Comparing Quantization Across Hardware

Benchmarking LLM Inference on Edge Devices

Mike Doan · Jenny Dinh

CSC 4228 & 6228 — Security in IoT

Georgia State University

Fall 2024

Introduction & Background

The Rise of Local LLM Deployment

- Growing interest in running LLMs **locally** rather than via cloud APIs
- Target environments:
  - **Edge devices:** smartphones, embedded systems
  - **Consumer hardware:** laptops, gaming PCs
  - **Enterprise GPUs:** data center deployments

The Memory Challenge

- LLMs contain **massive weights** requiring significant memory

- Activations during inference are also substantial
- Example: Llama-3.1-8B requires 16GB in FP16

Background: What is Quantization?

Quantization reduces the precision of model weights and activations to decrease memory footprint and improve inference speed.

Precision	Bits per Weight	Memory (8B model)
FP16 / BF16	16 bits	16 GB
INT8 (W8A8)	8 bits	8 GB
INT4 (W4A8)	4 bits	4 GB
NF4	4 bits	4 GB

Trade-off: Lower precision → smaller memory, faster inference, but potential accuracy loss

Background: Quantization Strategies

Weight-Only Quantization

- **INT8**: 8-bit integer weights
- **INT4**: 4-bit integer weights
- **NF4**: 4-bit NormalFloat (QLoRA)

KV-Cache Quantization

- Reduces memory for long contexts
- INT8 or INT4 KV cache

Activation Quantization

- **W8A8**: Both weights and activations in INT8
- **W4A4**: Aggressive 4-bit for both

Group-wise Quantization

- **GPTQ:** Post-training quantization
- **AWQ:** Activation-aware weights

Purpose & Motivation

Research Questions

1. How do different **quantization strategies** impact inference performance across hardware?
2. What are the **accuracy-efficiency trade-offs** for edge deployment?
3. Can we create a **reproducible benchmarking framework** for the community?

Gap in Existing Work

- Few plug-and-play toolkits for **cross-platform quantization sweeps**
- Limited public data on **edge device performance**
- Need for standardized comparison methodology

Target Hardware & Scope

Edge Devices

- **Google Pixel 7/8** (Android)
  - ARM CPU + Adreno GPU
  - OpenCL backend

Consumer Hardware

- **MacBook** (Apple Silicon)
  - Metal backend
  - Unified memory

Runtimes Evaluated

- **llama.cpp** (GGUF format)
- **tinygrad** (Metal/OpenCL)
- MLC-LLM (TVM-based)

Models

- Llama-3.2-1B-Instruct
- Qwen2.5-1.5B
- (Extensible to larger models)

System Architecture

Benchmark Scripts

tinygrad_benchmark.py
llamacpp_benchmark.py
mlc_benchmark.py

Data Collection

*_collate.py
*_parse.py CSV output

Analysis & Viz

visualize_benchmarks.py
benchmark_analysis.py

Key Design Principles

- **Reproducibility:** Seeded runs, UUID tracking
- **Extensibility:** Easy to add new backends/devices

- **Standardized schema:** Consistent CSV format across all backends

Benchmark Workflow

0. Pixel Setup
Termux + SSH + OpenCL
croc, cmake

1. Model Prep
Download models
Quantize (GGUF/HF)

2. Run Benchmarks
Execute on devices
Output: .txt logs

3. Collate
Parse logs
Structured CSV

4. Visualize
Generate plots
.png files

5. Presentation
Embed plots
typst compile

Data Schema

Each benchmark run captures:

```
BenchmarkRow:
  step: int                     # Token generation step
  enqueue_latency_ms: float     # Time to enqueue
  total_latency_ms: float       # Total time
  tokens_per_sec: float         # Throughput
  memory_throughput_gb_s: float
  param_throughput_gb_s: float
  platform: str                 # Darwin, Linux, Android
  device: str                   # Metal, OpenCL, CUDA
  hostname: str                 # Machine identifier
  size: str                     # Model size (1B, 3B, 8B)
  quantize: str                 # nf4, int8, float16, default
  seed: int                     # For reproducibility
  uuid: str                     # Unique run identifier
```

Implementation: Tinygrad Backend

- Pure Python ML framework with Metal/OpenCL support
- Server mode for OpenAI-compatible API

```
# Start inference server
PYTHONPATH=./deps/tinygrad/ python tinygrad_benchmark.py \
    --port 7776 --size 1B
```

Quantization Support

- `default`: No quantization
- `nf4`: 4-bit NormalFloat
- `int8`: 8-bit integer
- `float16`: Half precision

Implementation: llama.cpp Backend

- C/C++ implementation with GGUF model format
- Highly optimized for CPU and GPU inference

```
# Run benchmark
python llamacpp_benchmark.py

# Collate results
python llamacpp_collate.py
```

Key Features

- Native quantization support in model files
- Metal backend for Apple Silicon
- OpenCL for Android devices

Implementation: Pixel Device Setup

Bootstrap (Termux)

```
# Install from F-Droid
pkg update && pkg upgrade
pkg install python git openssh
pkg install cmake clang ninja
pkg install golang

# Install uv & croc
curl -LsSf astral.sh/uv/install.sh | sh
go install github.com/schollz/croc/v10@latest
```

SSH Access (Tailscale)

```
# On Pixel
sshd
passwd
id  # Get username (u0_a190)

# From host
ssh u0_a190@<tailscale-ip> -p 8022
```

OpenCL GPU Setup

```
# Install OpenCL for Adreno GPU
pkg install opencl-headers opencl-vendor-driver

# Set environment
export LD_LIBRARY_PATH=/system/vendor/lib64:
$LD_LIBRARY_PATH
export GPU=1
export OPENCL=1
```

Build & Transfer

```
# Build llama.cpp with OpenCL
cmake .. -DGGML_OPENCL=ON
cmake --build . -j$(nproc)

# Transfer models with croc
croc send model.gguf
```

Full setup guide: `docs/WORKFLOW.md` & `setup/PIXEL-SSH.md`

LLM Evaluation: Verifiers Framework

Beyond raw throughput, we measure **downstream task accuracy:**

Setup

```
# Start model server
python tinygrad_benchmark.py \
    --port 7776 --size 1B

# OpenAI proxy (streaming)
python openai_proxy.py \
    --backend-port 7776 \
    --proxy-port 7777
```

Run Evaluation

```
# GSM8K math benchmark
OPENAI_API_KEY=dummy \
uv run vf-eval gsm8k \
    -m local \
    -b http://localhost:7777/v1 \
    -n 20 -r 1 -t 512
```

Available benchmarks: gsm8k, math, gpqa, simpleqa, wordle

Metrics Collected

Performance Metrics

Latency

- Time to first token (TTFT)
- Per-token generation time

Throughput

- Tokens per second (tok/s)
- Memory bandwidth (GB/s)

Resource Metrics

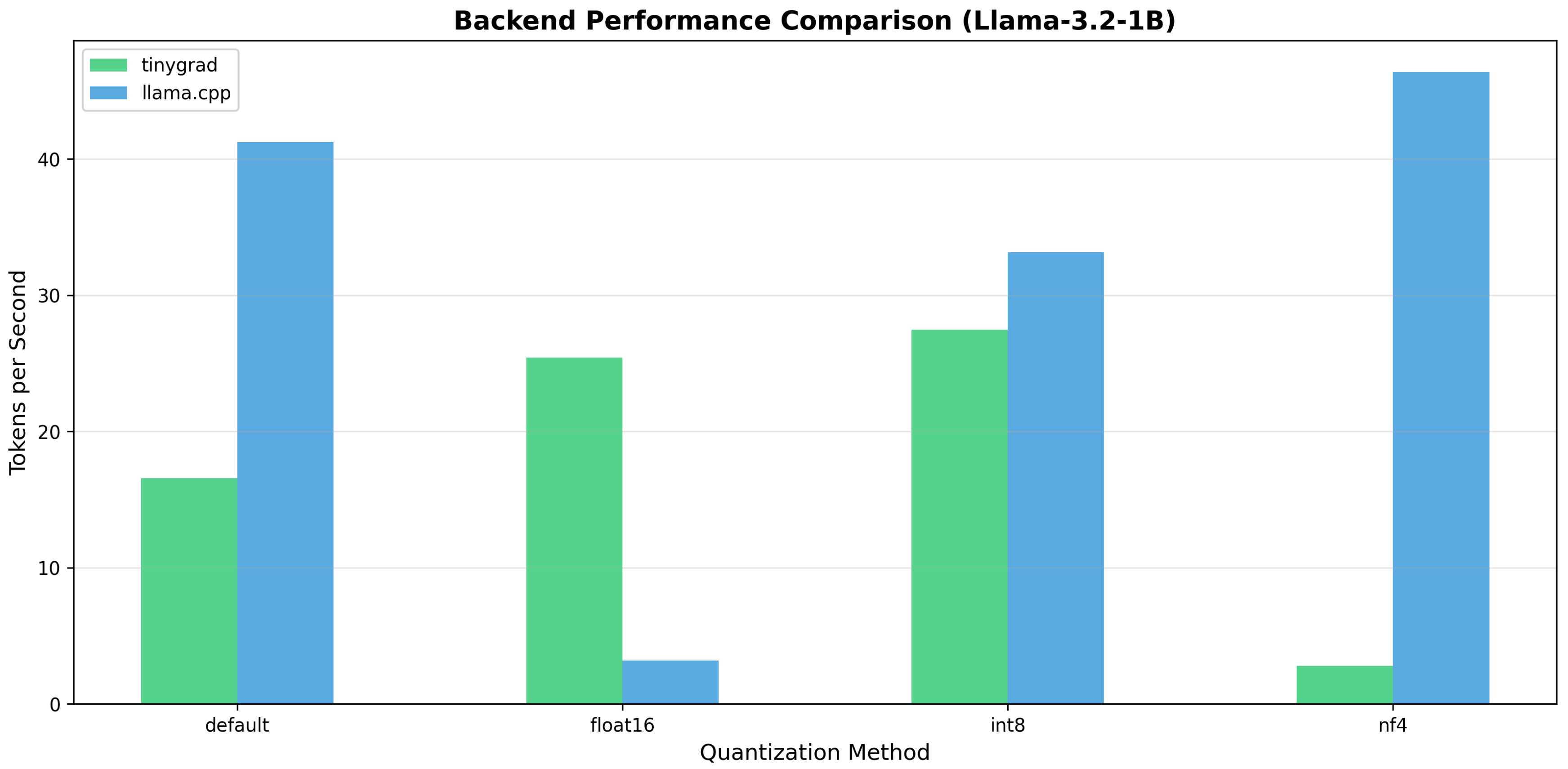
Memory

- Model load size (MB)
- Peak utilization

Accuracy

- Downstream benchmark scores
- (GSM8K, MATH, etc.)

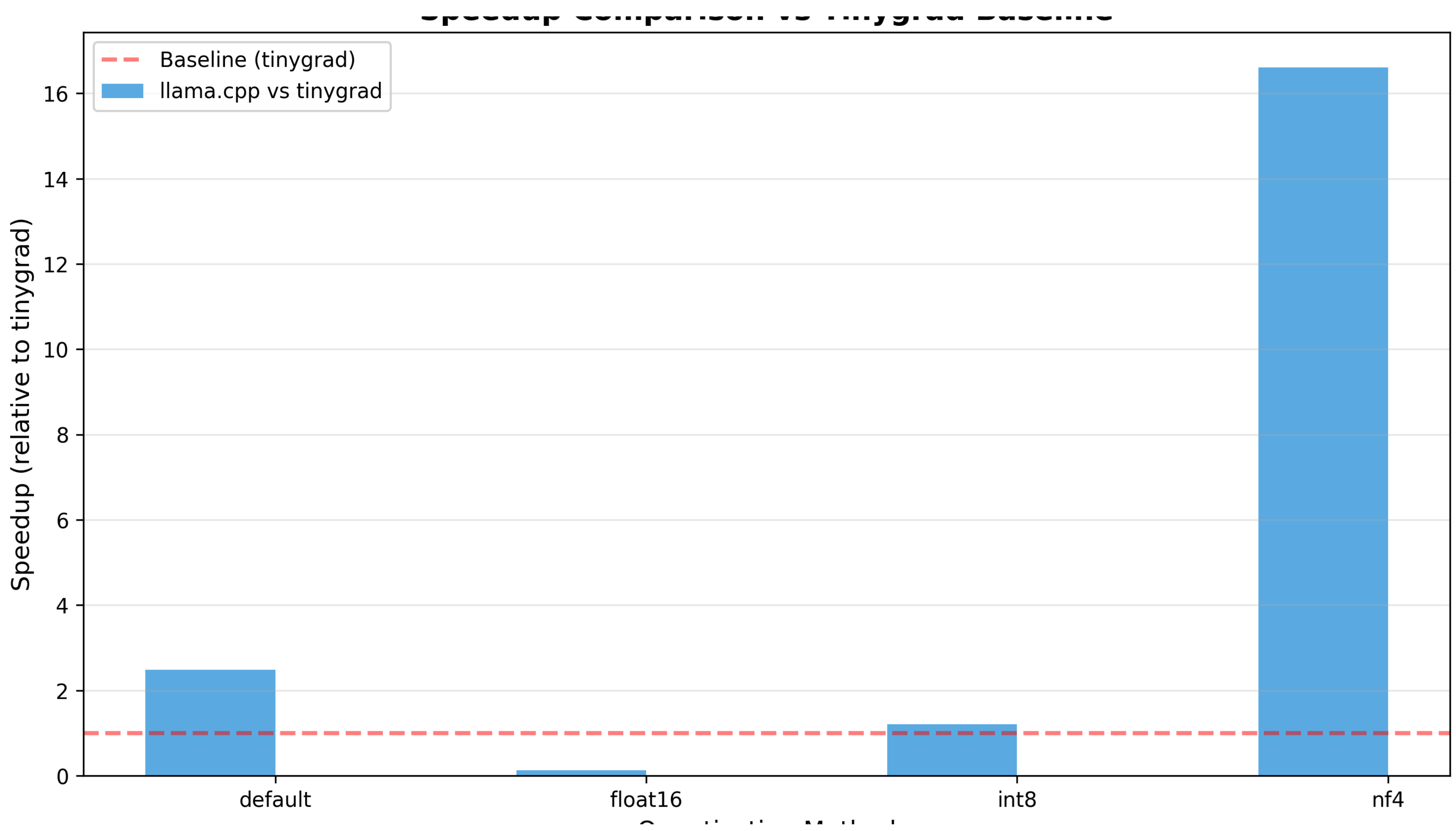
Results: Throughput Comparison



Key findings:

- llama.cpp excels with NF4 (46.4 tok/s) and default (41.3 tok/s)
- tinygrad performs best with INT8 (27.5 tok/s) and float16 (25.4 tok/s)
- Performance varies significantly by quantization method

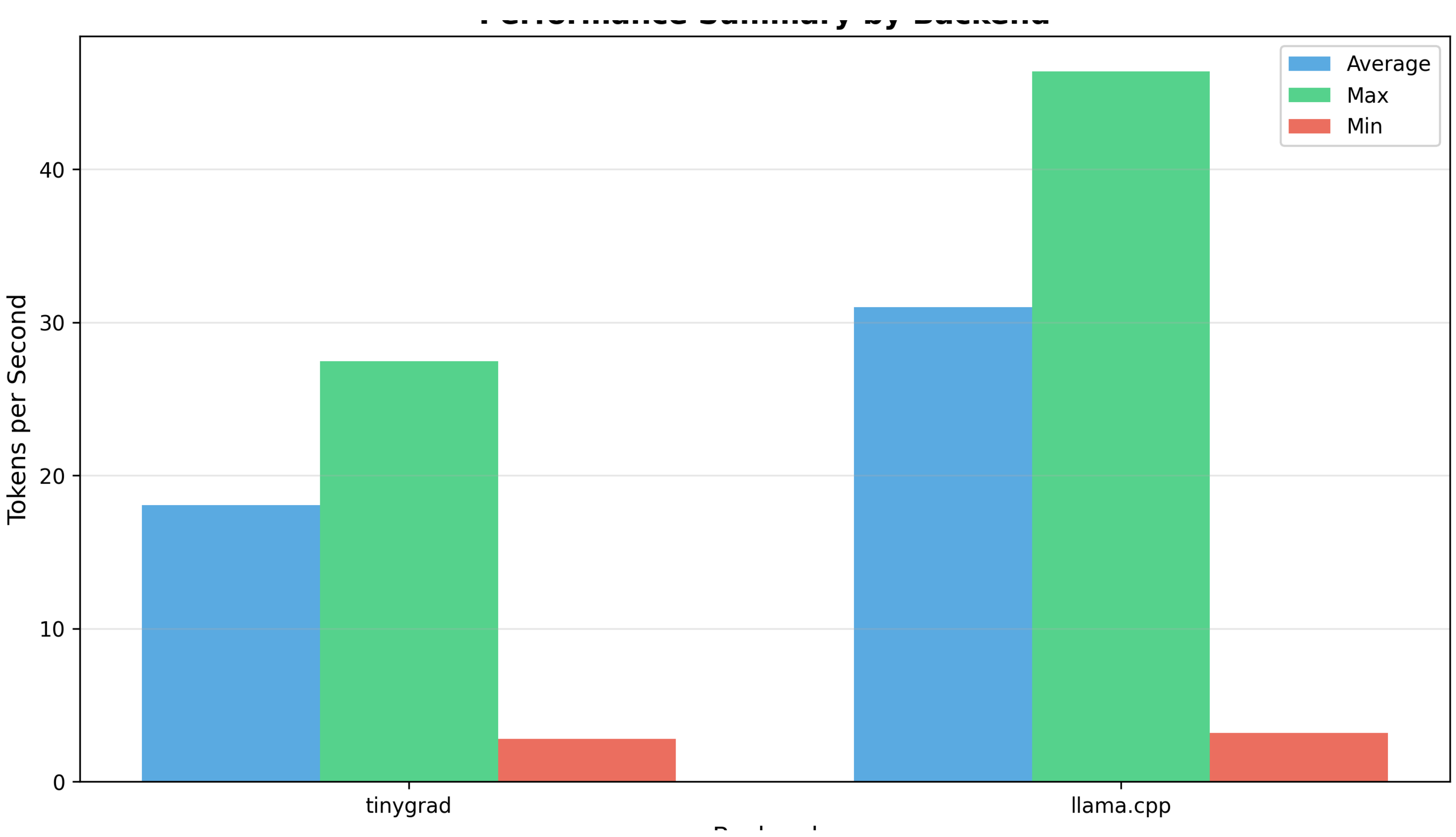
Results: Speedup Analysis


Baseline (tinygrad)
llama.cpp vs tinygrad
Speedup (relative to tinygrad)
0
2
4
6
8
10
12
14
16
default
float16
int8
nf4

Observations:

- llama.cpp shows 16.6x speedup over tinygrad for NF4
- tinygrad leads by 8.0x for float16 workloads
- INT8 and default show modest llama.cpp advantage (1.2-2.5x)

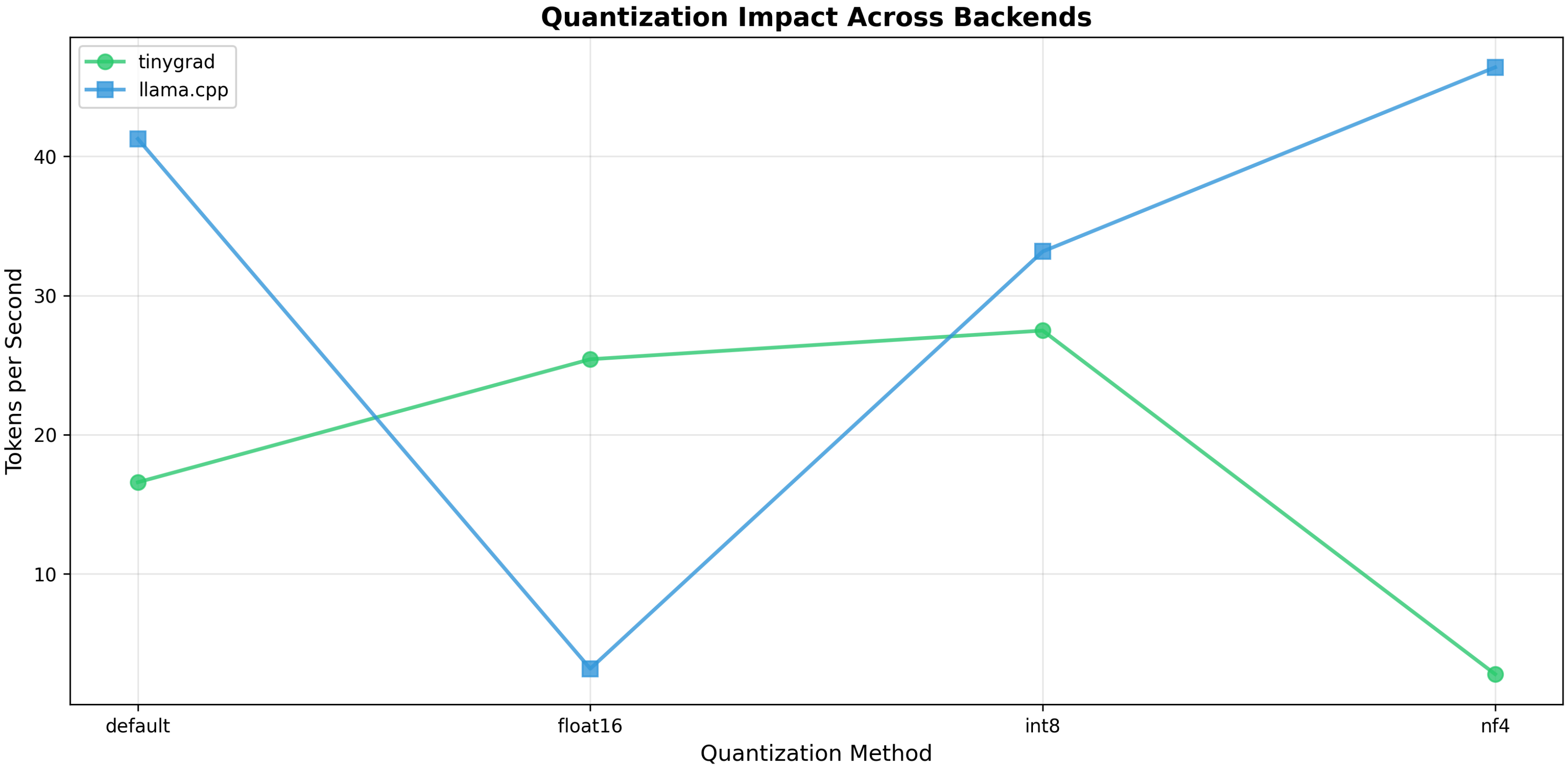
Results: Performance Summary


Average
Max
Min
Tokens per Second
0
10
20
30
40
tinygrad
llama.cpp

Backend comparison:

- llama.cpp: Average 31.0 tok/s, peak 46.4 tok/s (NF4)
- tinygrad: Average 18.1 tok/s, peak 27.5 tok/s (INT8)
- Overall llama.cpp shows 1.7x better average performance

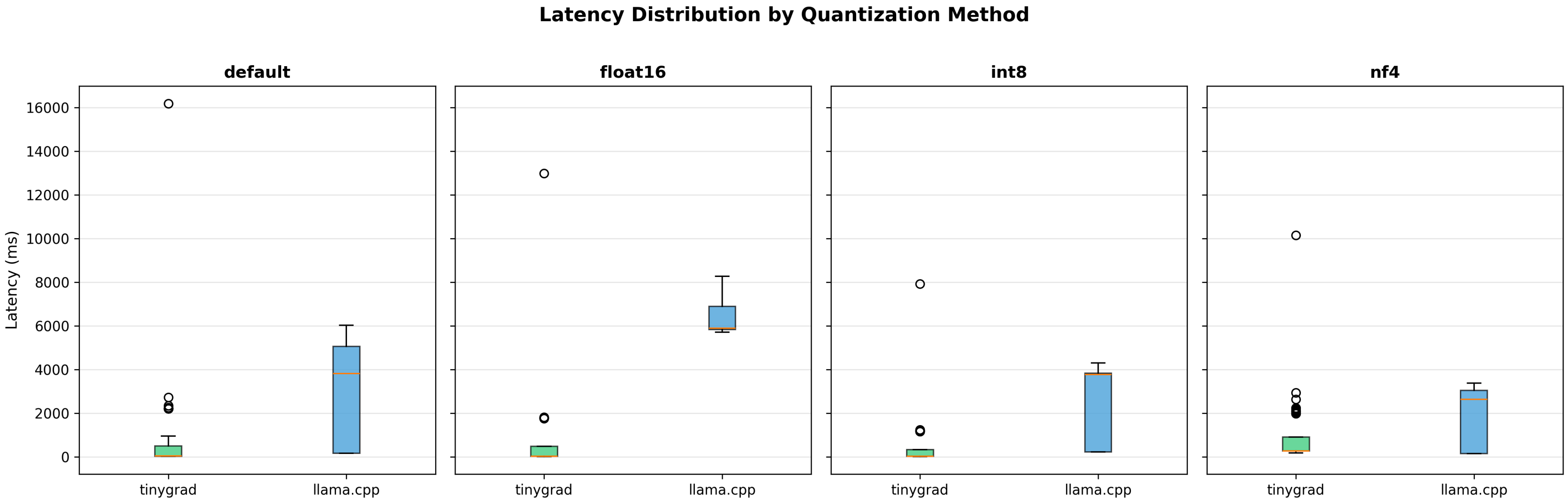
Results: Quantization Impact



Trends:

- Different backends favor different quantization strategies
- llama.cpp benefits most from aggressive quantization (NF4)
- tinygrad shows more consistent performance across methods

Results: Latency Analysis

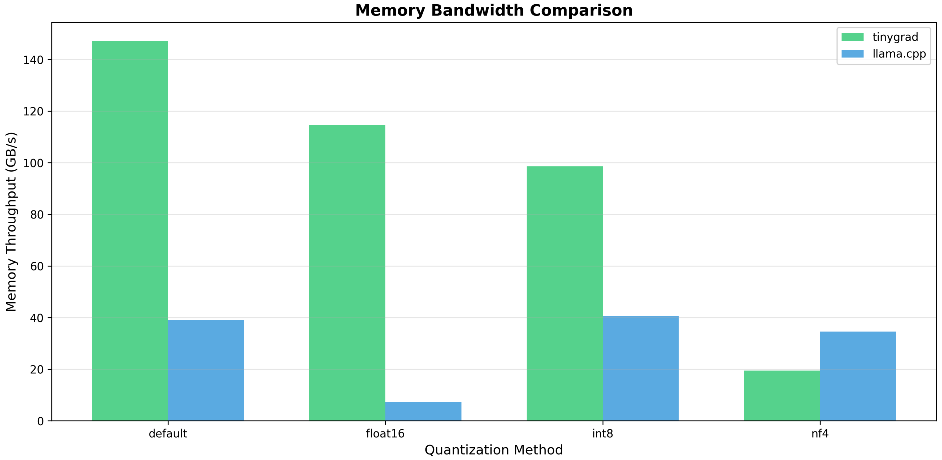


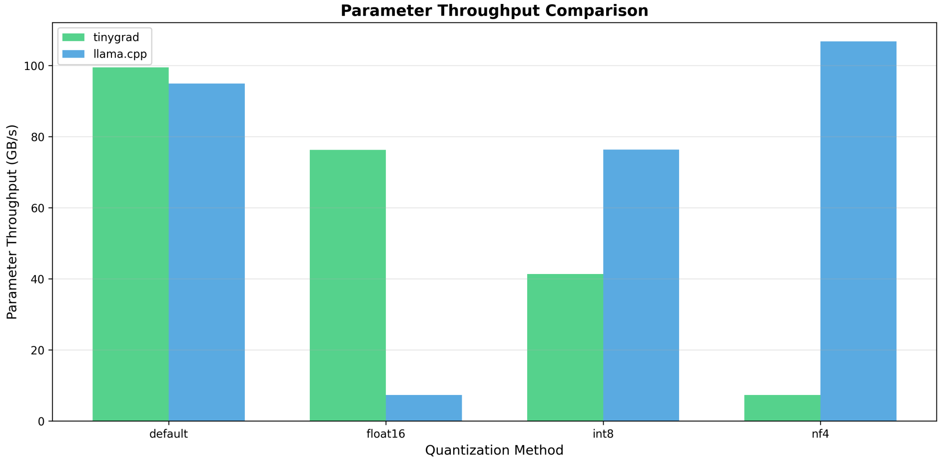
Latency variance:

- Box plots show distribution of per-token generation time
- llama.cpp shows lower variance for NF4 and default

- tinygrad has more consistent latency across quantizations

Results: Memory & Parameter Throughput

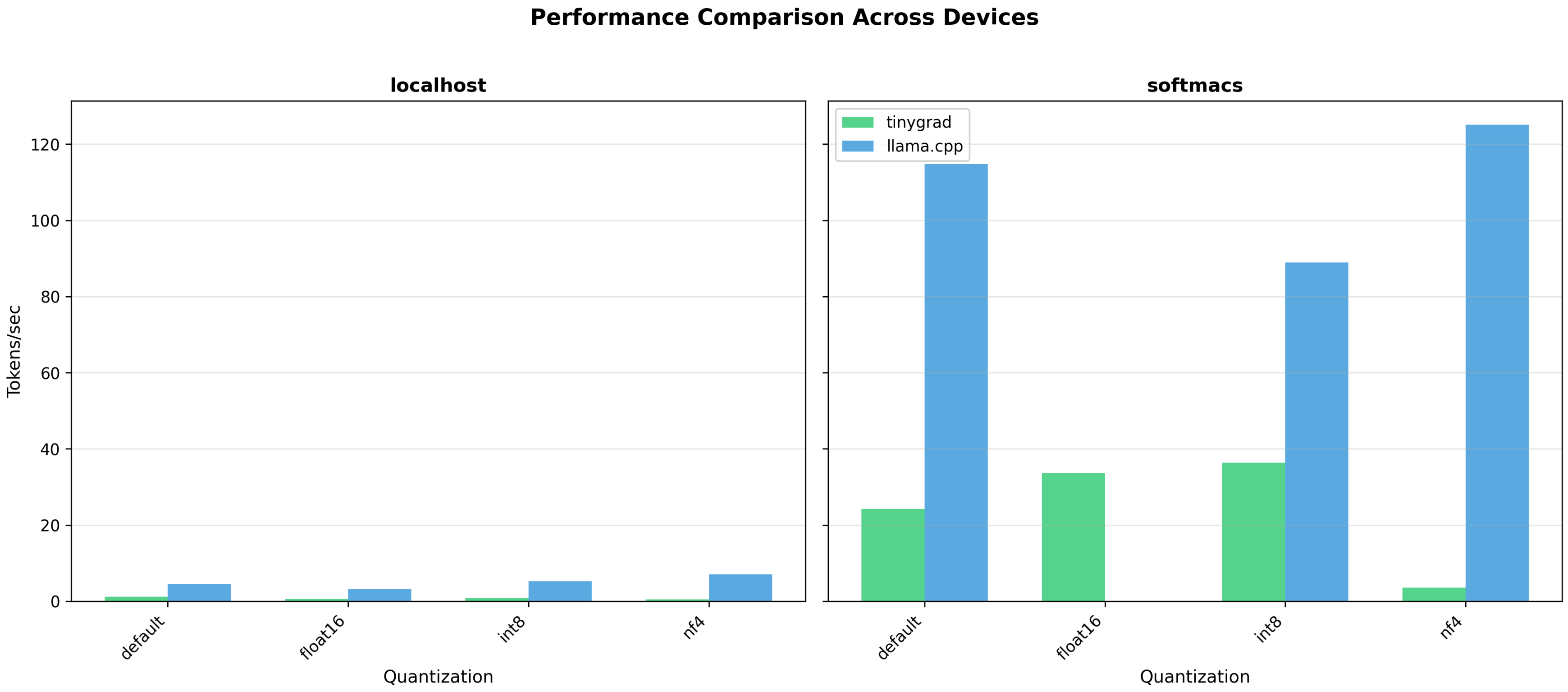




Observations:

- Memory bandwidth closely correlates with tokens/sec
- Parameter throughput shows compute efficiency per quantization

Results: Multi-Device Comparison



Cross-device insights:

- MacBook (softmacs) shows stronger performance overall
- Android device (localhost) exhibits different characteristics
- Backend choice impacts relative performance across hardware

Results: Performance Summary Table

Backend	Default	FP16	INT8	NF4
llama.cpp	41.3 tok/s	3.2 tok/s	33.2 tok/s	46.4 tok/s
tinygrad	16.6 tok/s	25.4 tok/s	27.5 tok/s	2.8 tok/s

Note: Results from Llama-3.2-1B-Instruct on MacBook (Metal). Performance varies with context length and workload.

Challenges Encountered

Software Compatibility

- Different backends for ARM vs x86
- MLC-LLM build complexity

Implementation Inconsistencies

- Quantization standards differ across backends
- Leads to performance discrepancies

Engineering Challenges

- Creating durable benchmarking suite
- Handling device-specific configurations

Android Setup

- SSH into Pixel devices
- OpenCL library path configuration

Contributions

Reproducible Framework

- Open-source benchmarking toolkit
- Standardized data format
- Easy to extend to new devices

Public Codebase

- MIT licensed
- Well-documented
- GitHub repository

Empirical Results

- Cross-device performance data
- Quantization trade-off analysis
- Downstream accuracy evaluation

Community Benefit

- Others can benchmark their devices

- Append results to shared data bank

Future Work

- **Enterprise GPUs**: Extend benchmarks to A100, H100
- **More Models**: Phi-3 mini, Qwen2.5-3B, Llama-3.1-8B
- **Energy Profiling**: Watt/token measurements
- **KV-Cache Quantization**: INT8/INT4 cache strategies
- **Longer Contexts**: Prefill tokens 256, 1024, 2048

Potential Extensions

- Automated setup script (`curl | sh`)
- CI/CD backend for continuous benchmarking
- Public data aggregation platform

Conclusion

- **Quantization enables practical edge LLM deployment**
  - 4-bit models run on smartphones with acceptable performance
- **Trade-offs are workload-dependent**
  - Memory-constrained? → INT4/NF4
  - Accuracy-critical? → INT8 or FP16
- **Framework enables reproducible research**
  - Standardized benchmarks across diverse hardware
  - Community can contribute additional results

Code: github.com/spikedoanz/t-eai-project

References

1. **llama.cpp** — Georgi Gerganov et al. High-performance LLM inference in C/C++.
   https://github.com/ggerganov/llama.cpp
2. **tinygrad** — George Hotz et al. A simple deep learning framework.
   https://github.com/tinygrad/tinygrad
3. **MLC-LLM** — Machine Learning Compilation for LLMs.
   https://github.com/mlc-ai/mlc-llm
4. **Verifiers** — Prime Intellect. LLM evaluation framework.
   https://github.com/PrimeIntellect-ai/verifiers
5. **QLoRA** — Dettmers et al. (2023). Efficient Finetuning of Quantized LLMs.
6. **GPTQ** — Frantar et al. (2022). Accurate Post-Training Quantization.
7. **AWQ** — Lin et al. (2023). Activation-aware Weight Quantization.

Thank You!

Questions?

Mike Doan · Jenny Dinh

github.com/spikedoanz/t-eai-project