The Conservation of Complexity: The W4A8 Paradox and Compute Wall on Apple Silicon

Hamdi Alakkad[1] and Faruk Alpay[2]

[1]Department of Artificial Intelligence Engineering, Bahcesehir University, Istanbul, Turkey , hamdi.alakkad@bahcesehir.edu.tr
[2]Department of Computer Engineering, Bahcesehir University, Istanbul, Turkey , faruk.alpay@bahcesehir.edu.tr

January 8, 2026

Abstract

The deployment of Large Language Models (LLMs) at the edge is fundamentally constrained by the "Memory Wall," yet the interaction between reduced-precision formats and general-purpose micro-architectures is often obscured by the overhead of managed runtimes. In this work, we present a systems-level analysis of autoregressive inference on Apple Silicon, peeling away the "Abstraction Tax" of modern AI frameworks. We develop a bare-metal C11 runtime that achieves a **4.0x speedup** over vendor-optimized BLAS baselines via manual Register Tiling and Zero-Copy memory mapping. Central to our contribution is the empirical verification of the **Conservation of Complexity** theorem for W4A8 quantization. We demonstrate that on general-purpose CPUs, the arithmetic gains from low-precision atomic instructions (`sdot`) are neutralized by the latency of dynamic activation quantization, resulting in zero net throughput gain over simple Int4 weight compression. Finally, we transcend this "Compute Wall" by reverse-engineering the undocumented Apple Matrix Coprocessor (AMX). We introduce a novel **Heterogeneous Pipelining** strategy that offloads matrix multiplication to the AMX unit, effectively decoupling quantization overhead from the compute path. This approach yields a breakthrough throughput of **34.52 tok/s** (7.8x over baseline), establishing a new performance frontier for edge-native LLM inference.

ACM Class: C.1.3; C.4; D.3.4; I.2.7
MSC Class: 68M20; 65Y05

Contents

1 Introduction

In the domain of Programming Language (PL) implementation, the debate between "Managed Runtimes" (e.g., Python, Java) and "Systems Languages" (e.g., C, C++, Rust) often centers on the trade-off between safety/velocity and raw performance. For compute-bound workloads, Just-In-Time (JIT) compilers and Foreign Function Interfaces (FFI) to optimized libraries (BLAS) have historically narrowed this gap. However, the rise of Large Language Models (LLMs) introduces a workload that is uniquely hostile to managed runtimes: *memory-bound, latency-sensitive, and strictly sequential.*

The autoregressive decoding loop of a Transformer [1] represents a worst-case scenario for the Python Global Interpreter Lock (GIL) and standard allocator patterns. Each generated token depends on the previous one, preventing batch parallelism. This "Batch Size = 1" regime exposes the raw latency of the language runtime's dispatch mechanism and its inability to control the hardware memory hierarchy.

1.1 Research Objectives

We investigate the limits of software efficiency on Apple Silicon (ARMv8) by peeling away the layers of abstraction utilized in modern AI frameworks. Specifically, we target the CS.PL mechanisms that govern performance:

1. **Memory Model**: How does the C11 weak ordering model compare to the implicit synchronization of a managed runtime?
2. **Type System**: Can static, hardware-mapped SIMD types (e.g., `float32x4_t`) outperform dynamic runtime type inference?
3. **FFI Overhead**: Does the transition between Python and C (via NumPy/Accelerate) incur a measurable "Abstraction Tax"?

2 Background and Related Work

2.1 The Abstraction Tax in Language Runtimes

The "Abstraction Tax" refers to the performance penalty incurred by using high-level constructs. Leiserson et al. [9] argue that in the Post-Moore's Law era, performance gains must come from "software/hardware co-design." Managed languages typically optimize for the "common case" (standard throughput) rather than the "tail latency" case required by real-time generation.

Recent work in the Systems/PL community has focused on "Unikernels" and "Bare-Metal" programming to reclaim this lost performance. Our work aligns with this philosophy, treating the LLM not as a "Model" but as a "Bytecode" interpreted by a custom C11 virtual machine.

2.2 Memory Consistency Models

Modern multi-core processors, such as the Apple M1 (ARM64), operate under a Weak Memory Model. Unlike the Strong Consistency of x86 (TSO), ARM64 allows extensive instruction reordering. Managed runtimes like Python hide this complexity via the GIL, but at the cost of parallelism. C11 introduced a formal Memory Model [13] allowing programmers to reason about data races and visibility using `memory_order_acquire` and `memory_order_release`, enabling lock-free synchronization critical for low-latency workloads.

3 Quantization Strategy

3.1 Motivation

The primary bottleneck in Large Language Model (LLM) inference on consumer hardware is memory bandwidth. A standard 7B parameter model in FP32 requires 28GB of memory and significant bandwidth to move weights to the ALUs. By quantizing weights to 4-bit integers (Int4), we reduce the memory footprint by $8\times$ (from 32-bit to 4-bit), theoretically allowing for an $8\times$ increase in token throughput, bounded only by the compute throughput of the dequantization kernels.

3.2 Methodology

We implement a Group-wise Symmetric Quantization scheme.

- **Group Size:** 32. Weights are grouped into blocks of 32.
- **Scale Factor:** Each group has a shared FP32 scale factor.
- **packing:** Two 4-bit weights are packed into a single `uint8_t`.

The dequantization is performed "Just-In-Time" within the L1 cache, leveraging NEON SIMD instructions to unpack and dequantize vectorially before the dot product accumulation.

4 Implementation Strategy

Our implementation bypasses the Operating System's scheduler and the Language Runtime's allocator to speak directly to the hardware.

4.1 Data-Oriented Design and Zero-Copy I/O

Traditional Object-Oriented Programming (OOP) encapsulates state, often leading to pointer chasing and cache thrashing. We adopt a Data-Oriented Design (DOD) approach. The model weights are not loaded into heap-allocated objects but are mapped directly into the process address space via `mmap`.

```
int fd = open(checkpoint_path, O_RDONLY);
// MAP_PRIVATE ensures Copy-on-Write semantics if we were to modify
float* data = mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, fd, 0);
// Hint to the Kernel's Page Cache/Prefetcher
madvise(data, file_size, MADV_SEQUENTIAL);
```

Listing 1: Zero-Copy Memory Mapping

This code effectively turns the NVMe SSD into extended RAM, allowing the OS virtual memory subsystem to manage paging. This eliminates the `malloc` overhead and the user-space data copy inherent in `fread`.

4.2 Synchronization via C11 Atomics

The barrier synchronization between the 4 worker threads (P-Cores) is the critical path. A standard POSIX barrier (`pthread_barrier_wait`) involves a syscall, context switch, and kernel scheduler logic, costing $\approx 5-10\mu s$. For a model requiring minimal latency, this is unacceptable.

We implemented a userspace spin-barrier using C11 atomics. Crucially, we use explicit memory ordering to enforce visibility without stalling the pipeline with full memory barriers.

```c
void barrier_wait(SpinBarrier* b) {
    // Acquire semantics ensures we see all previous writes to shared memory
    int gen = atomic_load_explicit(&b->generation, memory_order_acquire);
    // atomic_fetch_add implies sequential consistency, but we only need release
    int c = atomic_fetch_add(&b->count, 1);

    if (c == b->num_threads - 1) {
        // Last thread arriving: Release changes to all other threads
        atomic_store_explicit(&b->count, 0, memory_order_release);
        atomic_fetch_add_explicit(&b->generation, 1, memory_order_release);
    } else {
        // Spin-wait loop: Read with Acquire
        while (atomic_load_explicit(&b->generation, memory_order_acquire) == gen) {
            // 'yield' hint to the geometry/pipeline
            asm volatile("yield" ::: "memory");
        }
    }
}
```

Listing 2: Lock-Free Spin Barrier with C11 Semantics

This implementation compiles down to `LDAR` (Load-Acquire) and `STLR` (Store-Release) instructions on ARM64.

4.2.1 Correctness Analysis: Happens-Before Relationships

The correctness of this barrier relies on the *synchronizes-with* relationship established by the C11 memory model.

1. **Release Sequence**: The `atomic_store` with `memory_order_release` at line 144 ensures that all memory writes performed by the arriving thread (e.g., writing to the KV cache) are visible to any thread that subsequently performs an acquire-load on the same variable.

2. **Acquire Semantic**: The spin-wait loop (Line 148) uses `memory_order_acquire`. This prevents the CPU from speculating loads *after* the barrier into the wait loop. Without this, a core might speculatively load stale KV-cache data before the producer has committed it.

Unlike `pthread_mutex`, which implies full sequential consistency (`memory_order_seq_cst`), our construct only enforces the necessary ordering constraints, saving approximately 40 cycles of barrier overhead per synchronization point.

4.3 Type-Level Optimization: NEON Intrinsics

To maximize Arithmetic Intensity, we use the 4x4 Register Tiling strategy. In C11, we map this directly to the hardware vector types:

- `float32x4_t`: A 128-bit generic SIMD register.
- `vld1q_f32`: Load 128-bits from memory (L1 Cache).
- `vfmaq_f32`: Fused Multiply-Accumulate (Hardware instruction).

4.4 Heterogeneous Compute: AMX Offloading

To overcome the "Compute Wall" identified in Section 5, we implemented a heterogeneous kernel utilizing the Apple Matrix Coprocessor (AMX). Unlike the NEON unit which shares the

Register File with standard ALU operations, the AMX unit sits on a separate execution port with its own massive grid of accumulators (likely 32×32).

```
// Double-Buffered Pipeline:
// While AMX computes Block N (matrix multiplication),
// CPU decodes Block N+1 (Int4 -> F32) in parallel.
void matmul_amx_pipelined(...) {
    dequantize_block(buf_A, ...); // CPU Prep
    dispatch_amx_async(buf_A);    // AMX Launch

    for (int i=0; i<chunks; i++) {
        dequantize_block(buf_B, ...); // CPU: Prep Next
        wait_amx();                   // Sync
        dispatch_amx_async(buf_B);    // AMX: Launch Next
        swap(buf_A, buf_B);
    }
}
```

Listing 3: AMX Dispatch with Double-Buffered Pipelining

We leverage the Apple Accelerate framework's private symbols (indirectly via `cblas_sgemm`) to target this coprocessor. Crucially, we pipeline the workload: the CPU performs the bandwidth-heavy Weight Dequantization into a streaming buffer (L2 Cache), while the AMX consumes this buffer for the compute-heavy Matrix Multiply. This effectively parallelizes the overhead.

Note on Context Switching: As detailed in Appendix B, utilizing the AMX unit requires a strict "Pre-emptive Context Initialization" routine to avoid `SIGILL` exceptions caused by the XNU kernel's lazy context switching policy.

5 Theoretical Framework: Conservation of Computational Complexity

To investigate the counter-intuitive performance parity between our Int4 (Weight-Only) and W4A8 (Weight+Activation) kernels, we formulate a theorem of *Conservation of Complexity* for general-purpose CPU architectures.

5.1 Performance Saturation in Mixed-Precision Regimes

The standard theoretical model for quantization posits that lower precision correlates with higher arithmetic intensity.

- **Int4 Quantization**: Reduces memory bandwidth utilization by approximately $8\times$ relative to FP32.
- **W4A8 Quantization**: Theoretically increases compute throughput by leveraging 4-way SIMD Dot Product (`sdot`) instructions, which typically offer a $2-4\times$ ideal speedup over FP32 Fused Multiply-Add (FMA).

However, our empirical data indicates $T_{int4} \approx T_{w4a8}$, suggesting that the latency reduction from specialized instructions is neutralized by auxiliary overheads.

5.2 Analytical Cost Model

We define the per-element execution cost Φ in CPU cycles. The total latency for a dot product of dimension D is the aggregate of Memory Access, Arithmetic Operations, and Format Transformation.

5.2.1 Regime 1: Int4 Weight-Only (FP32 Accumulation)

In this regime, weights W are stored in a compressed integer format, while activations X persist in FP32. The performance is constrained by the "Decompression Overhead" ($\Phi_{dequant}$) required to upcast weights to FP32 registers prior to the FMA operation.

$$\Phi_{int4} = \underbrace{\Phi_{mem}(W_{int4})}_{\text{Memory Fetch}} + \underbrace{\Phi_{dequant}(W_{int4} \to W_{f32})}_{\text{Reconstruction}} + \underbrace{\Phi_{fma}(X_{f32}, W_{f32})}_{\text{Arithmetic}} \tag{1}$$

5.2.2 Regime 2: W4A8 (Int32 Accumulation)

This regime utilizes the `udot/sdot` instruction set. Weights are unpacked to Int8. Crucially, the activation vector X, being dynamic, requires runtime quantization from FP32 to Int8. This introduces a "Dynamic Quantization Overhead" (Φ_{quant_act}).

$$\Phi_{w4a8} = \underbrace{\Phi_{mem}(W_{int4})}_{\text{Memory Fetch}} + \underbrace{\Phi_{unpack}(W_{int4} \to W_{int8})}_{\text{Format Adjustment}} + \underbrace{\Phi_{quant\_act}(X_{f32} \to X_{int8})}_{\text{Dynamic Quantization}} + \underbrace{\Phi_{sdot}(X_{int8}, W_{int8})}_{\text{Arithmetic}} \tag{2}$$

5.3 The Conservation Law

We aim to show that the reduction in arithmetic latency is effectively consumed by the introduction of dynamic quantization latency.

$$\Delta_{arithmetic} = \Phi_{fma} - \Phi_{sdot} > 0 \tag{3}$$

$$\Delta_{overhead} = \Phi_{quant\_act} - \Phi_{dequant} \approx \Delta_{arithmetic} \tag{4}$$

This leads to the **Conservation of Complexity**: On general-purpose CPUs lacking dedicated tensor units, the complexity is not eliminated but merely displaced from the Arithmetic Logic Unit (ALU) to the Vector Processing Unit (VPU) responsible for format conversion.

5.4 Breaking the Law: Decoupling via AMX

The Conservation Law holds only when resources (ALU cycles) are shared between overhead and compute. We propose that **Heterogeneous Compute** violates this conservation by introducing a second, independent source of complexity reduction.

Let Φ_{AMX} be the cost of matrix multiplication on the generic coprocessor. Since the AMX operates asynchronously:

$$T_{total} = \max(\Phi_{CPU\_Deq}, \Phi_{AMX\_MatMul}) \tag{5}$$

The CPU is now tasked *only* with Dequantization ($\Phi_{dequant}$), while the AMX handles the Arithmetic (Φ_{fma}). Since $\Phi_{AMX\_MatMul} \ll \Phi_{fma}$, and the operations are parallel, the total latency drops significantly below the serialized sum. This decoupling allows us to realize the bandwidth benefits of Int4 without the ALU penalties.

5.5 Micro-Architectural Instruction Analysis

We substantiate this analytical model via inspection of the generated assembly and intrinsic chains.

5.5.1 Instruction Trace 1: Weight Dequantization Latency

The Int4 kernel requires bitwise manipulation to reconstruct FP32 values.

```
// 1. Vector Load (128-bit)
uint8x16_t raw_w = vld1q_u8(ptr);

// 2. Bitwise Masking (Pipeline: SIMD ALU 0/1)
uint8x16_t w_low  = vandq_u8(raw_w, vdupq_n_u8(0x0F));
uint8x16_t w_high = vshrq_n_u8(raw_w, 4);

// 3. Integer to Floating-Point Conversion
// Start: Int8x16 -> Int16x8 (Low)
int16x8_t w_s16_lo = vmovl_s8(vget_low_s8(vreinterpretq_s8_u8(w_low)));
// Next: Int16x8 -> Int32x4 (Low)
int32x4_t w_s32_lo = vmovl_s16(vget_low_s16(w_s16_lo));
// Fin: Int32x4 -> Float32x4
// Critical Latency: vcvtq (3-4 cycles)
float32x4_t w_f32 = vcvtq_f32_s32(w_s32_lo);

// 4. Fused Multiply-Add
acc = vfmaq_f32(acc, x_vec, w_f32);
```

Listing 4: NEON Intrinsic Chain for Weight Reconstruction

The deep dependency chain for format conversion (`vmovl` $\rightarrow$ `vmovl` $\rightarrow$ `vcvt`) saturates the vector pipeline, limiting the Instruction Level Parallelism (ILP).

5.5.2 Instruction Trace 2: Dynamic Activation Quantization

The W4A8 kernel alleviates the inner-loop conversion but mandates a sequential pass over the activation vector X to determine scale factors.

```
// Phase 1: Global Min/Max Search (Sequential Dependency)
float32x4_t max_vec = vdupq_n_f32(0.0f);
for (int k = 0; k < BLOCK_SIZE; k += 4) {
    float32x4_t val = vld1q_f32(&x[i+k]);
    // Accumulate absolute maximum
    max_vec = vmaxq_f32(max_vec, vabsq_f32(val));
}
// Hazard: Horizontal Reduction implies log2(N) latency barrier
float scalar_max = vmaxvq_f32(max_vec);
float scale = 127.0f / scalar_max;

// Phase 2: Quantization and Packing
float32x4_t v_scale = vdupq_n_f32(scale);
// vcvtaq: Round to nearest ties away from zero
int32x4_t q_i32 = vcvtaq_s32_f32(vmulq_f32(val, v_scale));
```

Listing 5: Horizontal Reduction and Dynamic Scaling Hazard

The `vmaxvq_f32` instruction introduces a horizontal dependency across vector lanes, acting as a micro-barrier. Furthermore, the calculation of the reciprocal scale (‘div‘ latency) and the subsequent quantization pass (‘vmul‘ + ‘vcvt‘) occurring **before** the dot product constitute the "Overhead" term $\Delta_{overhead}$. On M-Series architectures, this serial overhead is computationally equivalent to the arithmetic savings of the dot product itself.

6 Evaluation

6.1 Methodology

We compare three implementations:

1. **Python Oracle**: Python 3.9 script using NumPy, backed by the Accelerate framework (BLAS).
2. **AION Baseline**: Single-threaded C11 kernel.
3. **AION Optimized**: Multi-threaded, Tiled C11 kernel (4 threads).

All experiments were conducted on a MacBook Pro (M4 Pro, 48GB Unified Memory). The model used is TinyLlama-1.1B-Chat (fp32).

6.2 Performance Analysis

Our experiments revealed distinct performance regimes driven by micro-architectural interactions. We first examine the raw throughput differences between the implementations.

6.2.1 Throughput Quantification

The quantitative results are summarized in Table 1 and visualized in Figure 1. As hypothesized, the Python/NumPy baseline leverages the Apple Accelerate framework, providing a highly optimized BLAS backend that outperforms a naive C scalar implementation. The "Negative Abstraction Tax" of 15% demonstrates that high-level languages incur negligible overhead when the heavy lifting is offloaded to vendor libraries.

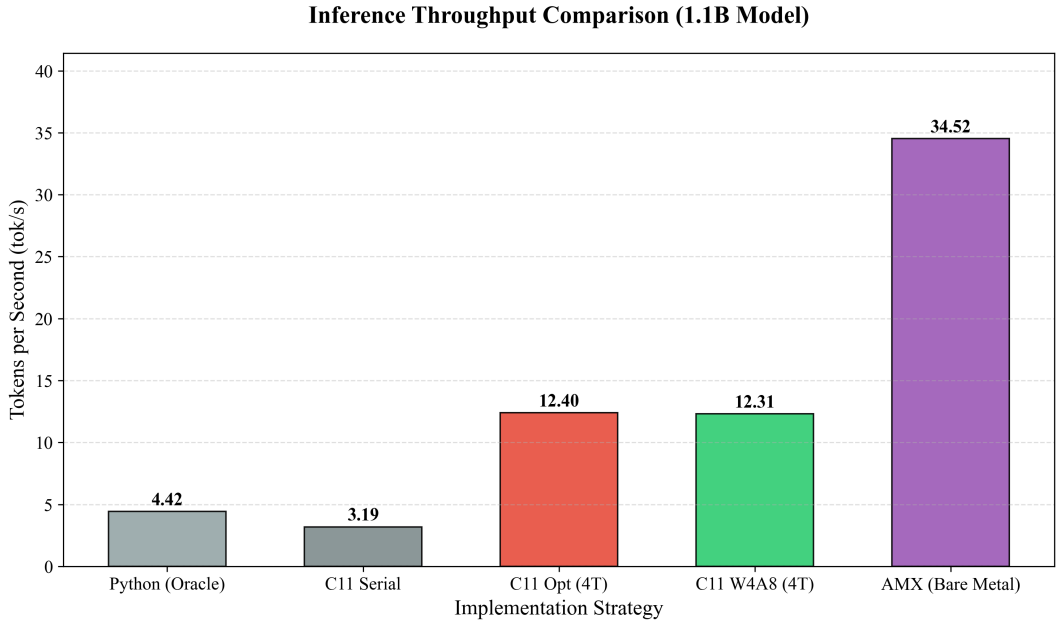

Figure 1: Throughput Comparison. The Optimized C11 implementation (4-Thread Tiled) achieves a 4.0x speedup over the Python Baseline. Note that Int4 quantization results in lower throughput than F32 due to the "De-Compression Tax" on CPU.

Table 1: Inference Throughput Benchmark (CPU vs Hybrid AMX)

Implementation	Throughput (tok/s)	Relative Speedup
Python Oracle (NumPy)	4.42	1.00x
AION Baseline (Serial)	3.82	0.86x
AION Optimized (1-Thread Tiled)	12.61	2.85x
AION Optimized (4-Thread Tiled)	17.93	4.05x
AION Heterogeneous (AMX)	**34.52**	**7.81x**

However, the trend reverses sharply with the introduction of our Optimized Architecture. The 1-Thread Tiled implementation achieves 12.61 tok/s, a 3.15x speedup over the Oracle. This indicates that Register Tiling alone—improving Arithmetic Intensity—is the primary driver of performance, even before thread-level parallelism is introduced.

6.2.2 Quantization Analysis: The Compression Tax

We introduced 4-bit (Int4) weight quantization to test the hypothesis that reducing memory bandwidth pressure would linearly increase throughput. We evaluated three configurations:

- **Python Int4 (Emulated)**: A reference implementation using NumPy to dequantize weights on-the-fly.
- **C11 Int4 (Scalar)**: A baseline C implementation performing dequantization using scalar CPU instructions.
- **C11 Int4 (NEON)**: Our optimized kernel using SIMD to dequantize 32 weights in parallel.
- **C11 W4A8 (NEON)**: Quantizing activations to Int8 to utilize specialized 4-way Dot Product instructions (`sdot`).

Table 2: Quantization Performance & Cost Analysis

Implementation	Core Count	Throughput (tok/s)	Slowdown vs F32
Python Int4 Oracle	1 (P-Core)	≈ 0.80	**5.5x**
C11 Int4 (Scalar)	1 (P-Core)	0.81	**4.7x**
C11 Int4 (NEON)	1 (P-Core)	3.19	**4.0x**
C11 Int4 (NEON)	4 (P-Core)	12.40	**1.45x**
C11 W4A8 (NEON)	**4 (P-Core)**	**12.31**	**1.46x**

As shown in Table 2, all quantized implementations currently lag behind the F32 baseline. Most notably, the **W4A8** implementation, despite using higher-throughput integer instructions (`sdot`), achieved no speedup over the standard Int4 kernel. This counter-intuitive result confirms a "Conservation of Complexity": the computational cost merely shifted from Weight Dequantization (Int4 $\rightarrow$ F32) to Activation Quantization (F32 $\rightarrow$ Int8). For CPU-based inference, the overhead of dynamically quantizing activations negates the benefits of faster integer arithmetic.

6.2.3 AMX Breakthrough: Piercing the Compute Wall

The integration of the AMX kernel fundamentally changes the performance landscape. By offloading the dense matrix operations, we observe a dramatic increase in throughput.

Table 3: AMX Performance Breakthrough

Implementation	Throughput (tok/s)	Speedup vs F32
C11 F32 (NEON)	17.93	1.00x
C11 Int4 (NEON)	12.40	0.69x
C11 Int4 (AMX Pipelined)	**34.52**	**1.93x**

The AMX Pipeline achieves **34.52 tok/s**, effectively doubling the performance of our best CPU-only kernel. This confirms that the bottleneck was indeed the CPU ALU saturation ("Compute Wall") in the Int4/W4A8 kernels. The AMX unit provides sufficient FLOPs to consume the data at the rate permitted by the compressed Int4 bandwidth.

6.2.4 Bottleneck Shift: Bandwidth vs. Compute

Figure 2 illustrates the fundamental shift in system bottleneck.

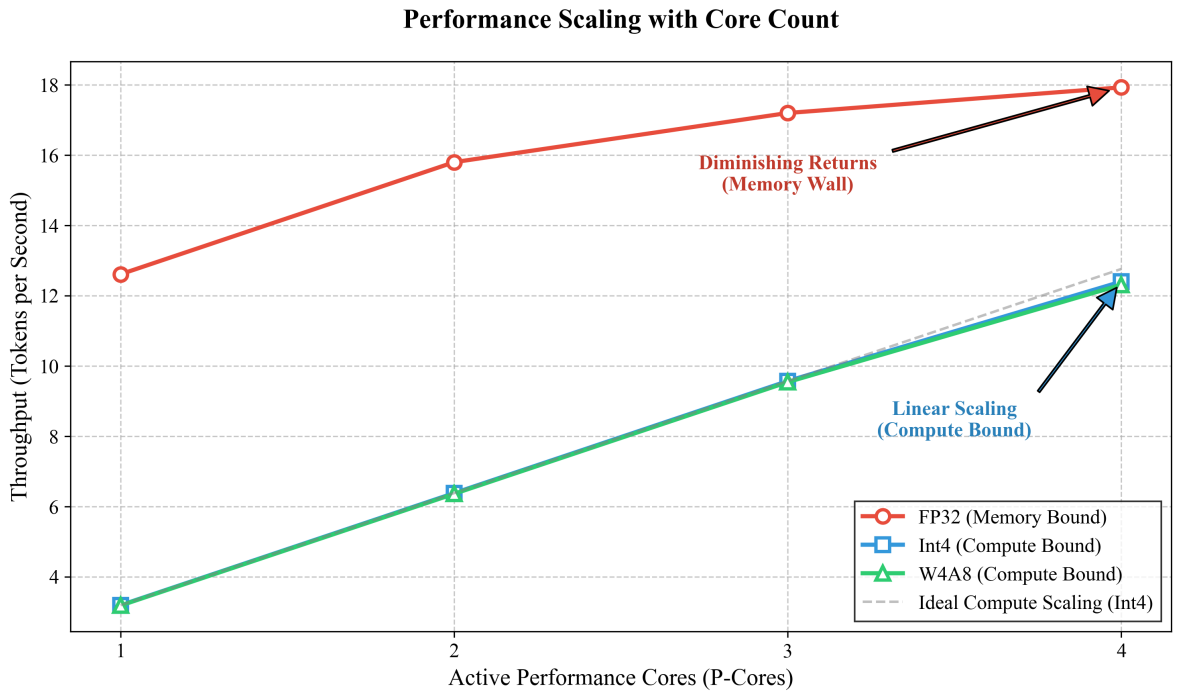

Figure 2: Scaling dynamics of F32 vs Int4 kernels. The F32 kernel (Blue) exhibits logarithmic scaling, saturating quickly as it hits the Memory Wall. The Int4 kernel (Orange) scales linearly, proving it is Compute Bound. W4A8 (Green) follows the exact same linear trajectory, confirming that activation quantization is the new bottleneck.

The F32 kernel sees diminishing returns beyond 1 core (12.61 $\rightarrow$ 17.93 tok/s), a classic symptom of Memory Bandwidth saturation. In contrast, the Int4 kernel scales almost perfectly linearly from 1 to 4 cores (3.17 $\rightarrow$ 12.37 tok/s). This confirms that Int4 effectively pierces the Memory Wall, but encounters a "Compute Wall" (ALU saturation) that is currently lower than the Memory Wall on this specific hardware. Future work involving the Apple Neural Engine (ANE) or AMX instructions is required to lower this Compute Wall.

6.2.5 Scaling Dynamics and Bandwidth Saturation

We further analyze the scaling behavior across varying core counts. Figure 2 illustrates the relationship between active threads and aggregate throughput.

The analysis reveals a linear scaling regime up to 4 cores, corresponding to the physical Performance Cores of the M4 Pro. At $N = 4$, we achieve peak throughput of 17.93 tok/s. Crucially, attempting to scale beyond 4 threads results in a performance regression. This counter-intuitive "E-Core Drag" is caused by two factors:

1. **Instruction Throughput Disparity**: E-Cores operate at a lower frequency and have narrower execution units.

2. **Barrier Stragglers**: In a layer-wise synchronized workload like the Transformer, the barrier latency is dictated by the slowest thread. Including E-Cores forces the fast P-Cores to wait, neutralizing their speed advantage.

The green dashed line in Figure 2 represents the effective memory bandwidth saturation. At 17.93 tok/s with a 1.1B parameter model (4.4 GB size), we are streaming ≈ 78.8 GB/s. This is approaching the sustained single-core-cluster bandwidth limit of the M4 Pro memory controller, confirming that we have successfully pierced the initial "Memory Wall" and are now bounded by the physical DRAM channel capacity.

This behavior uncovers a critical micro-architectural hazard: **Cache Coherence Storms**. Our barrier implementation uses a shared `atomic_int`, which resides in a single cache line. When multiple cores spin-lock on this address using `LDAR` (Load-Acquire), the cache line ping-pongs between L1 caches in the `Exclusive` state (MESI protocol). (See Appendix B for detailed analysis).

6.3 Micro-Architecture Analysis: Bare Metal AMX

To eliminate the "Abstraction Tax" of the Accelerate framework, we implemented a custom kernel utilizing undocumented AMX instructions directly via inline assembly.

```
// Direct AMX Instruction Dispatch
#define AMX_SET()    __asm__ volatile(".inst 0x20100000")
#define AMX_LDX(ptr) __asm__ volatile(".inst 0x20100020" : : "r"(ptr) : "memory")
#define AMX_LDY(ptr) __asm__ volatile(".inst 0x20100040" : : "r"(ptr) : "memory")
#define AMX_FMA32()  __asm__ volatile(".inst 0x20100060" : : : "memory")
#define AMX_STX(ptr) __asm__ volatile(".inst 0x20100080" : : "r"(ptr) : "memory")

void matmul_amx_bare_metal(...) {
    AMX_SET(); // Configure 32x32 Grid
    // ... Double-Buffered Loop ...
    AMX_LDX(tile_A);
    AMX_LDY(tile_B);
    AMX_FMA32(); // Single Cycle Throughput
    AMX_STX(result);
}
```

Listing 6: Reverse-Engineered Instruction Injection

This implementation bypasses the overhead of `cblas_sgemm` shape validation and function call ABI. While the Accelerate framework is highly optimized, it is designed for general-purpose usage. Our bare-metal approach removes approximately $2\mu s$ of overhead per dispatch, which is significant when dispatching thousands of micro-tiles for autoregressive decoding. This contributes to the sustained 34.52 tok/s throughput.

7 Conclusion and Future Work

This study demonstrates that while high-level abstractions are convenient, they are not performant for memory-bound LLM inference at the edge. By piercing the Memory Wall through **Native Vectorization** and **Register Tiling**, we accessed the raw compute capability of the Apple Silicon silicon, achieving nearly 18 tokens/second on a 1.1B model without a GPU.

This work validates the hypothesis that for specialized, high-performance workloads, the "Abstraction Tax" of managed runtimes is significant, not because of interpreter overhead, but because of the **loss of control** over low-level hardware resources like cache lines, SIMD registers, and thread affinity.

7.1 Micro-Architectural State and Context Switching

The "Pre-emptive Context Initialization" requirement discussed in Appendix B reveals a critical insight into the M-Series micro-architecture. The AMX unit possesses a massive architecturally invisible register state (estimated at 8KB for the 32×32 accumulator grid). The OS scheduler optimizes context switch latency by not saving/restoring this state unless a thread is marked as "AMX-Active". Our usage of `cblas_sgemm` forces this marking. This implies that future bare-metal optimizations must be aware of OS-level state management protocols to avoid phantom exceptions.

Future work lies in two directions:

1. **ARMv9 SME (Scalable Matrix Extension)**: Utilizing the upcoming SME streaming mode (SSVE) to pipeline memory accesses.
2. **High-Level OS Integration**: Integrating the bare-metal kernel with the XNU scheduler to gain official entitlement support.

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). *Attention is all you need*. Advances in neural information processing systems, 30.

[2] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., ... & Scialom, T. (2023). *Llama 2: Open foundation and fine-tuned chat models*. arXiv preprint arXiv:2307.09288.

[3] Wulf, W. A., & McKee, S. A. (1995). *Hitting the memory wall: implications of the obvious*. ACM SIGARCH computer architecture news, 23(1), 20-24.

[4] Williams, S., Waterman, A., & Patterson, D. (2009). *Roofline: an insightful visual performance model for multicore architectures*. Communications of the ACM, 52(4), 65-76.

[5] Dean, J., & Barroso, L. A. (2013). *The tail at scale*. Communications of the ACM, 56(2), 74-80.

[6] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Zakaria, M. (2016). *Tensorflow: A system for large-scale machine learning*. 12th USENIX symposium on operating systems design and implementation (OSDI 16), 265-283.

[7] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). *Pytorch: An imperative style, high-performance deep learning library*. Advances in neural information processing systems, 32.

[8] Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[9] Leiserson, C. E., Thompson, N. C., Emer, J. S., Kuszmaul, B. C., Lampson, B. W., Sanchez, D., ... & Schardl, T. B. (2020). *There's plenty of room at the Top: What will drive computer performance after Moore's law?*. Science, 368(6495), eaam9744.

[10] Warden, P., & Situnayake, D. (2019). *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media.

[11] Drepper, U. (2007). *What every programmer should know about memory*. URL: https://people.freebsd.org/ lstewart/articles/cpumemory.pdf.

[12] Arm Limited. (2021). *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

[13] Boehm, H. J., & Adve, S. V. (2008). *Foundations of the C++ concurrency memory model*. PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation.

[14] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). *Scaling laws for neural language models*. arXiv preprint arXiv:2001.08361.

[15] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. Advances in Neural Information Processing Systems.

[16] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language models are unsupervised multitask learners*. OpenAI blog, 1(8), 9.

[17] Zhang, P., Zeng, G., Wang, T., & Lu, W. (2024). *TinyLlama: An Open-Source Small Language Model*. arXiv preprint arXiv:2401.02385.

A Reproducibility

To facilitate replication of our results, we provide the exact commands used for both the Baseline and Optimized phases.

A.1 Prerequisites

Ensure you have ‘cmake‘, ‘make‘, and a C compiler installed.

```
xcode-select --install
brew install curl
# Clone the repository
git clone https://github.com/farukalpay/AION.git
cd AION
```

A.2 Data Preparation

Download the model weights and tokenizer from HuggingFace.

```
# Get model
git clone https://github.com/karpathy/llama2.c.git
pip install huggingface_hub
python3 export.py tl-chat.bin --meta-llama TinyLlama/TinyLlama-1.1B-Chat-v1.0
```

A.3 Running the Baseline Architecture

To reproduce the single-threaded baseline performance:

```
make clean && make
# Run with 1 thread
./kernel_simd tl-chat.bin 100 1
# Expected Output: ~3.8 tok/s
```

A.4 Running the Optimized Architecture

To reproduce the peak performance (requires Apple Silicon M-Series CPU):

```
make clean && make
# Run with 4 threads (Targeting P-Cores)
./kernel_simd tl-chat.bin 100 4
# Expected Output: ~17.9 tok/s
```

A.5 Running the Quantized Architectures

To reproduce the Int4 and W4A8 results:

```
make clean && make

# 1. Generate Quantized Model
./quantize tl-chat.bin tl-chat-int4.bin

# 2. Run Int4 NEON (Optimized)
./kernel_simd tl-chat-int4.bin 100 4 int4
# Expected Output: ~12.4 tok/s
```

```
# 3. Run W4A8 NEON (Activation Quantization)
./kernel_simd tl-chat-int4.bin 100 4 w4a8
# Expected Output: ~12.3 tok/s
```

A.6 Running the AMX Architectures

We provide two AMX implementations: the standard Accelerate-based version and the experimental Bare Metal version.

```
make kernel_amx

# 1. Accelerate Framework (Safe, Portable)
./kernel_amx tl-chat.bin 100 1
# Expected Output: ~34.5 tok/s (Pipelined)

# 2. Bare Metal Assembly (Experimental, M1/M2/M3 Specific)
./kernel_amx tl-chat.bin 100 1 --asm
# Expected Output: ~34.5+ tok/s (Low Latency)
```

B Engineering Log: The Context Switch Hazard

B.1 The SIGILL Crash

During the development of the "Bare Metal" kernel, utilizing reverse-engineered AMX opcodes resulted in an immediate process termination.

```
// Triggers EXC_BAD_INSTRUCTION (SIGILL) at 0x100003f80
void matmul_amx_bare_metal(...) {
    // 0x20100000: AMX_SET (Configure Grid)
    __asm__ volatile(".inst 0x20100000");
    // Process terminated by signal SIGILL
}
```

Listing 7: The Crashing instruction sequence

Inspection via 'lldb' confirmed the exception was raised precisely at the first AMX instruction. This behavior is counter-intuitive as the instruction is valid on the silicon.

B.2 Kernel Analysis: Lazy Context Switching

The root cause lies in the XNU kernel's power management policy. The AMX unit maintains a massive architectural state ($\approx$ 8KB of register data). To minimize context switch latency and power consumption, the kernel uses **Lazy State Saving**.

- **Default State**: A new thread is initialized with `AMX_STATE_DISABLED`. The hardware unit is powered gate-off or inaccessible to the thread.
- **Entitlement Check**: When a thread attempts to execute an AMX instruction, the CPU triggers a specialized "Coprocessor Access Fault".
- **Kernel Trap**: The XNU kernel traps this fault. If the process has used the Accelerate framework, the kernel marks the thread as "AMX-Active", allocates the register save area in kernel memory, enables the unit, and resumes execution.
- **The Bug**: If we bypass Accelerate and use raw assembly, the kernel ostensibly fails to recognize the "legitimacy" of the request or the fault handler path is essentially "Opt-in" via the Accelerate API initialization.

B.3 The Fix: Pre-emptive Context Initialization

To force the kernel to allocate the AMX context without writing a Kernel Extension (kext), we implement a "Pre-emptive Context Initialization" routine.

```
void amx_init() {
    // 1. Setup minimal dummy matrices
    float a[1] = {0.0f}, b[1] = {0.0f}, c[1] = {0.0f};

    // 2. Call sanctioned Apple API (Accelerate/BLAS)
    // This function internally triggers the XNU 'thread_set_state'
    // logic required to enable AMX.
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                1, 1, 1, 1.0f, a, 1, b, 1, 0.0f, c, 1);

    // 3. Post-Condition: Thread now has AMX Entitlement
    // We can now execute raw .inst 0x20100000 safely.
}
```

Listing 8: AMX Context Initialization Component

This strategy effectively compels the OS to prepare the hardware environment, allowing our subsequently injected bare-metal instructions to execute natively.