamic Approaches

* **Attention-Based Eviction**: Remove tokens with low attention scores
* **Gradient-Based Importance**: Use gradient information to score token importance
* **Learned Eviction**: Train small networks to predict token importance

### 3. Commercial Solutions

#### Hardware-Specific Optimizations

* **Apple Neural Engine**: iOS-optimized attention kernels
* **Qualcomm Hexagon DSP**: Snapdragon-specific optimizations
* **Google Edge TPU**: Custom attention acceleration
* **Samsung NPU**: Exynos neural processing optimizations

**Common Limitations:**

* Vendor lock-in and limited flexibility
* Black-box implementations difficult to customize
* May not support latest attention mechanisms
* Limited academic/research access

## Innovation Opportunities

### 1. Adaptive Sliding Window Attention

**Novel Concept**: Dynamic window sizing based on content and context

**Key Innovations:**

* **Content-Aware Windows**: Larger windows for complex content, smaller for simple patterns
* **Task-Specific Adaptation**: Different window strategies for chat vs completion vs summarization
* **Real-time Adjustment**: Modify window size based on available memory and computational budget
* **Hierarchical Windows**: Multi-scale attention with different window sizes per layer

**Implementation Strategy:**

if content\_complexity > threshold:

window\_size = max\_window\_size

elif memory\_pressure > high\_threshold:

window\_size = min\_window\_size

else:

window\_size = adaptive\_size(content, memory\_available)

### 2. Semantic-Aware Token Eviction

**Novel Concept**: Use NLP understanding to make intelligent eviction decisions

**Key Innovations:**

* **Entity Preservation**: Never evict named entities, dates, numbers
* **Coreference Tracking**: Maintain pronouns and their antecedents
* **Sentence Boundary Awareness**: Avoid breaking mid-sentence
* **Topic Coherence**: Preserve tokens related to main conversation topics

**Scoring Function:**

token\_score = (

attention\_weight \* 0.3 +

semantic\_importance \* 0.4 +

positional\_decay \* 0.2 +

entity\_bonus \* 0.1

)

### 3. Hybrid Memory Architecture

**Novel Concept**: Multi-tier token storage with different retention policies

**Architecture Levels:**

1. **Hot Cache**: Recently accessed tokens in fast memory
2. **Warm Storage**: Important tokens in compressed format
3. **Cold Archive**: Summarized/compressed historical context
4. **Retrieval System**: On-demand token restoration from compressed storage

### 4. Predictive Attention Management

**Novel Concept**: Use lightweight ML models to predict future attention patterns

**Components:**

* **Attention Pattern Predictor**: Small neural network predicting which tokens will be attended to
* **Preemptive Loading**: Load likely-to-be-accessed tokens into cache
* **Speculative Eviction**: Remove tokens predicted to have low future attention
* **Confidence-Based Decisions**: Only act on high-confidence predictions

## Implementation Strategy

### Phase 1: Foundation and Analysis (Weeks 1-4)

#### Week 1-2: Environment Setup

1. **Development Environment**
   * Set up SSNeuralcore-infer development kit
   * Install profiling tools (Samsung profiler, ARM Development Studio)
   * Configure cross-compilation for target devices
   * Set up continuous integration pipeline
2. **Baseline Implementation**
   * Implement standard full attention for comparison
   * Create comprehensive benchmarking suite
   * Establish memory and performance baselines
   * Document current bottlenecks and limitations

#### Week 3-4: Basic Sliding Window

1. **Simple SWA Implementation**
   * Fixed-size sliding window attention
   * Basic FIFO token eviction
   * Integration with existing inference pipeline
   * Initial performance measurements
2. **Analysis and Profiling**
   * Memory usage analysis across different window sizes
   * Attention quality metrics (perplexity, downstream tasks)
   * Computational overhead measurement
   * Identification of optimization opportunities

### Phase 2: Core Algorithm Development (Weeks 5-10)

#### Week 5-6: Advanced Sliding Window Patterns

1. **Multiple Window Strategies**
   * Exponential decay windows
   * Overlapping windows with different sizes
   * Layer-specific window configurations
   * Hierarchical attention patterns
2. **Quality Preservation Techniques**
   * Global attention tokens for critical information
   * Summary tokens to compress evicted context
   * Attention residual connections
   * Quality-aware window size adjustment

#### Week 7-8: Intelligent Token Eviction

1. **Importance Scoring Systems**
   * Attention weight-based scoring
   * Semantic importance using lightweight NLP
   * Positional and recency factors
   * Entity and keyword preservation
2. **Adaptive Eviction Policies**
   * Dynamic threshold adjustment
   * Context-aware eviction strategies
   * Task-specific eviction rules
   * Memory pressure response systems

#### Week 9-10: Integration and Optimization

1. **Pipeline Integration**
   * Seamless integration with SSNeuralcore-infer
   * Configuration management system
   * Runtime parameter adjustment
   * Error handling and recovery mechanisms
2. **Performance Optimization**
   * Memory layout optimization
   * SIMD/NEON instruction utilization
   * Cache-friendly data structures
   * Parallel processing implementation

### Phase 3: Hardware-Specific Optimization (Weeks 11-14)

#### Week 11-12: NPU Acceleration

1. **Samsung NPU Integration**
   * Custom kernels for attention computation
   * Memory hierarchy optimization
   * Batch processing optimization
   * Hardware-specific data formats
2. **Memory Management Optimization**
   * Custom memory allocators
   * Memory pool management
   * DMA optimization for large transfers
   * Cache coherency management

#### Week 13-14: Mobile-Specific Features

1. **Power Management**
   * Dynamic voltage and frequency scaling
   * Thermal throttling awareness
   * Battery-aware computation scheduling
   * Sleep/wake optimization
2. **Real-world Integration**
   * Android/iOS integration patterns
   * App lifecycle management
   * Background processing optimization
   * User experience considerations

### Phase 4: Advanced Features and Evaluation (Weeks 15-18)

#### Week 15-16: Novel Algorithm Implementation

1. **Predictive Systems**
   * Attention pattern prediction models
   * Preemptive token loading
   * Speculative eviction systems
   * Confidence-based decision making
2. **Adaptive Systems**
   * Content-aware window sizing
   * Dynamic policy selection
   * Real-time performance adaptation
   * User behavior learning

#### Week 17-18: Comprehensive Evaluation

1. **Performance Benchmarking**
   * Comparison with existing solutions
   * Scalability testing across model sizes
   * Edge case handling verification
   * Long-running stability tests
2. **Quality Assessment**
   * Task-specific performance evaluation
   * Human evaluation studies
   * Attention visualization and analysis
   * Context preservation verification

## Technical Implementation Details

### 1. Sliding Window Attention Architecture

#### Core Data Structures

struct SlidingWindowCache {

// Ring buffer for efficient token management

float\* key\_cache; // [num\_heads][window\_size][head\_dim]

float\* value\_cache; // [num\_heads][window\_size][head\_dim]

int\* token\_ids; // [window\_size]

float\* importance\_scores; // [window\_size]

int head\_ptr; // Current position in ring buffer

int valid\_tokens; // Number of valid tokens in window

};

struct AttentionConfig {

int window\_size; // Base window size

int global\_tokens; // Number of global attention tokens

float eviction\_threshold; // Minimum score for token retention

bool adaptive\_sizing; // Enable dynamic window adjustment

};

#### Attention Computation Flow

1. **Token Input Processing**
   * Compute query, key, value projections
   * Update position embeddings for sliding context
   * Calculate attention mask for current window
2. **Sliding Window Update**
   * Add new token to window (overwrite oldest if full)
   * Update importance scores for all tokens
   * Apply eviction policy if memory pressure exists
3. **Attention Calculation**
   * Compute attention weights within window
   * Apply causal masking for autoregressive generation
   * Aggregate value vectors with attention weights
4. **Output and Cache Update**
   * Generate output token representation
   * Update KV-cache with new token information
   * Prepare for next iteration

### 2. Token Eviction System

#### Importance Scoring Algorithm

def calculate\_token\_importance(token\_idx, attention\_weights, semantic\_features, position):

# Base attention score (how much this token is attended to)

attention\_score = np.mean(attention\_weights[:, token\_idx]) # Average across heads

# Semantic importance (entity, keyword, topic relevance)

semantic\_score = calculate\_semantic\_importance(semantic\_features[token\_idx])

# Positional decay (recent tokens more important)

position\_score = np.exp(-0.1 \* (current\_position - position))

# Special token bonus (punctuation, entities, etc.)

special\_bonus = get\_special\_token\_bonus(token\_idx)

# Combined importance score

importance = (

0.4 \* attention\_score +

0.3 \* semantic\_score +

0.2 \* position\_score +

0.1 \* special\_bonus

)

return importance

#### Eviction Policy Implementation

class TokenEvictionManager {

private:

float memory\_threshold\_;

int min\_tokens\_;

int max\_tokens\_;

public:

std::vector<int> selectTokensForEviction(

const std::vector<float>& importance\_scores,

int target\_eviction\_count

) {

// Create score-index pairs for sorting

std::vector<std::pair<float, int>> scored\_tokens;

for (int i = 0; i < importance\_scores.size(); ++i) {

scored\_tokens.emplace\_back(importance\_scores[i], i);

}

// Sort by importance (lowest first for eviction)

std::sort(scored\_tokens.begin(), scored\_tokens.end());

// Select tokens for eviction (lowest importance)

std::vector<int> eviction\_candidates;

for (int i = 0; i < target\_eviction\_count && i < scored\_tokens.size(); ++i) {

// Additional checks for critical tokens

if (!isCriticalToken(scored\_tokens[i].second)) {

eviction\_candidates.push\_back(scored\_tokens[i].second);

}

}

return eviction\_candidates;

}

};

### 3. Memory Management System

#### Custom Memory Allocator

class AttentionMemoryPool {

private:

void\* memory\_pool\_;

size\_t pool\_size\_;

std::vector<bool> block\_allocated\_;

size\_t block\_size\_;

public:

void\* allocateAttentionBuffer(size\_t size) {

// Find contiguous free blocks

size\_t required\_blocks = (size + block\_size\_ - 1) / block\_size\_;

for (size\_t i = 0; i <= block\_allocated\_.size() - required\_blocks; ++i) {

bool can\_allocate = true;

for (size\_t j = i; j < i + required\_blocks; ++j) {

if (block\_allocated\_[j]) {

can\_allocate = false;

break;

}

}

if (can\_allocate) {

// Mark blocks as allocated

for (size\_t j = i; j < i + required\_blocks; ++j) {

block\_allocated\_[j] = true;

}

return static\_cast<char\*>(memory\_pool\_) + i \* block\_size\_;

}

}

return nullptr; // Out of memory

}

};

## Performance Targets and Metrics

### 1. Memory Efficiency Targets

* **Peak Memory Reduction**: 60-80% reduction compared to full attention
* **KV-Cache Size**: <512MB for 7B parameter models on mobile devices
* **Memory Fragmentation**: <5% wasted space in attention buffers
* **Allocation Efficiency**: <1ms for memory allocation/deallocation operations

### 2. Performance Targets

* **Inference Latency**: <200ms for typical chat responses (100-200 tokens)
* **First Token Latency**: <50ms for response initiation
* **Throughput**: >15 tokens/second sustained generation
* **Attention Computation**: <10ms per attention layer

### 3. Quality Preservation Targets

* **Perplexity Degradation**: <5% increase compared to full attention
* **Downstream Task Performance**: <3% accuracy loss on standard benchmarks
* **Context Coherence**: Maintain conversational quality for 2000+ token contexts
* **Information Retention**: >90% of critical entities and facts preserved

### 4. Energy Efficiency Targets

* **Power Consumption**: <3W during active inference
* **Battery Life Impact**: <20% battery drain per hour of usage
* **Thermal Management**: Stay below 45°C during sustained operation
* **DVFS Integration**: Dynamic performance scaling based on workload

## Evaluation Framework

### 1. Quantitative Metrics

#### Memory Analysis

def measure\_memory\_efficiency(model, test\_sequences):

metrics = {}

for seq\_len in [512, 1024, 2048, 4096]:

# Measure peak memory usage

memory\_tracker = MemoryTracker()

with memory\_tracker:

outputs = model.generate(

input\_ids=test\_sequences[:seq\_len],

max\_length=seq\_len + 100,

sliding\_window\_size=256

)

metrics[f'peak\_memory\_{seq\_len}'] = memory\_tracker.peak\_usage

metrics[f'allocation\_count\_{seq\_len}'] = memory\_tracker.allocation\_count

metrics[f'fragmentation\_{seq\_len}'] = memory\_tracker.fragmentation\_ratio

return metrics

#### Performance Benchmarking

def benchmark\_inference\_speed(model, test\_cases):

results = {}

for test\_name, inputs in test\_cases.items():

latencies = []

for \_ in range(100): # Multiple runs for statistical significance

start\_time = time.perf\_counter()

outputs = model.generate(

input\_ids=inputs['input\_ids'],

max\_length=inputs['max\_length'],

temperature=0.7,

top\_p=0.9

)

end\_time = time.perf\_counter()

latencies.append(end\_time - start\_time)

results[test\_name] = {

'mean\_latency': np.mean(latencies),

'p95\_latency': np.percentile(latencies, 95),

'p99\_latency': np.percentile(latencies, 99),

'tokens\_per\_second': len(outputs[0]) / np.mean(latencies)

}

return results

### 2. Quality Assessment

#### Attention Quality Metrics

def evaluate\_attention\_quality(reference\_model, sliding\_window\_model, test\_data):

quality\_metrics = {}

for sample in test\_data:

# Get attention patterns from both models

ref\_attention = reference\_model.get\_attention\_weights(sample['input'])

sw\_attention = sliding\_window\_model.get\_attention\_weights(sample['input'])

# Calculate attention pattern similarity

attention\_similarity = cosine\_similarity(

ref\_attention.flatten(),

sw\_attention.flatten()

)

# Measure information retention

important\_tokens = identify\_important\_tokens(sample['input'])

retention\_score = calculate\_retention\_score(sw\_attention, important\_tokens)

quality\_metrics[sample['id']] = {

'attention\_similarity': attention\_similarity,

'information\_retention': retention\_score,

'context\_coherence': evaluate\_coherence(sample['output'])

}

return quality\_metrics

### 3. Real-world Testing

#### End-to-End Application Testing

* **Chat Applications**: Multi-turn conversations with context preservation
* **Code Generation**: Technical accuracy with long context requirements
* **Document Summarization**: Information retention across long documents
* **Creative Writing**: Coherence and creativity in extended narratives

#### Stress Testing Scenarios

* **Memory Pressure**: Testing behavior under extreme memory constraints
* **Long Sequences**: Performance with 8K+ token contexts
* **Rapid Switching**: Quick transitions between different conversation topics
* **Edge Cases**: Handling of unusual input patterns and failure modes

## Success Criteria and Milestones

### Primary Success Criteria

1. **Memory Efficiency**: Achieve 70%+ reduction in peak memory usage vs full attention
2. **Performance**: Maintain <10% latency increase while enabling 4x longer contexts
3. **Quality**: <5% degradation in standard NLP benchmarks
4. **Stability**: Zero crashes during 48-hour stress testing

### Milestone Timeline

#### Month 1: Foundation

* ✅ Basic sliding window attention implementation
* ✅ Simple FIFO eviction policy
* ✅ Integration with SSNeuralcore-infer pipeline
* ✅ Baseline performance measurements

#### Month 2: Core Algorithms

* ✅ Advanced eviction policies (importance-based, semantic-aware)
* ✅ Adaptive window sizing
* ✅ Multi-head attention optimization
* ✅ Memory management system

#### Month 3: Optimization

* ✅ Hardware-specific optimizations (NPU, SIMD)
* ✅ Memory layout optimization
* ✅ Parallel processing implementation
* ✅ Power management integration

#### Month 4: Advanced Features

* ✅ Predictive attention management
* ✅ Context compression techniques
* ✅ Real-time adaptation systems
* ✅ Comprehensive evaluation and documentation

### Deliverables

1. **Core Implementation**: Production-ready SWA and token eviction system
2. **Performance Analysis**: Comprehensive benchmarking results
3. **Documentation**: Implementation guide, API documentation, best practices
4. **Research Paper**: Novel techniques and experimental results
5. **Demo Application**: Working example showcasing capabilities

## Future Research Directions

### 1. Learned Attention Patterns

* Train small neural networks to predict optimal attention patterns
* Reinforcement learning for dynamic window size selection
* Meta-learning approaches for rapid adaptation to new domains

### 2. Hierarchical Memory Systems

* Multi-tier storage with different compression levels
* Automatic summarization of evicted context
* Retrieval-augmented attention for historical information

### 3. Cross-Model Optimization

* Shared attention caches across multiple model instances
* Collaborative filtering for token importance scoring
* Federated learning for attention pattern optimization

### 4. Hardware Co-Design

* Custom silicon for sliding window attention
* Near-memory computing for attention operations
* Neuromorphic computing approaches for dynamic attention

This project represents a significant opportunity to advance the state of on-device LLM inference through innovative attention mechanisms and memory management. The combination of sliding window attention and intelligent token eviction addresses fundamental bottlenecks in mobile AI deployment while opening new research directions in efficient transformer architectures.

Understanding 3

# In-Depth Analysis: On-Device LLM Integration with SSNeuralcore-Infer for SWA and Token Eviction

## Project Overview

You're working on enabling an inference pipeline for **Sliding Window Attention (SWA)** and **Token Eviction** in on-device large language model (LLM) deployments using SSNeuralcore-infer. This is a cutting-edge problem at the intersection of efficient AI inference and edge computing.

## Problem Statement Deep Dive

**Core Challenge**: Implementing efficient inference pipelines that can handle:

1. **Sliding Window Attention (SWA)**: A memory-efficient attention mechanism that only considers a window of recent tokens rather than the full context
2. **Token Eviction**: Intelligent removal/discarding of less important tokens to maintain computational efficiency while preserving model performance

**Key Constraints**:

* Must run efficiently on edge devices with limited:
  + Memory (RAM and storage)
  + Compute resources
  + Power budgets
* Must maintain acceptable model accuracy despite the approximations
* Needs to handle variable sequence lengths dynamically

## What Needs to Be Done

### 1. Pipeline Architecture Design

* Design an inference pipeline that integrates SWA and token eviction
* Determine optimal points in the processing flow for each operation
* Handle the interaction between these two techniques

### 2. SWA Implementation

* Implement the sliding window attention mechanism
* Optimize window size selection (fixed vs adaptive)
* Handle edge cases (beginning of sequence, padding, etc.)

### 3. Token Eviction Strategy

* Develop eviction criteria (attention scores, positional importance, etc.)
* Implement eviction mechanisms
* Create preservation strategies for critical tokens

### 4. Memory Management

* Design efficient memory allocation for tokens
* Implement memory recycling for evicted tokens
* Optimize memory access patterns

### 5. Performance Optimization

* Quantization of models and operations
* Operator fusion for efficiency
* Parallelization strategies

### 6. Accuracy Preservation

* Implement compensation mechanisms for evicted tokens
* Develop fallback strategies when confidence is low
* Create evaluation metrics for quality assessment

## Knowledge Domains to Master

### Core Technical Areas:

1. **Transformer Architectures**: Deep understanding of attention mechanisms
2. **Edge AI**: On-device deployment constraints and optimization
3. **Memory Management**: Efficient allocation and access patterns
4. **Approximate Computing**: Tradeoffs between accuracy and efficiency

### Specific Techniques:

* Sliding Window Attention variants
* Token importance scoring methods
* KV cache optimization
* Hardware-aware neural network design

### Tools and Frameworks:

* ONNX Runtime or similar inference engines
* Hardware-specific SDKs (Qualcomm SNPE, ARM Compute Library)
* Profiling tools (Perf, VTune, etc.)
* ML frameworks with edge support (TensorFlow Lite, PyTorch Mobile)

## Existing Solutions and Their Limitations

### Current Approaches:

1. **Full Attention with Pruning**:
   * Maintains full context but prunes less important attention heads
   * Still memory intensive for long sequences
2. **Fixed Context Windows**:
   * Simple but loses important long-range dependencies
   * Can miss critical early context
3. **Memorizing Transformers**:
   * Use external memory for important tokens
   * Adds complexity and memory overhead
4. **StreamingLLM**:
   * Recent approach for infinite-length inputs
   * Maintains attention to initial tokens + recent tokens
   * May keep unnecessary initial tokens

### Limitations to Address:

* Static window sizes that don't adapt to input
* Simple eviction policies that don't consider semantic importance
* Lack of hardware-aware implementations
* Inflexible memory management

## Improvement Opportunities

### Novel Directions to Explore:

1. **Dynamic Window Sizing**:
   * Adjust window size based on content complexity
   * Use lightweight meta-network to predict optimal window size
2. **Semantic-Aware Eviction**:
   * Combine attention scores with positional and linguistic features
   * Implement hierarchical eviction (partial vs complete)
3. **Hardware-Cooperative Design**:
   * Design eviction policies around memory hierarchy
   * Align window sizes with cache line sizes
4. **Compensation Mechanisms**:
   * Maintain compressed representations of evicted tokens
   * Implement attention redistribution for evicted tokens
5. **Hybrid Approaches**:
   * Combine SWA with sparse attention patterns
   * Use different strategies for different layers

## Implementation Strategy

### Phase 1: Foundation

1. **Baseline Implementation**:
   * Implement vanilla transformer inference
   * Add profiling instrumentation
2. **SWA Integration**:
   * Start with fixed window size
   * Add dynamic adjustment capability
3. **Basic Eviction**:
   * Implement simple score-based eviction
   * Add eviction tracking

### Phase 2: Optimization

1. **Memory System Integration**:
   * Align data structures with hardware capabilities
   * Optimize memory access patterns
2. **Adaptive Policies**:
   * Implement content-aware window adjustment
   * Add sophisticated eviction criteria
3. **Accuracy Recovery**:
   * Add compensation mechanisms
   * Implement selective full attention fallback

### Phase 3: Hardware Integration

1. **SSNeuralcore Optimization**:
   * Map operations to neural processor capabilities
   * Utilize specialized instructions
2. **End-to-End Pipeline**:
   * Integrate with preprocessing/postprocessing
   * Optimize data movement

## Evaluation Metrics

### Performance Metrics:

* Memory usage reduction
* Inference latency
* Energy consumption
* Throughput (tokens/second)

### Quality Metrics:

* Task accuracy (vs full attention baseline)
* Coherence scores
* Human evaluation (for subjective tasks)

## Potential Challenges and Mitigations

1. **Accuracy Drop**:
   * Mitigation: Implement accuracy recovery mechanisms
   * Fallback to full attention when confidence is low
2. **Hardware Limitations**:
   * Mitigation: Profile early and often
   * Design with hardware constraints in mind from start
3. **Dynamic Inputs**:
   * Mitigation: Robust handling of variable length sequences
   * Stress test with extreme cases

## Resources and References

### Key Papers:

1. "Efficient Streaming Language Models with Attention Sinks" (StreamingLLM)
2. "Longformer: The Long-Document Transformer" (Sliding Window Attention)
3. "H2O: Heavy-Hitter Oracle for Efficient Generative Inference"
4. "Scissorhands: Exploiting the Persistence of Importance in LLMs"

### Open Source Projects:

* HuggingFace Transformers (reference implementations)
* vLLM (optimized inference server)
* TensorRT-LLM (hardware-optimized inference)

### Tools:

* ONNX Runtime for cross-platform deployment
* Qualcomm SNPE for DSP/NPU acceleration
* ARM Compute Library for CPU optimization

## Proposed Timeline

1. **Week 1-2**: Literature review and baseline implementation
2. **Week 3-4**: SWA integration and basic eviction
3. **Week 5-6**: Dynamic policies and accuracy recovery
4. **Week 7-8**: Hardware optimization and profiling
5. **Week 9-10**: End-to-end integration and evaluation

Would you like me to elaborate on any specific aspect of this project plan? I can provide more details about particular implementation strategies, mathematical formulations of the attention mechanisms, or specific optimization techniques for the SSNeuralcore architecture.

1. System Architecture

Core Components:

Copy

Download

┌───────────────────────────────────────────────────────┐

│ Inference Pipeline │

├───────────────────┬───────────────────┬───────────────┤

│ Token Manager │ Attention Engine │ Memory Ops │

└─────────┬─────────┴─────────┬─────────┴───────┬───────┘

│ │ │

┌─────────▼─────────┐ ┌───────▼───────┐ ┌───────▼───────┐

│ Sliding Window │ │ Token Evict │ │ KV Cache │

│ Attention (SWA) │ │ Controller │ │ Manager │

└───────────────────┘ └───────────────┘ └───────────────┘

Data Flow:

1. Input tokens enter Token Manager
2. Token Evict Controller evaluates token importance
3. SWA processes tokens within current window
4. KV Cache Manager handles memory operations
5. Output generated while maintaining memory constraints

2. Sliding Window Attention Implementation

Core Algorithm:

python

Copy

Download

def sliding\_window\_attention(Q, K, V, window\_size, attention\_mask=None):

"""

Q: Query matrix [batch\_size, num\_heads, seq\_len, dim]

K: Key matrix [batch\_size, num\_heads, seq\_len, dim]

V: Value matrix [batch\_size, num\_heads, seq\_len, dim]

window\_size: size of attention window

"""

seq\_len = Q.size(2)

scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(Q.size(-1))

*# Apply sliding window mask*

if window\_size < seq\_len:

mask = torch.ones\_like(scores) \* float('-inf')

for i in range(seq\_len):

start = max(0, i - window\_size // 2)

end = min(seq\_len, i + window\_size // 2 + 1)

mask[:, :, i, start:end] = 0

scores = scores + mask

if attention\_mask is not None:

scores = scores + attention\_mask

attn\_weights = torch.softmax(scores, dim=-1)

output = torch.matmul(attn\_weights, V)

return output, attn\_weights

Optimizations:

1. **Causal Mask Integration**:

python

Copy

Download

*# Combine causal mask with sliding window*

if is\_causal:

causal\_mask = torch.triu(torch.ones(seq\_len, seq\_len) \* float('-inf'), diagonal=1)

mask = torch.maximum(mask, causal\_mask)

1. **Window Size Adaptation**:

python

Copy

Download

def dynamic\_window\_size(sequence):

*# Implement content-based window sizing*

complexity = calculate\_complexity(sequence) *# Could be based on entropy, novelty, etc.*

base\_size = 64 *# Minimum window size*

return min(max\_window\_size, base\_size + int(complexity \* scaling\_factor))

3. Token Eviction System

Eviction Criteria:

python

Copy

Download

class TokenEvictor:

def \_\_init\_\_(self, evict\_strategy='attention'):

self.strategy = evict\_strategy

self.attention\_history = {}

def calculate\_importance(self, token\_idx, attn\_weights):

if self.strategy == 'attention':

*# Average attention received*

return torch.mean(attn\_weights[:, :, :, token\_idx])

elif self.strategy == 'positional':

*# Favor middle positions (empirically found important)*

seq\_len = attn\_weights.size(-1)

position\_score = 1 - abs((token\_idx/seq\_len) - 0.5)\*2

return position\_score

elif self.strategy == 'hybrid':

attention\_score = torch.mean(attn\_weights[:, :, :, token\_idx])

position\_score = 1 - abs((token\_idx/seq\_len) - 0.5)\*2

return 0.7\*attention\_score + 0.3\*position\_score

def update\_history(self, token\_idx, importance):

if token\_idx not in self.attention\_history:

self.attention\_history[token\_idx] = []

self.attention\_history[token\_idx].append(importance)

def get\_eviction\_candidates(self, current\_memory, target\_memory):

importance\_scores = {}

for idx in current\_memory:

*# Use exponentially weighted moving average*

history = self.attention\_history.get(idx, [0])

ewma = sum(0.9\*\*i \* score for i, score in enumerate(reversed(history)))

importance\_scores[idx] = ewma / sum(0.9\*\*i for i in range(len(history)))

*# Sort by importance (ascending)*

sorted\_tokens = sorted(importance\_scores.items(), key=lambda x: x[1])

return [token[0] for token in sorted\_tokens[:len(current\_memory)-target\_memory]]

Eviction Strategies:

1. **Hard Eviction**:

python

Copy

Download

def hard\_evict(kv\_cache, evict\_indices):

*# Simply remove the evicted tokens*

new\_k = torch.cat([kv\_cache['k'][:, :, :, :, i]

for i in range(kv\_cache['k'].size(-1))

if i not in evict\_indices], dim=-1)

new\_v = torch.cat([kv\_cache['v'][:, :, :, :, i]

for i in range(kv\_cache['v'].size(-1))

if i not in evict\_indices], dim=-1)

return {'k': new\_k, 'v': new\_v}

1. **Soft Eviction (Compression)**:

python

Copy

Download

def soft\_evict(kv\_cache, evict\_indices):

*# Cluster similar tokens and merge*

keep\_indices = [i for i in range(kv\_cache['k'].size(-1)) if i not in evict\_indices]

k\_keep = kv\_cache['k'][:, :, :, :, keep\_indices]

v\_keep = kv\_cache['v'][:, :, :, :, keep\_indices]

*# For evicted tokens, find nearest neighbor and merge*

for idx in evict\_indices:

k\_token = kv\_cache['k'][:, :, :, :, idx]

v\_token = kv\_cache['v'][:, :, :, :, idx]

*# Find most similar token (cosine similarity)*

similarities = F.cosine\_similarity(k\_token, k\_keep, dim=-1)

closest = torch.argmax(similarities)

*# Merge with closest token*

k\_keep[:, :, :, :, closest] = (k\_keep[:, :, :, :, closest] + k\_token) / 2

v\_keep[:, :, :, :, closest] = (v\_keep[:, :, :, :, closest] + v\_token) / 2

return {'k': k\_keep, 'v': v\_keep}

4. Memory Management System

KV Cache Implementation:

python

Copy

Download

class KVCacheManager:

def \_\_init\_\_(self, max\_size, evict\_policy='hybrid'):

self.cache = {'k': None, 'v': None}

self.max\_size = max\_size

self.evictor = TokenEvictor(evict\_policy)

self.current\_size = 0

def update(self, new\_k, new\_v, attention\_weights):

if self.cache['k'] is None:

self.cache = {'k': new\_k, 'v': new\_v}

self.current\_size = new\_k.size(-1)

return self.cache

*# Concatenate new tokens*

self.cache['k'] = torch.cat([self.cache['k'], new\_k], dim=-1)

self.cache['v'] = torch.cat([self.cache['v'], new\_v], dim=-1)

self.current\_size += new\_k.size(-1)

*# Update importance scores*

for i in range(self.current\_size):

importance = self.evictor.calculate\_importance(i, attention\_weights)

self.evictor.update\_history(i, importance)

*# Evict if needed*

if self.current\_size > self.max\_size:

evict\_indices = self.evictor.get\_eviction\_candidates(

range(self.current\_size),

self.max\_size)

self.cache = hard\_evict(self.cache, evict\_indices)

self.current\_size = self.cache['k'].size(-1)

return self.cache

Memory Optimization Techniques:

1. **Quantized KV Cache**:

python

Copy

Download

def quantize\_kv\_cache(cache, bits=8):

*# Dynamic quantization for KV cache*

def quantize\_tensor(tensor):

scale = tensor.abs().max() / (2\*\*(bits-1)-1)

quantized = torch.clamp(torch.round(tensor/scale), -2\*\*(bits-1), 2\*\*(bits-1)-1)

return quantized.to(torch.int8), scale

k\_quant, k\_scale = quantize\_tensor(cache['k'])

v\_quant, v\_scale = quantize\_tensor(cache['v'])

return {

'k': (k\_quant, k\_scale),

'v': (v\_quant, v\_scale)

}, lambda: (

cache['k'].dtype,

cache['v'].dtype

)

1. **Paged Attention**:

python

Copy

Download

class PagedKVCache:

def \_\_init\_\_(self, page\_size=256, max\_pages=16):

self.page\_size = page\_size

self.max\_pages = max\_pages

self.pages = []

self.page\_table = {} *# token\_idx -> (page\_num, offset)*

def add\_tokens(self, new\_k, new\_v):

num\_new = new\_k.size(-1)

current\_pos = sum(len(p['k']) for p in self.pages)

*# Add to existing page if space*

if self.pages and len(self.pages[-1]['k']) + num\_new <= self.page\_size:

page = self.pages[-1]

page['k'] = torch.cat([page['k'], new\_k], dim=-1)

page['v'] = torch.cat([page['v'], new\_v], dim=-1)

else:

*# Create new page*

if len(self.pages) >= self.max\_pages:

self.evict\_page()

self.pages.append({'k': new\_k, 'v': new\_v})

*# Update page table*

for i in range(num\_new):

self.page\_table[current\_pos + i] = (len(self.pages)-1,

len(self.pages[-1]['k'])-num\_new+i)

def evict\_page(self):

*# Implement page eviction policy (LRU, LFU, etc.)*

evict\_page = determine\_page\_to\_evict() *# Implementation needed*

del self.pages[evict\_page]

*# Rebuild page table*

self.\_rebuild\_page\_table()

def \_rebuild\_page\_table(self):

new\_table = {}

global\_pos = 0

for page\_num, page in enumerate(self.pages):

for offset in range(len(page['k'])):

new\_table[global\_pos] = (page\_num, offset)

global\_pos += 1

self.page\_table = new\_table

5. Complete Inference Pipeline

End-to-End Processing:

python

Copy

Download

class EfficientInferencePipeline:

def \_\_init\_\_(self, model, window\_size=64, max\_memory=512):

self.model = model

self.window\_size = window\_size

self.kv\_cache = KVCacheManager(max\_memory)

self.evictor = TokenEvictor()

def generate(self, input\_ids, max\_length=100):

outputs = []

past\_key\_values = None

for step in range(max\_length):

*# Forward pass with current window*

model\_outputs = self.model(

input\_ids,

attention\_mask=self.\_create\_attention\_mask(),

past\_key\_values=past\_key\_values

)

*# Get attention weights for eviction decisions*

attn\_weights = model\_outputs.attentions[-1] *# Last layer attention*

*# Update KV cache*

past\_key\_values = self.kv\_cache.update(

model\_outputs.past\_key\_values['k'],

model\_outputs.past\_key\_values['v'],

attn\_weights

)

*# Get next token*

next\_token = self.\_sample\_token(model\_outputs.logits)

outputs.append(next\_token)

input\_ids = torch.tensor([[next\_token]])

return outputs

def \_create\_attention\_mask(self):

*# Create combined mask for SWA and causal attention*

seq\_len = self.kv\_cache.current\_size + 1 *# +1 for current token*

mask = torch.zeros(seq\_len, seq\_len)

*# Sliding window*

if self.window\_size < seq\_len:

for i in range(seq\_len):

start = max(0, i - self.window\_size // 2)

end = min(seq\_len, i + self.window\_size // 2 + 1)

mask[i, start:end] = 1

*# Causal mask*

mask = torch.tril(mask)

*# Convert to attention mask format (0=attend, -inf=ignore)*

return torch.where(mask > 0, torch.zeros\_like(mask), float('-inf'))

def \_sample\_token(self, logits):

*# Implement sampling strategy (greedy, top-k, etc.)*

return torch.argmax(logits[:, -1]).item()

6. Optimization Techniques

Hardware-Specific Optimizations:

1. **SSNeuralcore Intrinsics**:

cpp

Copy

Download

*// Example of optimized attention computation using hardware intrinsics*

void optimized\_attention(float\* Q, float\* K, float\* V, float\* output, int dim, int seq\_len) {

for (int i = 0; i < seq\_len; i++) {

*// Use neural core's matrix acceleration*

neuralcore\_mm(Q + i\*dim, K, output + i\*seq\_len, dim, seq\_len);

*// Apply sliding window mask in hardware*

neuralcore\_window\_mask(output + i\*seq\_len, i, seq\_len, window\_size);

*// Hardware-accelerated softmax*

neuralcore\_softmax(output + i\*seq\_len, seq\_len);

*// Final matrix multiplication*

neuralcore\_mm(output + i\*seq\_len, V, output + i\*dim, seq\_len, dim);

}

}

1. **Memory Access Patterns**:

python

Copy

Download

def optimized\_cache\_access(cache, indices):

*# Reorder indices for sequential memory access*

sorted\_indices = sorted(indices)

*# Prefetch next cache lines*

for i in range(0, len(sorted\_indices), cache\_line\_size):

prefetch(cache['k'][..., sorted\_indices[i:i+cache\_line\_size]])

prefetch(cache['v'][..., sorted\_indices[i:i+cache\_line\_size]])

*# Process in batches*

processed = []

for i in sorted\_indices:

processed.append(process\_token(cache['k'][..., i], cache['v'][..., i]))

return processed

7. Evaluation Metrics Implementation

Quality and Performance Tracking:

python

Copy

Download

class InferenceMetrics:

def \_\_init\_\_(self):

self.metrics = {

'memory\_usage': [],

'latency': [],

'accuracy': [],

'eviction\_rate': []

}

def log\_memory(self, current, max\_allowed):

utilization = current / max\_allowed

self.metrics['memory\_usage'].append(utilization)

def log\_latency(self, start\_time):

self.metrics['latency'].append(time.time() - start\_time)

def log\_accuracy(self, reference, generated):

score = calculate\_similarity(reference, generated) *# Implement suitable metric*

self.metrics['accuracy'].append(score)

def log\_eviction(self, evicted, total):

self.metrics['eviction\_rate'].append(evicted / total)

def get\_summary(self):

return {

'avg\_memory': np.mean(self.metrics['memory\_usage']),

'avg\_latency': np.mean(self.metrics['latency']),

'avg\_accuracy': np.mean(self.metrics['accuracy']),

'total\_evictions': sum(self.metrics['eviction\_rate'])

}

8. Integration with Existing Frameworks

ONNX Runtime Integration:

python

Copy

Download

def create\_onnx\_session(model\_path, providers=['SNPEExecutionProvider']):

*# Create session with hardware acceleration*

sess\_options = onnxruntime.SessionOptions()

sess\_options.graph\_optimization\_level = onnxruntime.GraphOptimizationLevel.ORT\_ENABLE\_ALL

*# Register custom ops for SWA and eviction*

register\_custom\_op('SlidingWindowAttention', sliding\_window\_attention\_impl)

register\_custom\_op('TokenEviction', token\_eviction\_impl)

return onnxruntime.InferenceSession(model\_path, sess\_options, providers=providers)

def optimized\_inference(session, inputs):

*# Prepare inputs with KV cache*

ort\_inputs = {

'input\_ids': inputs['input\_ids'],

'attention\_mask': create\_swa\_mask(inputs['seq\_len']),

'past\_key\_values': inputs['kv\_cache']

}

*# Run inference*

ort\_outputs = session.run(None, ort\_inputs)

*# Process outputs with eviction*

updated\_kv\_cache = apply\_eviction(ort\_outputs['present\_key\_values'])

return ort\_outputs['logits'], updated\_kv\_cache

Next Steps for Implementation

1. **Benchmarking**:
   * Profile each component to identify bottlenecks
   * Compare against baseline full-attention implementation
2. **Adaptive Tuning**:
   * Implement automatic window size adjustment
   * Develop dynamic eviction thresholds
3. **Hardware Optimization**:
   * Map critical operations to neural processor instructions
   * Optimize memory access patterns for the specific architecture
4. **Error Recovery**:
   * Implement mechanisms to detect and recover from accuracy drops
   * Add fallback strategies for critical sections