AI-Based Hardware Performance Predictor — Technical Report

Executive summary

The system tells you, for a GPU kernel on a given GPU, which ceiling limits it and how long it will really take: on a GPU architecture it has never seen, the model trained on real measurements is off by a median 21% on kernel time, against 58% for plain Roofline (3% on new problem shapes, 13% on a new GPU).

Three predictors share one code base, and they differ in how much of their evidence is real:

Component	What it predicts	Training data	Status
Roofline analysis	Arithmetic intensity, ridge point, ideal time, attainable % of peak	None (equations)	Exact for the ceilings; optimistic about real kernels
Bottleneck classifier (`predict.py`)	Memory-bound, mixed or compute-bound, with probabilities	Simulated labels from a formula written for this project	Pipeline validated; results are not a hardware finding
Efficiency model (`predict_real.py`)	Measured time and efficiency relative to Roofline	452,120 measured kernels on 11 NVIDIA GPUs (PipeWeave)	Cross-validated on new shapes, GPUs and architectures
Nsight harness (`measure_gpu.py` , `detect_specs.py`)	Real bottleneck labels and achievable ceilings	Your own GPU	Written, never run on hardware

What the evidence supports:

- **Roofline is least reliable where the ceilings meet.** On real GEMMs the median efficiency is 0.61 to 0.65 within half a decade of the ridge point, against 0.97 deep in compute-bound territory. Attention falls much lower there.

- **Interpolation is easy, hardware extrapolation is not.** On a new architecture only 56% of predictions land within 25%, and the 80% intervals cover just 27% of true values, so they must not be trusted on unseen hardware.
- **The classifier's margin over Roofline (about 0.96 to 0.97 against 0.88 accuracy) comes from a simulator.** It shows the pipeline can learn the effects built into the simulator, not that real GPUs behave that way.

A Streamlit app (app.py, hosted on Streamlit Cloud) puts the Roofline analysis, the simulated-label classifier, the what-if table and the advisor behind a form, with no GPU needed. The real-data efficiency model is command-line only so far. The repository holds no measurements from the author's own GPU as of the files reviewed. Collecting them is the largest remaining step.

Problem statement and research question

The project asks whether a model can predict, from a workload description and a GPU's specifications, if a kernel is limited by compute or by memory bandwidth, and whether it can do so better than the analytic Roofline rule.

The Roofline model bounds a kernel's attainable throughput by two ceilings. With F the FLOPs performed and Q the bytes moved, arithmetic intensity I = F/Q is compared with the hardware's ridge point:

```
I = \frac{F}{Q}, \qquad I_{\text{ridge}} = \frac{P_{\text{peak}}}{B}, \qquad
P \le \min\left(P_{\text{peak}},\ I \cdot B\right)
```

The baseline classifier follows directly:

```
\hat{y}_{\text{Roofline}} = \begin{cases} \text{memory-bound}, & I <
I_{\text{ridge}} \\ \text{compute-bound}, & I \ge I_{\text{ridge}}
\end{cases}
```

Roofline is exact about the ceilings but optimistic about what a real kernel reaches. It assumes:

- every byte crosses DRAM once, so it ignores cache reuse and tile-reload traffic;
- the kernel runs at peak throughput, so it ignores wave quantisation, small matrix dimensions and tensor-core eligibility;
- launch and latency overhead are zero.

The research question is therefore: given workload features and hardware features, f(workload, hardware) maps to {memory-bound, mixed, compute-bound}. The questions are where Roofline fails, whether a learned model captures those failures, and how well each generalises to workloads and GPUs it has not seen.

Scope: single GPU kernels (GEMM, convolution, attention, normalisation, elementwise, reduction, gather, mixture-of-experts), NVIDIA GPUs, no whole-network analysis.

Hardware and workload model

Every prediction starts from two records, a hardware record of seven numbers and a workload record of FLOPs, bytes and structure, and the Roofline quantities follow from them analytically.

The hardware record holds FP32 peak, FP16 CUDA-core peak, FP16 tensor-core peak (0 if the GPU has none), memory bandwidth, L2 size, SM count and device memory. The effective compute ceiling for a kernel is chosen by precision: an FP16 kernel that can use tensor cores on a GPU that has them gets the tensor peak, any other FP16 kernel gets the FP16 vector peak, and FP32 kernels get the FP32 peak. The baseline is deliberately given this workload-aware peak, which is the fairest version of Roofline to beat.

GPU	Architecture	FP32 (TFLOPS)	FP16 vector (TFLOPS)	FP16 tensor (TFLOPS)	Bandwidth (GB/s)	L2 (MB)	SMs	Memory (GB)
P100	Pascal	9.3	18.7	0	732	4	56	16
T4	Turing	8.1	16.2	65	320	4	40	16
V100	Volta	15.7	31.4	125	900	6	80	32
A10	Ampere	31.2	31.2	125	600	6	72	24
RTX3090	Ampere	35.6	35.6	71	936	6	82	24
A100-40	Ampere	19.5	78	312	1555	40	108	40
A100-80	Ampere	19.5	78	312	2039	40	108	80
L4	Ada	30.3	30.3	121	300	48	58	24
RTX4090	Ada	82.6	82.6	165	1008	72	128	24
H100	Hopper	67	133.8	989	3350	50	132	80
GTX1650Ti	Turing (laptop)	3.0	6.0	0	192	1	16	4

These are approximate datasheet peaks entered by hand, and the GTX 1650 Ti row is an estimate for the laptop part. They should be checked against vendor datasheets, or replaced with ceilings measured by `detect_specs.py`, before being cited.

The workload record carries FLOPs, compulsory bytes, working-set size, tile-reload bytes, an irregular-access flag, contiguous row size, tile count, smallest dimension and tensor-core eligibility. Tiles use a 128 x 128 block, and reload bytes are the operand traffic repeated across tiles. The 12 kernel families come from nine operators, with b the bytes per element (4 for FP32, 2 for FP16):

Operator (families)	FLOPs	Compulsory bytes
GEMM (gemm, decode_gemv, bert_ffn)	$2 \cdot M \cdot N \cdot K \cdot \text{batch}$	$(M \cdot K + K \cdot N + M \cdot N) \cdot b \cdot \text{batch}$
Convolution (conv3x3, conv1x1)	$2 \cdot N \cdot K \cdot Ho^2 \cdot C \cdot R^2$	$(N \cdot C \cdot H^2 + K \cdot C \cdot R^2 + N \cdot K \cdot Ho^2) \cdot b$, Ho = H / stride
Depthwise convolution	$2 \cdot R^2 \cdot N \cdot C \cdot H^2$	$(2 \cdot N \cdot C \cdot H^2 + C \cdot R^2) \cdot b$
Attention scores QK^T	$2 \cdot B \cdot h \cdot L^2 \cdot d$	$(2 \cdot B \cdot h \cdot L \cdot d + B \cdot h \cdot L^2) \cdot b$
Softmax	$5 \cdot n$	$2 \cdot n \cdot b$
LayerNorm	$8 \cdot T \cdot H$	$(2 \cdot T \cdot H + 2 \cdot H) \cdot b$
Elementwise add, GELU, SiLU	1, 8, 6 per element	3, 2, 2 elements per element, times b
Reduction	n	$n \cdot b$
Embedding gather	rows·D	$2 \cdot \text{rows} \cdot D \cdot b + 8 \cdot \text{rows}$

Because the formulas count each byte once, they are lower bounds on traffic. Roofline treats them as exact, and the simulator and the learned models try to account for the difference.

Data: simulated labels, Nsight measurements and real latencies

The project uses three data paths that answer different questions, and only one of them is measured data at scale: a simulator that produces bottleneck labels, a measurement harness for your own GPU, and 452,120 real kernel latencies from other people's GPUs.

Simulated labels (feeds the classifier)

`rooflinelab/simulate.py` turns every (workload, GPU) pair into a compute time and a memory time and labels the pair by their ratio:

```
t_c = \frac{F}{P\,\eta_c}, \qquad t_m = \frac{Q_{\text{DRAM}}}{B\,\eta_m} + 1.5\ \mu\text{s}, \qquad r = \frac{t_c}{t_m}
```

The label is compute-bound when r > 1. In the three-class version, r within a factor of 1.5 of 1 is labelled mixed. The effects Roofline ignores enter through three terms:

- **DRAM traffic:** Q_DRAM starts from the compulsory bytes, adds 3% of the tile-reload traffic that misses L2, and subtracts warm-cache reuse of 0.8 × (L2 coverage)², floored at 2% of the compulsory bytes.
- **Compute efficiency:** η_c is a per-kernel base efficiency times wave efficiency, tiles / (SMs × ceil(tiles / SMs)), times a small-dimension penalty, times a per-GPU hidden factor between 0.72 and 0.92.
- **Memory efficiency:** η_m is 0.9 times a per-GPU hidden factor between 0.78 and 0.93, and gathers lose bandwidth when rows are shorter than a 128-byte burst.

Total time adds 4 µs of launch overhead and 6% lognormal noise, and overlap between the two sides is 0.8 (0.5 for irregular access). Rows where launch overhead exceeds half the runtime are dropped. The grid gives 1,158 workloads on 11 GPUs, 12,738 rows, of which 10,796 remain. Every constant was chosen for plausibility and none was calibrated against a real GPU.

Nsight measurement harness (feeds nothing yet)

`measure_gpu.py` runs each workload under Nsight Compute and reads the speed-of-light counters, so the label reflects what the hardware did. With sm__throughput and gpu__dram_throughput as percentages of peak, r = SM % / DRAM %, and the same label bands apply. Counters are needed because a label computed from runtime and datasheet peaks equals AI / ridge exactly, which is the Roofline rule again. Workloads whose working set exceeds 30% of device memory are skipped, which matters on a 4 GB laptop GPU. `detect_specs.py` measures the FP32 and FP16 GEMM throughput and copy bandwidth the GPU really reaches and writes a hardware JSON. Neither script has been run on a GPU, and Nsight metric names may need adjusting for your version.

Real kernel latencies (feeds the efficiency model)

The PipeWeave dataset (paper, repository) holds kernel latencies measured with the PyTorch Profiler on 11 NVIDIA GPUs: A100-80, A40, H100, H20, H200, H800, L20, L40, RTX 6000 Ada, RTX A6000 and RTX PRO 6000. The repository is Apache-2.0, but the dataset's own licence is not stated, so cite the paper and check with the authors before redistributing it. The training set uses 452,120 rows:

Operator	Rows	GPUs	Formulas and assumptions
GEMM (BF16)	118,800	11	FLOPs 2MNK; bytes 2(MK + KN + MN); BF16 tensor peak
GEMM (FP8)	25,228	4 (H100, H20, H200, H800)	Bytes MK + KN + 2MN (BF16 output assumed); peak is twice the BF16 tensor peak
Attention (FA2/FA3)	104,958	11	FLOPs Σ 4·q·kv·heads·head_dim, causal masking not discounted; bytes 2 × (Q + O + K + V) in BF16
RMSNorm	65,036	11	Bytes from one read and one write; FLOPs 4·seq·dim assumed; FP32 CUDA-core peak
SiLU×Mul	104,834	11	Reads seq × 2dim, writes seq × dim; FLOPs 6·seq·dim assumed; FP32 peak
Fused MoE, first GEMM	33,264	11	Expected distinct experts E(1 − (1 − 1/E)^(M·topk)) are read; the least certain of the six

The dataset has runtimes only, so the learning target is how far below Roofline a kernel runs, y = log10(measured time / ideal time). Only inputs a user can know in advance are features: shape, FLOPs, bytes, peak, bandwidth and SM count. The dataset's own cycle and operation-count columns and the cuBLAS tile choices are excluded because they are outputs. One data correction is applied: the RTX 6000 Ada's listed tensor peak looks doubled, since it matches the L40 on identical GEMMs, so it is halved. That is an inference from the data, not vendor-confirmed. `gemm_train.csv` (about 131 MB, Git LFS) was not available, and the communication-kernel files (all-reduce, send/recv) are not used.

Methods

Both learned models are judged against the same yardstick: the analytic Roofline rule, which uses no data, so every claim is a margin over a rule that already encodes the hardware.

Bottleneck classifier

The baseline predicts memory-bound when the ratio of arithmetic intensity to ridge point is below 1 and compute-bound otherwise. In the three-class task the mixed band is a factor of 1.5 either side of 1.

The classifier sees 21 features, 18 of them base features and 3 Roofline-derived ones:

- **Workload:** log FLOPs, log bytes, log working set, irregular-access flag, log row size, log tile count, log smallest dimension, tensor-core eligibility, and an FP16 flag.
- **Hardware:** log effective peak, log bandwidth, log L2 size, log SM count, log memory capacity, tensor-core presence and use, log working set over L2, and log tiles per SM.
- **Roofline-derived:** log arithmetic intensity, log ridge point, and their log ratio.

There is no workload identifier among them, so an unseen-workload test measures real generalisation. Two feature sets are compared: `full` with the three Roofline-derived columns and `raw` without them. A logistic regression on `raw` can still express the Roofline rule exactly, since the log ratio is linear in the other logs.

Five methods form a ladder of complexity: Roofline; logistic regression (standardised inputs, C = 1); random forest (200 trees, minimum leaf size 2); histogram gradient boosting (250 iterations, learning rate 0.06); and a small neural network (layers of 64 and 32, early stopping). The deployed classifier is the gradient-boosting model with three classes. Run with `--real`, `train.py` also blends in measured rows from the Nsight harness at five times the weight, and first reports how Roofline and the simulation-trained model score on those real rows.

The experiments hold out different things:

Experiment	What is held out	Folds	What it tests
Random split (leaky)	20% of rows	1	Contrast only: the same shape on other GPUs stays in training
A, unseen workload	One kernel family	12	Generalisation to a new type of kernel

Experiment	What is held out	Folds	What it tests
B, unseen hardware	One GPU	10	Generalisation to a new GPU
C, unseen pair	20% of (family, GPU) pairs	5 repeats	Generalisation to a new combination

Metrics are accuracy, balanced accuracy, macro-F1, and accuracy near the ridge, meaning within a factor of 3 of it, where Roofline should struggle most.

Efficiency model

The real data has runtimes but no bottleneck labels, so this model learns how far below Roofline a kernel runs. For each kernel the ideal time is the larger of FLOPs / peak and bytes / bandwidth, and the target is y = log10(measured time / ideal time). The predicted time is the ideal time times 10^ŷ.

The 30 features are the operator identity; log FLOPs, bytes, arithmetic intensity, peak, bandwidth, ridge point, AI-to-ridge ratio and ideal time; log SM count and peak per SM; and the shape (M, N, K, sequence length, hidden size). Attention adds batch size, summed and maximum query and key lengths, head counts, head dimension, the share of single-token queries and the attention type, and MoE adds expert count, top-k, hidden size and expected distinct experts.

Three histogram gradient-boosting regressors with quantile loss at 0.1, 0.5 and 0.9 give a median prediction and an 80% interval (up to 300 iterations, learning rate 0.08, 63 leaves, early stopping, quantiles forced to be monotone). Three baselines are compared: plain Roofline (efficiency 1), a per-operator constant efficiency, and a per-operator power law of efficiency against ideal time.

Split	What is held out	Folds	What it tests
Random (leaky)	20% of rows	1	Contrast only
Unseen shape	Problem shapes (grouped), on every GPU	5	Interpolation on known hardware
Unseen GPU	One GPU	11	New GPU, with same-die siblings still in training

Split	What is held out	Folds	What it tests
Unseen architecture	A whole family: Hopper, Ada, Ampere or Blackwell	4	New hardware generation, the fair hardware test

Metrics are the median relative time error $|\hat{t} - t| / t$, the share of predictions within 25%, the mean absolute error of log10 time ratio, the median error for kernels of at least 20 µs, and how often the 80% interval contains the truth. Each cross-validation fold trains on at most 150,000 rows for speed, and the shipped models are fitted on all rows.

Experiments and results

On real measurements the learned model beats Roofline by a wide margin when the hardware family is known and by a narrow one when it is not.

Real data: how far below Roofline kernels run

All results are cross-validated, and every fold trains only on its training rows. The error is the median of |predicted time − measured time| / measured time, over 452,120 kernels.

Plain Roofline is off by a median 58% because real kernels do not run at the ceiling. The learned model's error depends on what it has seen:

- **New shapes on known GPUs:** 3.3% median error, with 96% of predictions within 25%. The 80% intervals contain 79% of true values, so they are well calibrated here.
- **A new GPU:** 12.8% median error and 76% within 25%. The intervals cover only 45%. This setting flatters the model, because same-die siblings (H100, H200 and H800; L20, L40 and RTX 6000 Ada; A40 and RTX A6000) stay in training.
- **A new architecture:** 21.3% median error and 56% within 25%, with intervals covering 27%. Only four architectures exist in the data, so this is the fair hardware test, and it is weak.

The gain over a per-operator constant efficiency is uneven. On the unseen-architecture test:

Operator	Plain Roofline	Constant efficiency per operator	Learned model
GEMM (BF16)	25%	17%	19%
GEMM (FP8)	65%	53%	25%

Operator	Plain Roofline	Constant efficiency per operator	Learned model
Attention	59%	71%	33%
RMSNorm	61%	39%	24%
SiLU×Mul	79%	26%	20%
Fused MoE	61%	12%	10%
All operators	58%	27%	21%

For plain BF16 GEMM the learned model does not beat a constant efficiency, and for MoE it barely does. Its gains come mostly from small, latency-bound kernels and from attention and FP8, where a single efficiency number fits poorly.

Roofline is least reliable where compute and memory ceilings meet. Efficiency is lowest within about half a decade of the ridge point for GEMM and attention. Attention shows efficiency above 1 on the compute side because causal masking is not discounted in its FLOP count, so its absolute efficiency should be read with care.

Simulated data: classifier accuracy

These experiments ran on the original 10-GPU grid (11,580 rows, 9,716 after the launch-overhead filter); the GTX 1650 Ti was added afterwards. Labels come from the simulator, so the numbers show that the pipeline learns the effects built into it and say nothing about real devices.

The learned models beat Roofline in every setting, by 7 to 11 points for the best model of each experiment. Near the ridge the gap is larger: Roofline reaches 0.65 to 0.71 accuracy there, against 0.85 to 0.96 for the best model. Further observations:

- Unseen workload (A) is the hardest generalisation. With raw features the random forest trails logistic regression (0.915 against 0.944), so tree models extrapolate to new kernel families worse.
- The leaky random split inflates the tree models by only one to two points, because the simulator is smooth. Do not expect so small a gap on real measurements.
- Mixed is only about 9% of rows and is where every model is weakest, with balanced accuracy between 0.66 and 0.88 in the three-class task.
- Permutation importance ranks tensor-core eligibility, FLOPs, smallest dimension and peak throughput highest. That reflects how the simulator was built, not hardware behaviour.

The predictor, command line and app

The same Roofline analysis, classifier verdict, what-if table and advisor are reachable from the command line and from a Streamlit app, while the real-data efficiency model has a command of its own.

Front end	Model behind it	Hardware it accepts	Output
`predict.py`	Roofline + simulated-label classifier + advisor	Any GPU, including ones without tensor cores	Verdict, probabilities, diagnostics, what-if table, ranked optimisations
`predict_real.py`	Real-data efficiency model	Tensor-core GPUs, BF16 and FP8 kernels; refuses GEMM, attention and MoE on GPUs without tensor cores	Predicted time with an 80% interval, efficiency, per-operator validation
`app.py` (Streamlit)	Same as `predict.py`	Preset or custom form	The `predict.py` analysis in a browser, no GPU required

Inputs

Hardware comes from a preset name (11 GPUs), a JSON file written by `detect_specs.py`, or raw numbers: FP32 TFLOPS, bandwidth, L2 size, SM count and memory are required, and FP16 vector and tensor TFLOPS are optional. The workload is one of the 11 operators with its size parameters, or a custom kernel given by FLOPs and bytes, with optional working set, irregular-access and tensor-eligibility flags. `predict_real.py` takes tensor TFLOPS, FP32 TFLOPS, bandwidth and SM count, and the operators GEMM, FP8 GEMM, attention (given as per-request query and key lengths), RMSNorm, SiLU×Mul and the first MoE GEMM.

Anatomy of a `predict.py` report

1. Hardware summary and the workload's FLOPs, compulsory bytes and working set.

2. Roofline analysis: arithmetic intensity, ridge point, ideal compute and memory time, attainable percent of peak, and the analytic verdict.
3. ML verdict with the three class probabilities and the provenance of the training labels.
4. Structural diagnostics from heuristic formulas: L2 coverage, estimated DRAM traffic against compulsory bytes, wave efficiency and the small-dimension penalty.
5. What-if table: doubled bandwidth, doubled compute, 4× L2, FP16 in place of FP32, 8× larger batch for GEMM and convolution, and fusion for streaming operators.
6. Optimisations ranked by Roofline upper-bound gain.
7. Notes: ML disagreement with Roofline and why, near-ridge and launch-dominated warnings, and out-of-range hardware.

Worked examples

A decode-style GEMV (FP16, M = 1, N = K = 8192) on a GTX 1650 Ti has arithmetic intensity 1 FLOP/B against a ridge point of 31.2, so AI / ridge is 0.032, the ideal time is 699 µs set by memory, and the attainable throughput is 3.2% of peak. Both the Roofline and the classifier say memory-bound. The advisor's top levers are a larger batch (Roofline crossover to compute-bound near M = 32, at most 8× per item at M = 8), INT4 weights (up to 4×) and INT8 weights (up to 2×).

The real-data model gives time predictions for the same kind of kernel on GPUs it was trained on:

Case	Roofline side	Ideal time	Predicted time (80% interval)	Efficiency
BF16 GEMM 4096^3 on H100	Compute-bound	139.0 µs	178.4 µs (173.6 to 184.1)	78%
BF16 GEMV, M = 1, N = K = 8192 on A100-80	Memory-bound	65.8 µs	94.3 µs (84.5 to 105.6)	70%

Both lie within the model's strongest regime, a known GPU and a typical shape. The tool prints this caveat itself, because its intervals cover far less than 80% on unseen hardware.

Streamlit app

The app (`app.py`) is a form-driven front end over `rooflinelab.advisor.analyze`. Step 1 describes the architecture, either as a preset (default A100-40) or as custom values for FP32, FP16 vector and tensor peaks, bandwidth, L2, SMs and memory. Step 2 defines the kernel from 12 options, the 11 operators plus a custom kernel, in FP32 or FP16. Pressing Analyze shows four headline metrics (Roofline verdict, ML verdict, arithmetic intensity and AI / ridge) over four tabs:

- **Roofline analysis:** compute, memory and ideal times, the two ceilings, a log-log Roofline plot labelled as an upper bound rather than measured throughput, and the raw inputs.
- **ML explanation:** a probability bar chart and the structural diagnostics, with a note that the probabilities are not calibrated.
- **What-if scenarios:** the same scenario table with ideal and per-item speed-up bounds.
- **Optimization advisor:** ranked suggestions in cards, each with its idealised bound.

A persistent warning states that the classifier is trained on simulated labels. The app trains that classifier in memory on first launch with a fixed seed, so it needs no model file, no dataset and no GPU. A dev container configures GitHub Codespaces to serve it on port 8501, and the repository README links the hosted instance. The app does not call the real-data efficiency model, so measured-data predictions are command-line only for now.

The optimisation advisor

The advisor is a rule engine, not a learned model. It reads the Roofline result, the classifier verdict (or the Roofline verdict when no classifier is loaded), the structural diagnostics and the what-if table, and emits a list of suggestions that it sorts by idealised speed-up bound. Every bound is an upper limit computed from the Roofline, not a measured gain.

What-if scenarios

Each scenario changes one thing and re-runs the Roofline and the classifier against the base ideal time:

Scenario	What changes	When it appears
2× memory bandwidth	Bandwidth doubled	Always
2× compute peak	FP32, FP16 vector and FP16 tensor peaks doubled	Always
4× L2 cache	L2 size quadrupled	Always
FP16 instead of FP32	Same operator, FP16	FP32 kernels with a known operator
8× larger M or batch	GEMM M, or convolution N, times 8; reported as a per-item bound because the work differs	GEMM and convolution
Fuse with neighbour op	DRAM bytes and working set halved	Softmax, LayerNorm, elementwise, depthwise convolution, custom

Rules by verdict

Verdict	Rules that can fire
Any, if launch-dominated	Batch small ops, fuse neighbours or use CUDA Graphs, when the ideal runtime is comparable to the 4 µs launch cost
Memory-bound	FP16/BF16 for FP32 kernels; fusion for streaming ops; larger batch with the Roofline crossover M for GEMM and convolution; gather coalescing for rows under 128 B, or index sorting and hot-row caching for rows of 128 B or more; INT8 and INT4 weight-only quantisation when weights exceed 80% of traffic at $M \leq 64$; L2-aware tiling when the working set exceeds L2; keep-in-L2 when it fits; FlashAttention-style fusion for softmax and attention scores; vectorised 128-bit loads

Verdict	Rules that can fire
Compute-bound	FP16/BF16 to use tensor cores or the faster FP16 vector rate; tensor-core eligibility when FP16 runs without them; wave-quantisation fix when the last wave is under 85% busy; small-dimension padding or grouped GEMM when the dimension penalty is below 0.9; Winograd or FFT for 3×3 convolution; algorithmic FLOP reduction
Mixed	Optimise both sides, with the bound capped at the ratio of the two ideal times; FP16/BF16

The rules are heuristics written from standard GPU practice. They encode which levers move which ceiling, and the numbers they quote come from the formulas above, so a suggestion's bound is only as good as the Roofline behind it.

Validation, limitations and threats to validity

What the project measures well, and what it does not, is the central point of this section.

What is validated

- **Real-data model:** evaluated on 452,120 measured kernels with four split types, including held-out GPUs and held-out architectures, against two baselines and the plain Roofline. Held-out shapes: 3.3% median time error. Held-out GPUs: 12.8%. Held-out architectures: 21.3%, against 27.3% for a constant-efficiency baseline.
- **Simulated classifier:** 93.6% to 97.1% two-class accuracy across unseen-workload, unseen-hardware and unseen-pair splits, against 86.4% to 89.0% for the plain Roofline.
- **Code:** two smoke tests, one checking Roofline consistency and one running the whole analysis end to end. Both were run by hand here and passed; `pytest` itself was not installed in the sandbox, so a full test-runner pass is still to be shown.

Limitations

1. **Simulated labels are not hardware findings.** The classifier learns the rules of the simulator the project wrote. Its accuracy shows that it recovers those rules, not that GPUs behave that way. The simulator's efficiencies and noise are assumed.

2. **The Nsight Compute harness has not been run on hardware.** `measure_gpu.py` is written and bounded for a 4 GB laptop GPU, but no labels from it exist.
3. **The app does not use the measured-data model.** It shows the simulated classifier only, and says so on screen.
4. **Measured data covers 11 datacentre and workstation GPUs.** Held-out-architecture error of 21% is the honest guide for a GPU outside that set. There is no consumer-GPU data, so a GTX 1650 Ti has no tensor-core predictions and gets the simulated path only.
5. **Intervals under-cover on new hardware.** The 80% interval covers 78.7% of held-out shapes, 44.9% of held-out GPUs and 26.7% of held-out architectures.
6. **Attention efficiency above 1** appears because causal attention is not discounted in the FLOP count, so the Roofline's ideal time is an overestimate there.
7. **Hardware table values were entered from memory** and are approximate; the RTX 6000 Ada tensor peak correction is an inference from the data.
8. **Dataset licence is unstated.** The PipeWeave repository is Apache-2.0, but the dataset carries no licence of its own, so publishing derived numbers is safe while redistributing the data is not assumed to be.

Threats to validity

- Random splits leak: the same shape on the same GPU appears in training and test, so the random-split score is shown for contrast only.
- Experiments on simulated data used the original 10-GPU grid, while the deployed classifier trains on 11 GPUs, so the reported simulated accuracy is for a slightly different set from the one the app trains on.
- Accuracy on the simulated near-ridge band, where the classes are hardest, is lower for every model, and the ML advantage there is largest: 85% to 96% against 65% to 71% for Roofline.

Reproducibility and deployment

Repository layout

Path	Contents
`rooflinelab/`	Library: `hw_specs`, `workloads`, `features`, `simulate`, `baseline`, `experiments`, `advisor`, `realdata`, `plots`
`app.py`	Streamlit front end
`predict.py`, `predict_real.py`	Command-line predictors for the simulated and measured-data models
`train.py`, `train_real.py`	Training scripts, writing `models/model.joblib` and `results/real/`
`run_all.py`	Simulated experiment suite
`fetch_real_data.py`, `make_real_figs.py`	Dataset download and figure generation
`measure_gpu.py`, `detect_specs.py`	Nsight Compute label harness and hardware probe
`tests/test_demo_core.py`	Smoke tests
`.devcontainer/`	Codespaces configuration, Python 3.11, port 8501

Running it

```
pip install -r requirements.txt
streamlit run app.py
python -m pytest -q
```

The other scripts are run as `python <script>.py`. `train.py` builds the simulated classifier; `train.py --real` blends in measured rows with a weight of 5. `fetch_real_data.py` downloads the PipeWeave CSVs, and `train_real.py` then reproduces the table of results. The `models/`, `results/` and `data/` folders are gitignored, so they are regenerated rather than committed.

Environment

The app needs only NumPy, pandas, scikit-learn, Matplotlib and Streamlit 1.40 or newer, and trains its classifier in memory with seed 0 in a few seconds. It uses no GPU. The hosted instance is on Streamlit Community Cloud; the dev container gives the same setup in Codespaces.

Snapshot

Results in this report come from the dataset as downloaded on 31 October 2026 and from the repository as cloned the same day.

Future work

1. **Run the label harness on real hardware.** Collecting Nsight Compute labels on the GTX 1650 Ti, even for a few hundred kernels, would give the first labels that are not simulated, and would test the simulator's assumptions.
2. **Bring the measured-data model into the app.** Predicted time and an interval would sit beside the Roofline verdict for supported GPUs and operators.
3. **Cover consumer GPUs and more operators.** Convolution, softmax and gather have no measured data yet; consumer cards would test the architecture-transfer error.
4. **Fix causal-attention FLOPs** so attention efficiency lies in the usual 0 to 1 range.
5. **Calibrate intervals** with conformal methods, aimed at the poor coverage on new hardware.
6. **Record the dataset licence position** by asking the PipeWeave authors, before any data is redistributed.