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)

Group-wise Quantization

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

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

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

Runtimes Evaluated

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

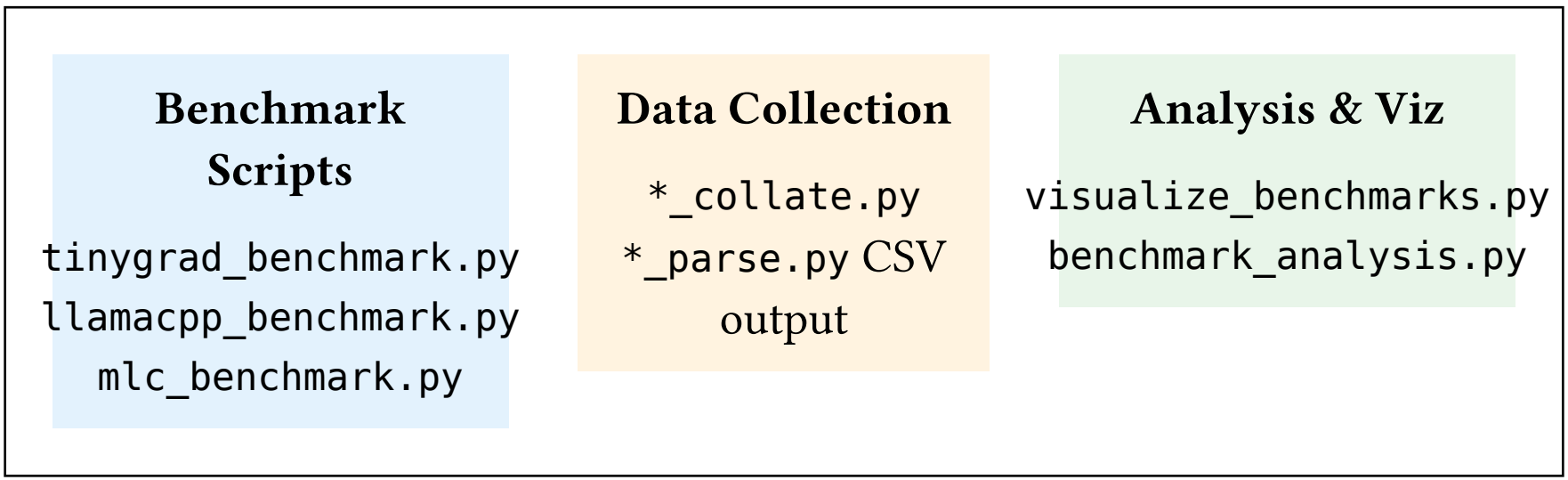
Consumer Hardware

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

Models

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

System Architecture



Key Design Principles

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

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

Data Schema

Each benchmark run captures:

```
BenchmarkRow:
  step: int                    # Token generation step
  enqueue_latency_ms: float    # Time to enqueue operation
  total_latency_ms: float      # Total time for step
  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

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

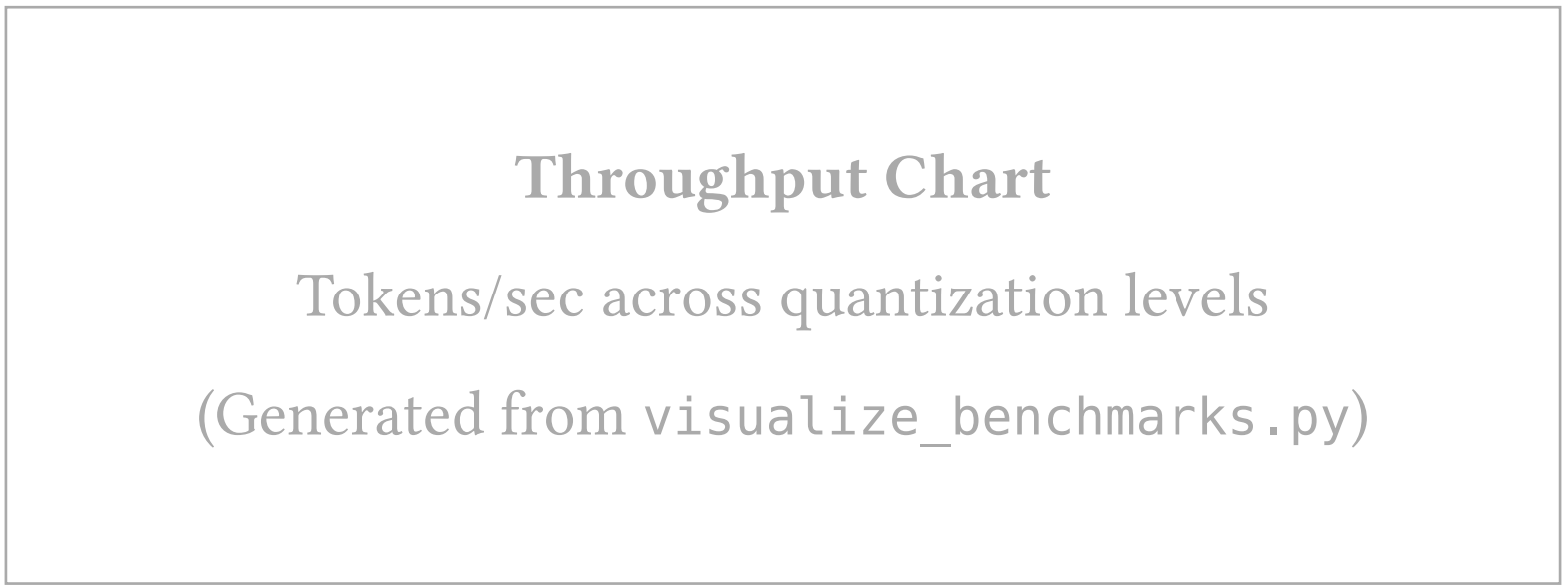
Memory

- Model load size (MB)
- Peak utilization

Accuracy

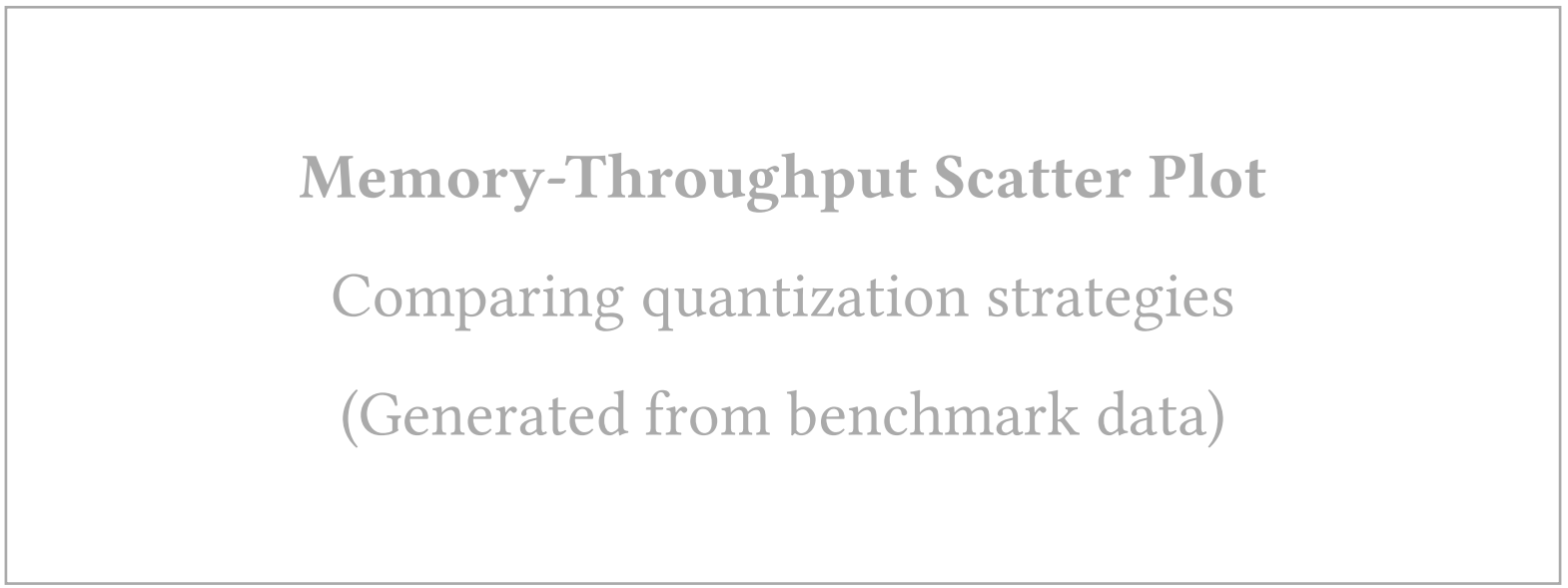
- Downstream benchmark scores
- (GSM8K, MATH, etc.)

Results: Throughput Comparison



Key finding: NF4 quantization achieves 3-5 tok/s on MacBook M-series with 1B model

Results: Memory vs Throughput Trade-offs



Observation: INT4/NF4 provides best memory-performance ratio for edge deployment

Results: Device Comparison

Device	FP16	INT8	NF4
MacBook (Metal)	2 tok/s	3.5 tok/s	4 tok/s
Pixel 7 (OpenCL)	TBD	TBD	TBD
Pixel 8 (OpenCL)	TBD	TBD	TBD

Note: Results are from Llama-3.2-1B-Instruct model. Actual numbers vary with context length and workload.

Challenges Encountered

Software Compatibility

- Different backends for ARM vs x86
- MLC-LLM build complexity
  - "Opted to skip due to finicky build process"

Engineering Challenges

- Creating durable benchmarking suite
- Handling device-specific configurations
  - Pixel 7 OpenCL requires special env vars

Implementation Inconsistencies

- Quantization standards differ across backends
- Leads to performance discrepancies

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

Empirical Results

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

Public Codebase

- MIT licensed
- Well-documented
- GitHub repository

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? $\longrightarrow$ INT4/NF4
  - Accuracy-critical? $\longrightarrow$ 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`