Mixed-Precision Inference on GPUs

Benchmarking FP32, FP16, and BLAS-Optimized Neural Network Inference

GPU Computing Project

Option 2 — Mixed Precision with Row-Wise Scaling

February 27, 2026

Outline

1. Problem & Objectives
2. Background Concepts
3. The Six Precision Modes
4. Architecture & Implementation
5. Key Algorithms
6. Metrics & Methodology
7. Optimizations & Challenges
8. Results & Analysis
9. Conclusions & Future Work

Problem Statement

The Core Challenge

GPU inference workloads are **memory-bandwidth bound**: time is spent moving data, not computing. Reducing precision (FP32 $\rightarrow$ FP16) halves traffic—but at what accuracy cost?

Central research question:

How does numerical precision format affect GPU inference performance, memory usage, and result accuracy—and can row-wise weight scaling recover accuracy lost to FP16 quantization?

Project Goal & Research Questions

Goal: Implement a two-layer MLP and benchmark it across **6 precision strategies**, measuring:

- Execution time (ms)
- Throughput (GFLOPS)
- Memory bandwidth (GB/s)
- Memory footprint (MB)
- Accuracy vs. FP32 baseline (MSE)

Research Questions:

1. How much faster is FP16 than FP32 on the same custom kernel?
2. Does row-wise scaling meaningfully reduce FP16 quantization error?
3. How much faster is CLBlast GEMM than a hand-written kernel?
4. Does CLBlast Mixed (FP16 storage $\rightarrow$ FP32 compute) offer the best of both worlds?

Success Criteria

Criterion	Definition
Correctness	FP32 output matches expected MLP math
Fair comparison	All 6 modes use the same input data & weights per round
Measurable speedup	CLBlast modes show higher throughput than custom kernels
Accuracy ordering	FP16 + Scale shows lower MSE than plain FP16
Stability	App runs continuously without crashing

FP32 vs. FP16 Precision

FP32 (Single Precision)

- 32 bits / 4 bytes per number
- $\sim$7 decimal digits of precision
- Range: $\pm 3.4 \times 10^{38}$
- The "gold standard" for inference

FP16 (Half Precision)

- 16 bits / 2 bytes per number
- $\sim$3 decimal digits of precision
- Range: $\pm 65\,504$
- **Half the memory $\Rightarrow$ double bandwidth**

Key Insight

GPU inference is often **memory-bandwidth limited**. Using FP16 moves twice as many values per second through the memory bus $\Rightarrow$ potential 1.5–2$\times$ speedup, but with measurable accuracy loss.

What is GEMM & CLBlast?

GEMM — General Matrix Multiply:

$$C = \alpha \cdot A \times B + \beta \cdot C$$

- Core operation in every neural network layer
- Two variants: **SGEMM** (FP32), **HGEMM** (FP16)

CLBlast — OpenCL BLAS library:

- Auto-tuned, tiled, vectorized GEMM kernels
- Selects optimal tile sizes per GPU at first call
- **5–20**× faster than naïve kernels at large sizes

Why CLBlast is Faster

1. **Tiled shared memory**
   Loads tiles into fast on-chip SRAM
2. **Coalesced access**
   Adjacent threads read adjacent memory
3. **Vectorized loads**
   Reads 4–8 floats per instruction
4. **Auto-tuning**
   Benchmarks many configs, picks the fastest

The Six Precision Modes

#	Mode	Kernel	Precision	Notes
1	FP32	Custom OpenCL	FP32	Gold standard reference
2	FP16	Custom OpenCL	FP16	$\sim$50% memory, some accuracy loss
3	FP16 + Scale	Custom OpenCL	FP16 weights + FP32 scales	Row-wise quantization
4	CLBlast FP32	BLAS SGEMM	FP32	Optimized matrix multiply
5	CLBlast FP16	BLAS HGEMM	FP16	Half-precision BLAS
6	CLBlast Mixed	BLAS SGEMM	FP16 stored $\rightarrow$ FP32 compute	Best of both worlds

All six modes receive the **same random input and deterministic weights** every round. FP32 runs first and stores its output as the **accuracy reference**.

Mode Comparison at a Glance

Mode	Storage	Compute	Bytes/Elem	Accuracy
FP32	FP32	FP32 (custom)	4	Perfect (reference)
FP16	FP16	FP16 (custom)	2	Some error
FP16 + Scale	FP16 + FP32 scales	FP32 accum.	$\sim$2.1	Better than FP16
CLBlast FP32	FP32	FP32 (SGEMM)	4	Perfect ($=$ FP32)
CLBlast FP16	FP16	FP16 (HGEMM)	2	Similar to FP16
CLBlast Mixed	FP16 + FP32 copies	FP32 (SGEMM)	$\sim$6	$\approx$ FP32

Technology Stack

Backend (Rust)

- `opencl3` — OpenCL bindings
- `libloading` — dynamic DLL loading
- `half` — FP16 $\leftrightarrow$ FP32 conversion
- `rand` — random input generation
- `serde` — JSON for Tauri IPC
- `tauri 2.x` — desktop app framework
- `plotters` — PNG chart generation
- `chrono` — timestamps for log folders

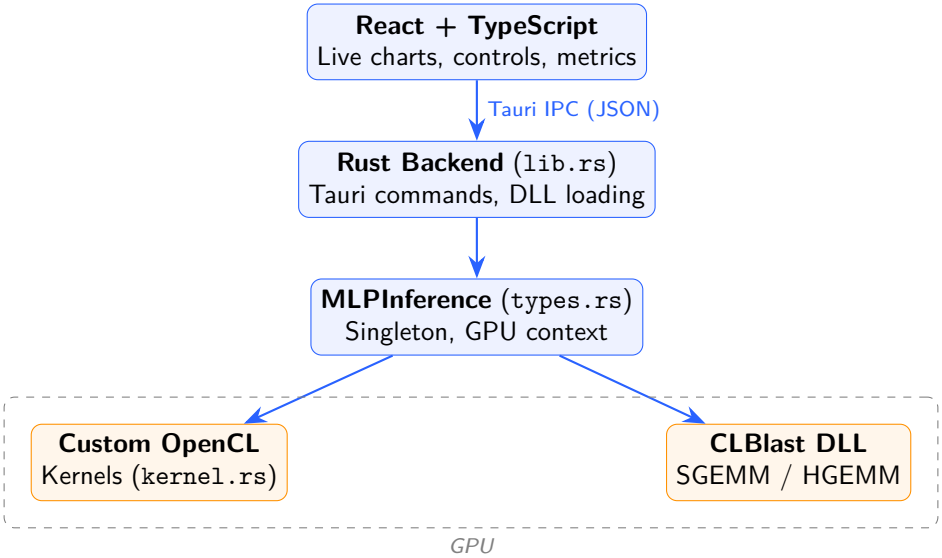
Frontend (TypeScript/React)

- React 19 + Vite 7
- Highcharts — live area charts
- TailwindCSS + DaisyUI
- Tauri IPC via `invoke()`

GPU

- OpenCL 1.2+ (any GPU vendor)
- CLBlast (embedded DLL)

System Architecture



Backend File Structure

File	Responsibility
`lib.rs`	Tauri commands, CLBlast DLL embedding & loading, device selection, FP16 helpers
`types.rs`	All structs (`MLPInference`, `RoundData`, metrics) + all 6 inference functions
`kernel.rs`	OpenCL C kernel source code (string constants, compiled at runtime)
`logger.rs`	CSV export + PNG chart generation (`plotters`) for automatic logging
`main.rs`	Minimal entry point — calls `lib::run()`

Key design decisions:

- **Singleton GPU context** — avoids repeated OpenCL init (∼200 ms each)

Two-Layer MLP Forward Pass

Network Architecture

$$\underbrace{\mathbf{X}}_{64\times N} \xrightarrow{\mathbf{W}_1^T+\mathbf{b}_1} \underbrace{\text{ReLU}(\mathbf{H})}_{64\times N} \xrightarrow{\mathbf{W}_2^T+\mathbf{b}_2} \underbrace{\mathbf{Y}}_{64\times N/2}$$

- `batch_size` $= 64$ (fixed)
- `input_size` $=$ `hidden_size` $=$ `matrix_size` (128 / 256 / 512 / 1024)
- `output_size` $=$ `matrix_size` $/ 2$

Layer 1: $\mathbf{H} = \text{ReLU}(\mathbf{X} \cdot \mathbf{W}_1^T + \mathbf{b}_1)$

Layer 2: $\mathbf{Y} = \mathbf{H} \cdot \mathbf{W}_2^T + \mathbf{b}_2$

Same random $\mathbf{X}$ and deterministic $\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2$ for all 6 modes each round.

Row-Wise Weight Scaling (Mode 3)

Problem: Naïve FP16 conversion loses precision for large weight values.

Solution: Normalize each row so all values fit in $[-1, 1]$ (FP16's sweet spot):

CPU Pre-processing

$$\text{scale}[h] = \max_i \left|\mathbf{W}[h][i]\right| \qquad \mathbf{W}_{\text{fp16}}[h] = \texttt{f32\_to\_f16}\left(\frac{\mathbf{W}[h]}{\text{scale}[h]}\right)$$

GPU Kernel — recovering original magnitude

$$\text{sum} \mathrel{+}= \texttt{(float)}\,\text{input}[i] \;\times\; \texttt{(float)}\,\mathbf{W}_{\text{fp16}}[h][i] \;\times\; \text{scale}[h]$$

- All FP16 weight values are in $[-1.0,\ 1.0]$ — maximum representational density
- Hidden buffer kept as FP32 to avoid a second round of quantization error
- Small overhead: one extra multiply per element + storing the scale vector

CLBlast GEMM Integration (Modes 4–6)

CLBlast SGEMM Call (per layer)

```
CLBlastSgemm(RowMajor, NoTrans, Trans, M, N, K, 1.0, A, B, 0.0, C)
```

- **Transpose on** B (weights) avoids costly memory rearrangement
- BLAS has no concept of bias or ReLU $\Rightarrow$ separate helper kernels:
  - `add_bias_relu` after Layer 1
  - `add_bias` after Layer 2

CLBlast Mixed mode additionally:

1. Uploads weights as FP16 (saves bandwidth)
2. Runs `convert_fp16_to_fp32` kernel **on GPU**
3. Calls SGEMM in full FP32 $\Rightarrow$ near-FP32 accuracy

Auto-tuning

First GEMM call per matrix size triggers CLBlast's auto-tuner (1–3 s).
Solved by `warmup_clblast()` before timing begins.

Performance Metrics

Metric	Unit	Formula / Method
Execution Time	ms	Wall-clock: `Instant::now()` → `queue.finish()`
Throughput	GFLOPS	$\frac{\text{batch} \times \left[H(2I+1) + O(2H+1)\right]}{t \times 10^9}$
Memory BW	GB/s	$\frac{\sum(\text{bytes read} + \text{bytes written})}{t \times 10^9}$
Memory Footprint	MB	$\frac{\text{total GPU buffer bytes}}{1024^2}$
Accuracy (MSE)	—	$\frac{1}{N}\sum_i \left(\text{output}_i - \text{fp32\_ref}_i\right)^2$

- FP16 buffers count **2 bytes/elem**; FP32 buffers count **4 bytes/elem**
- FP32 and CLBlast FP32 always report MSE = 0.0 (they *are* the reference)
- Timing excludes data generation and DLL loading

Evaluation Methodology

1. **Shared data:** One `RoundData` struct (random input + deterministic weights) passed by reference to all 6 modes
2. **Reference first:** FP32 always runs first; its output becomes the accuracy baseline
3. **Warmup:** `warmup_clblast()` fires untimed dummy GEMMs before measurement
4. **Continuous benchmarking:** Self-chaining async loop runs all 6 modes back-to-back, updating 4 live charts per round
5. **Adjustable workload:** Matrix size slider (128–1024) lets you study how problem size affects the tradeoffs

Automatic Logging & Plotting

Every **5 rounds**, the app automatically saves a full snapshot:

File	Contents
`metrics.csv`	All metrics from round 1 to current, all 6 modes
`execution_time.png`	Line chart comparing execution time
`throughput.png`	GFLOPS comparison
`bandwidth.png`	Memory bandwidth comparison
`accuracy_mse.png`	Accuracy MSE vs FP32 baseline

Log path: `parallel_log/{YYYY-MM-DD_HH-MM-SS}_{matrix_size}/`

- New session folder created automatically when matrix size changes
- Charts generated server-side in Rust via the **plotters** crate
- Colors match the live dashboard (green / blue / purple / yellow / cyan / pink)

Optimization Techniques

Technique	Where	Effect
Two-pass kernel design	Custom kernels	Each hidden value computed once, not O times
CLBlast auto-tune warmup	`warmup_clblast()`	Prevents tuning latency in benchmarks
Pre-compiled OpenCL programs	Bias/ReLU + convert	Saves $\sim$50 ms per round
Singleton GPU context	`MLP_INSTANCE`	Avoids repeated OpenCL init ($\sim$200 ms)
`spawn_blocking`	Tauri handlers	Keeps UI thread responsive
Single React dispatch	`chartReducer`	One re-render per round, not six
Embedded DLL	`include_bytes!`	Zero external dependencies

Challenges Encountered

CLBlast auto-tuner latency

- First GEMM call: 1–3 s
- Solution: untimed warmup calls

FP16 in Rust

- No native `f16` type
- Stored as `u16`, converted via `half` crate
- CLBlast `alpha`/`beta` also as `u16`

Thread safety of GPU state

- Raw OpenCL handles not `Send+Sync`
- `unsafe impl` guarded by mutex

DLL distribution

- External DLL is fragile
- Solution: `include_bytes!` bakes DLL into `.exe`, extracted to `%TEMP%` at runtime

Expected Performance Results

Approximate relative values at `matrix_size = 512`, `batch_size = 64`. *Actual numbers vary by GPU.*

Mode	Memory Footprint	Rel. Throughput	Accuracy MSE
FP32	100% (baseline)	1.0×	0.0 (reference)
FP16	∼50%	1.2–1.8×	Small, non-zero
FP16 + Scale	∼52%	1.0–1.5×	**Less than FP16**
CLBlast FP32	∼100%	3–10×	0.0
CLBlast FP16	∼50%	4–15×	Similar to FP16
CLBlast Mixed	∼150%	2–8×	≈ 0.0

CLBlast advantages grow significantly at larger matrix sizes (512+).
At `matrix_size = 128`, custom kernels may be competitive due to CLBlast overhead.

Analysis

Precision Format vs. Speed

FP16 kernels are faster due to **reduced memory bandwidth pressure**—each value is 2 B instead of 4 B. GPU cores run at similar speed for both.

BLAS vs. Custom Kernels

CLBlast is dramatically faster at large sizes: **tiled shared memory**, vectorized loads, and auto-tuned parameters vs. a simple inner loop.

Row-Wise Scaling Effect

FP16 + Scale consistently shows **lower MSE** than plain FP16. Most visible when weights have high dynamic range. Small speed cost from extra multiply + FP32 hidden buffer.

CLBlast Mixed Tradeoff

Uses the most GPU memory (both FP16 + FP32 buffers), but achieves **near-FP32 accuracy**

Conclusions

1. **CLBlast is significantly faster** than hand-written kernels at medium-to-large sizes — specialized BLAS optimizations cannot be matched by a simple loop.
2. **FP16 halves memory footprint** and produces modest speedups, but accuracy loss is **real and measurable**.
3. **Row-wise scaling meaningfully reduces FP16 error** by keeping values in FP16's most precise range. Accuracy benefit outweighs the small performance cost.
4. **CLBlast Mixed offers near-FP32 accuracy at FP16 memory cost** — the pragmatic production choice when accuracy cannot be compromised.
5. **CLBlast auto-tuning is a one-time cost** that must be accounted for in benchmarks.

Limitations

- **Windows-only** — CLBlast shipped as `.dll`; Linux/macOS would need `.so`/`.dylib`
- **Fixed batch size** (64) — configurable batch would reveal more tradeoffs
- **No hardware counters** — metrics are analytic, not measured from GPU perf counters
- **Auto-tuning per-session** — CLBlast cache lost on app close
- **Two layers only** — deeper networks may show different error accumulation behavior

Future Work

- INT8 quantization for further memory reduction
- Deeper networks (4–8 layers) to study error accumulation
- Configurable batch size in the UI
- Cross-platform support (Linux/macOS)
- Hardware counter profiling for true bandwidth measurement
- BF16 (Brain Float 16) as an additional comparison point
- Transformer attention block to study mixed-precision in modern architectures

Questions Addressed

Objectives achieved? Yes — all 6 modes implemented, correct, and benchmarked fairly on the same data per round. Tradeoffs clearly visible in live dashboard.

Biggest bottleneck? Custom kernels: **memory bandwidth**. CLBlast vs. custom: lack of **shared-memory tiling**.

vs. Expectations? CLBlast delivered **larger speedups** than expected. Row-wise scaling delivered **more accuracy improvement** than expected at almost no throughput cost.

Do differently? Use GPU hardware performance counters from the start; expose batch size as a UI parameter.

Thank You

Mixed-Precision Inference on GPUs

Built with **Rust** · **OpenCL** · **CLBlast** · **Tauri** · **React** · **TypeScript**

GPU Computing Project — Option 2: Mixed Precision with Row-Wise Scaling