HARDWARE-AWARE FEATURES FOR CUTLASS KERNEL SELECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

GPU libraries such as CUTLASS expose tens of thousands of semantically equivalent kernels for a single operation, making exhaustive autotuning expensive and execution-free selection difficult. Existing analytical selectors require hand-designed performance rules, while learned selectors operate on raw configuration parameters and must infer hardware consequences from data. We introduce a hardware-aware representation for CUTLASS kernel selection that augments candidate configurations with statically computable estimates of induced hardware behavior. We construct a dataset of 4.9 million CUTLASS kernels and train gradient-boosted and neural learning-to-rank models to rank candidates within each problem. On held-out exhaustive evaluation problems, hardware-aware representations reduce selection regret by up to 40% relative to structural baselines and 64.2% relative to NVIDIA's matrix-multiply heuristics. We further evaluate data-efficient cross-precision and epilogue-fusion transfer within CUTLASS GEMM, showing that explicitly representing candidate-induced hardware behavior provides a useful inductive bias for learned kernel selection.

1 INTRODUCTION

Modern GPU libraries expose increasingly large spaces of implementations for a single operation. CUTLASS (Thakkar et al. (2023)) is a particularly expressive instance of this problem – a CUTLASS GEMM may be implemented in over sixty thousand ways using different tile shapes, instruction shapes, pipeline depths, schedules, cluster configurations, tile schedulers, and epilogue implementations, interacting non-trivially with the problem dimensions, layouts, data types, accumulation types, and target architecture (Thakkar et al. (2025)). The best configuration is determined by the interaction of these effects rather than by any one parameter in isolation. This flexibility is valuable to kernel developers, but it also creates a very large configuration space. Selecting an efficient implementation is therefore a central requirement for achieving near-peak performance on contemporary accelerators (Bikshandi & Shah (2023)). Empirical autotuning, by compiling and benchmarking candidate kernels, is expensive (Barclay et al. (2025); Bai et al. (2025); Swann et al. (2025)). Compilation and measurement must be repeated for each target GPU, datatype, epilogue, and problem shape. Empirical selection becomes impossible when a library requires several different GEMM shapes or must make selections dynamically. This motivates selectors that can rank candidates without executing them (Barclay et al. (2025)).

Existing execution-free selectors follow two main approaches. Analytical methods encode manually designed performance rules, such as preferences for tile sizes, occupancy limits, or wave quantization based on the problem instance (Swann et al. (2025); Zhang et al. (2026a)). These methods are fast but brittle, require substantial architecture-specific knowledge for development, and must be updated as hardware and kernel libraries evolve. Learned autotuners (Ryu & Sung (2021); Zhai et al. (2023)) reduce the need for manual rules, but typically represent candidates using raw problem dimensions and template parameters, leaving the model to then infer how those parameters affect hardware behaviour. Thus, raw parameters are a poor description of these effects.

We propose hardware-aware representations for kernel selection. Each candidate is represented using its structural configuration together with inexpensive estimates of its induced hardware behaviour. We construct a 4.9M-kernel training corpus, train ranking models, score the entire valid catalogue at inference, and obtain 6.2% mean selection regret, compared with 17% for nvMMH

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

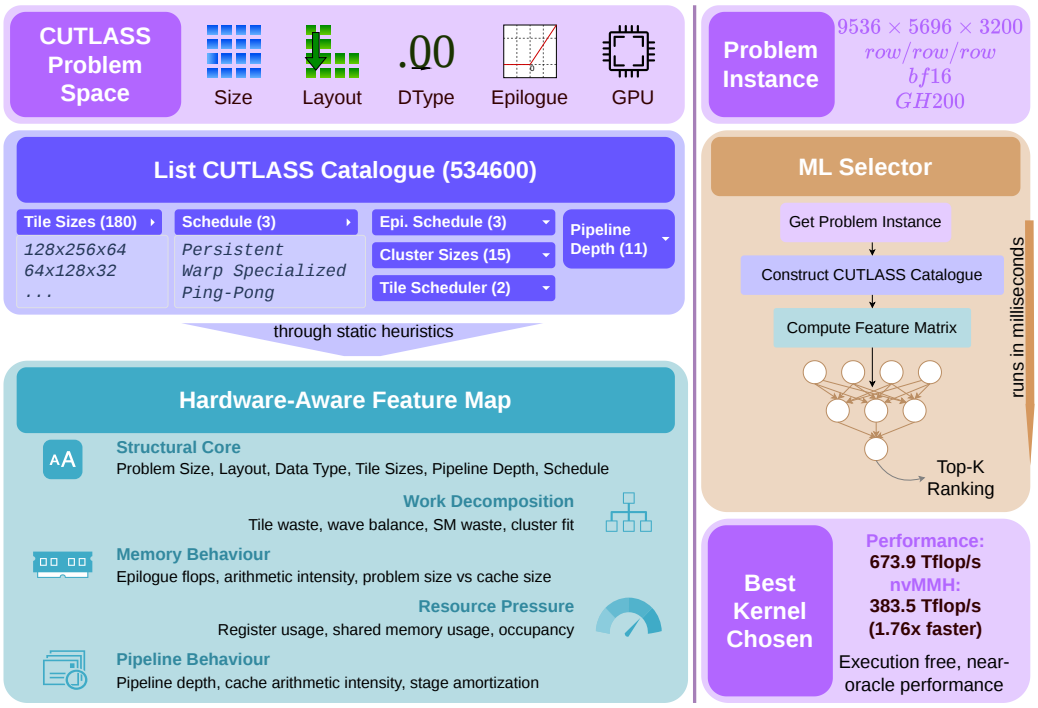


Figure 1: Overview of our execution-free CUTLASS kernel selector, which maps each problem and candidate catalogue to hardware-aware features and ranks kernels without benchmarking.

(nvMatmulHeuristics, NVIDIA's matrix multiply heuristics library), and 71.5% for random selection – a 64.2% and 91.3% relative reduction in selection regret, respectively.

Our approach combines the strengths of analytical selectors and measurement-driven learning: hardware behavior is exposed explicitly through static, mechanistic features, while the model learns how these effects interact from measured data rather than relying on a hand-written performance model. This gives the selector the best of both worlds: strong architectural inductive bias without brittle decision rules, and data-driven adaptability without forcing the model to rediscover hardware behavior from raw parameters. As a result, the same representation supports accurate selection, improved data efficiency, and transfer across related kernel spaces, establishing hardware-aware feature design as a practical foundation for learned performance modeling on modern accelerators.

2 Background

For a fixed GEMM problem on NVIDIA GPUs, CUTLASS (Thakkar et al. (2023)) provides a large catalogue of semantically equivalent kernels. A candidate is defined by several interacting choices – thread-block tile (T_M, T_N, T_K), pipeline depth, mainloop and epilogue schedules, thread-block cluster shape (Luo et al. (2024)), and tile scheduler – that determine how the same matrix multiplication is mapped onto the hardware. For one shape, our BF16 enumerator exposes $534{,}600$ combinations per layout before validity filtering ($2{,}138{,}400$ kernels throughout). Static CUTLASS legality and shared-memory checks retain around 11.36%, leaving a catalogue of around $61{,}000$ valid kernels per layout ($243{,}000$ throughout), based on which the selector must choose candidates.

On Hopper architecture (Luo et al. (2024)), interactions are especially pronounced and interleaved because high-performance CUTLASS kernels combine asynchronous memory transfers with tensor-core operations in warp-specialized, multi-stage pipelines. On a GH200, measured CUTLASS throughputs on BF16 GEMMs in our sweep span 0.01–763 Tflop/s depending on problem size and configuration. For one given GEMM shape and data layout, the empirical best and worst valid kernels can differ by a factor of $1{,}070\times$ in throughput (median across 68 evaluated groups; 3.9 million measured pairs), a uniform random pick achieves only 28.5% of the in-group oracle on average, and

choosing a static best configuration only 71%. Less than 2.4% of configurations get performance within 5% of the best kernel on average. Mis-selection is therefore not a rounding error; it can waste most of the hardware budget on every GEMM in a compiled program. Exhaustively compiling and benchmarking the full catalogue is impractical at scale: our evaluation set contains only 17 shapes, yet requires a benchmarking sweep over the 3.9 million distinct (kernel, shape) GPU measurements (68 unique groups, up to $61{,}000$ candidates per group). Compiling the heavily templated configuration catalogue once alone costs on the order of hours of CPU time even before any GPU benchmarking begins.

This motivates the selection problem we study. Hand-written analytical selectors must explicitly encode the interactions above and still miss schedule–tile synergies that only appear in timed data. We instead ask whether these interactions can be exposed through inexpensive, statically computable proxies of hardware behavior, allowing a learned ranker to score the complete valid catalogue at compile time without executing kernels. Architectural innovations like Stream-K (Osama et al. (2023)) and FlashAttention-3 (Shah et al. (2024)) demonstrate that GPU efficiency increasingly relies on smart work partitioning and pipeline coordination rather than just raw computation. We model these underlying mechanisms as statically computed features to select optimal configurations from the vast spaces exposed by CUTLASS. We provide additional details on Hopper execution mechanisms, the CUTLASS configuration space, and validity constraints in Appendix A.

3 From CUTLASS Candidates to Hardware-Aware Features

For each candidate configuration, we compute a feature map entirely from the problem description, configuration parameters, and hardware constants. This constraint is essential for an execution-free selector. To decide which execution effects are worth representing, we first profile kernels. We collect 408 Nsight Compute profiles of BF16 CUTLASS GEMM kernels on GH200 across 17 shapes. For 24 different metrics x and measured benchmark throughput T, we compute Spearman correlation (which measures monotonic agreement between the ordering induced by a metric and the ordering by measured throughput) separately within every fixed (M, N, K, layout) group g:

$$\rho_g(x) = \text{Spearman}\big(\{x(c)\}_{c\in C_g}, \{T(c)\}_{c\in C_g}\big)\,.$$

Because the problem and layout are fixed within each group, ρ_g measures whether a quantity helps rank candidate configurations rather than whether it merely tracks problem size – a global correlation can be high simply because larger GEMMs both sustain more absolute hardware activity and achieve higher throughput, even if that metric does not distinguish good from bad kernels for the same problem (examples include SM activity, barrier stalls, or occupancy).

Table 1: Spearman correlation with throughput, for 5 representative NCU counters out of 24.

	Tensor-core activity	DRAM throughput	SM activity	Barrier stalls	Achieved occupancy
Global ρ	$+0.992$	$+0.817$	$+0.910$	$+0.894$	$+0.388$
Mean within ρ_g	$\mathbf{+0.392}$	$\mathbf{+0.408}$	-0.001	$+0.156$	$+0.140$

We summarize the correlations of 5 representative counters out of 24 in Table 1. Two effects remain particularly informative once the problem is held fixed: DRAM throughput and tensor-core activity. Both remain informative for ranking candidates within a fixed problem, but neither can be known without executing the kernel. This motivates estimating them with static proxies for the mechanisms that produce them. Memory behavior is represented through arithmetic intensity, operand re-streaming, cache-relative working sets, and explicit traffic estimates; compute-pipeline behavior is represented through estimated data-feed versus compute balance, pipeline fill and amortization, and tile and instruction geometry.

The same analysis also shows why measured counters should not be treated as direct optimization targets. For example, SM activity and barrier stalls are almost perfectly correlated with throughput globally but carry essentially no within-problem ranking signal. This is because few warps are able to saturate compute resources on modern hardware, while more barrier stalls might simply be indicative of a deeper, more efficient pipeline. Likewise, raw configuration parameters are often

insufficient in isolation: the effect of pipeline depth depends on tile storage and resource pressure, and the effect of a tile dimension depends on the amount and geometry of work it induces.

Feature-selection criteria. We retain features using three complementary criteria:

1. **Static availability.** Statically computable from the problem, candidate, and hardware. When its effect is context-dependent, we include the corresponding contextual variables explicitly (and the context variables must be statically available).
2. **Empirical relevance.** The quantity either distinguishes candidate kernels within a fixed problem or provides context needed to interpret candidate-dependent quantities.
3. **Architectural grounding.** The quantity has a concrete mechanistic relationship to hardware behaviour, e.g. shared-memory usage affecting residency, edge-tile waste reducing useful work.

Table 2 summarizes our chosen feature set. The structural core identifies the problem and candidate. The remaining groups estimate the hardware behavior. For each problem p, candidate c, and target h, this procedure produces the feature vector $x = \phi(p, c, h)$; the feature matrix of the candidate catalogue is the sole input to the selector described in Section 4. Full profiling analysis, configuration-level correlations, and regime breakdowns are provided in Appendix B; we explain the features and the mathematical proxies in detail in Appendix C.

Table 2: Feature groups.

Group	Features and purpose
Structural core	$\log_2 M, \log_2 N, \log_2 K$; operand layouts; input, accumulation, and output types; tile and instruction shapes; pipeline stages; mainloop and epilogue schedules; cluster shape; tile scheduler; and architecture identifiers.
Work decomposition	Output-tile counts, reduction iterations, edge-tile waste, SM subscription, final-wave efficiency, Stream-K applicability, and cluster fit.
Memory behavior	Problem and tile arithmetic intensity, operand re-streaming, working-set and resident-panel size relative to L2, and input, data/epilogue/conversion traffic.
Resource pressure	Bytes per pipeline stage, total mainloop and epilogue shared-memory storage, register-pressure proxy, and shared-memory- and register-limited occupancy.
Pipeline behavior	Estimated TMA load time, WGMMA compute time, producer–consumer ratio, pipeline-fill fraction, stage amortization, and interactions between tile depth, stage count, and occupancy.

4 Training the Selector

Given an input configuration, we list the valid CUTLASS kernel catalogue, and use a machine learning model to rank the catalogue and choose the best kernel. In this section, we explain the data collection process for training and the model design.

4.1 Constructing The Training Dataset

The CUTLASS configuration space is too large to measure exhaustively for every problem shape. Even after validity filtering, a single BF16 GEMM layout exposes over sixty thousand candidate configurations. Exhaustively sweeping this space over thousands of shapes would require thousands of compiles and hundreds of millions of benchmarks.

We therefore separate the collection into two tasks. Our training dataset covers a broad range of GEMM shapes while also sampling candidates within each shape-layout group. The held-out evaluation dataset covers fewer shapes, but sweeps their valid candidates exhaustively to establish a trustworthy in-space oracle. We describe the evaluation dataset in detail in Appendix E. This section describes the training GEMM collection; the same collection design is used when extending the dataset to additional types, epilogues, and operator families.

4.1.1 Shape Selection

A GEMM problem shape is a triple (M, N, K). We define our structured anchor grid in 3 dimensions as $[32,16384]^3$ with each dimension snapped to a multiple of 32 – roughly 134 million lattice

points. The preferred CUTLASS kernel changes with problem geometry (square, tall, wide, skinny-K), reduction depth ($\lceil K/T_K \rceil$), and cache residency; a single random draw over this box would leave boundary (extreme large/small values of one/more dimensions) and hardware transition points severely underrepresented. We therefore use two complementary (but non-disjoint) construction axes: a *geometry class* (interior versus boundary/extreme) and, where needed, targeted *supplements* (anchor neighbours and transition points). The final dataset contains 593 base shapes ($\times$ 4 layouts $=$ 2372 unique groups):

- **Primary random coverage (469 shapes).** 297 *interior* shapes: log-uniform draws from the anchor grid. 172 *boundary* shapes: deliberate hull extremes; very tall, wide, skinny-K, and corner cases up to 16384 on an axis.
- **Targeted supplements (124 shapes).** 92 *anchor* shapes: Neighbours formed by ablating shapes. For every data point, the training set includes shapes that vary one dimension at a time where the domain permits it. This helps the model learn how changing different dimensions of the input shape changes performance and behaviour. This adds 47 interior and 45 boundary shapes. 32 *transition* shapes: points near the roofline ridge (BF16 arithmetic intensity $\approx$ 247.3 flop/byte on GH200) and the 50 MiB LLC-residency boundary. This adds 14 interior and 18 boundary shapes.

For a shape-layout group, we measure only a budgeted subset of the full catalogue during training. The default budget is 2000 candidates per group. It is reduced to 1000 for the largest shapes, where each benchmark is substantially more expensive, and increased to 3000 for atypical or transition shapes, where the configuration ranking changes more sharply. Uniform sampling fails because the performant tail is very small: only 2.4% of valid kernels lie within 5% of the oracle best on average, and the best kernel in a 1000-sized uniform sample reaches only 88.7% of the group oracle throughput on average, finding near-optimal kernels in fewer than 16% of groups. We present a study of sampling CUTLASS kernels and the performance distribution in Appendix G. Instead, we rank the valid candidates using a cheap, static efficiency score that ranks kernels likely to give better performance. Let c_M, c_N be the thead-block cluster shape, S_{SM} the number of SM clusters, S the pipeline stage count, and

$$n_c = \left\lceil \frac{M}{c_M T_M} \right\rceil \left\lceil \frac{N}{c_N T_N} \right\rceil$$

be the number of clusters required by the launch. The number of cluster waves is thus $W = \lceil \frac{n_c}{S_{\mathrm{SM}}} \rceil$. We use the following sampling score to rank kernels:

$$s_{\mathrm{sample}} = \frac{n_c}{W S_{\mathrm{SM}} c_M c_N} \cdot S \cdot \sqrt{T_M T_N T_K}$$

The first term favours efficient wave filling, penalizing oversized clusters that are too wasteful. The remaining terms favour configurations with deeper pipelines and sufficiently large tiles, which are likely to have higher reuse and provide better throughput. For a group budget, three quarters of the sampled configurations are drawn from the top of this ranking, while the remaining quarter is sampled across the rest of the ranked space. This preserves examples of poor and mediocre configurations. If the number of valid candidates is below the assigned budget, we collect the entire valid pool. The final training collection contains approximately 4.9 million measured kernels.

4.1.2 Collection Pipeline

We perform data collection on an HPC node equipped with 4 GH200 GPUs. We compile CUTLASS SM90a kernels from the commit `3476ddb7` using CUDA 13.1. Each measurement records the behavior of one candidate on one shape-layout group. Each successful benchmark consists of five warm-up iterations followed by five rounds of three timed iterations. Every attempted measurement is recorded, including successful executions, compilation failures, launch failures, crashes, and timeouts. We detail implementation optimizations in Appendix D.

4.2 Learning to Rank Candidates

We study two model families for mapping the hardware-aware representation $\phi(p, c, h)$ to a scalar candidate score. In both cases, candidates are grouped by GEMM problem p and supervision is derived from measured kernel throughput within that group. The resulting score is used only for

ordering candidates; at inference, the selected kernel is

$$\hat{c}(p) = \arg\max_{c \in C(p)} f_\theta(\phi(p, c, h)).$$

Thus, neither model is required to produce a calibrated prediction of performance. Exact label construction, model architecture, loss definitions, and hyperparameter ranges are given in Appendix F.

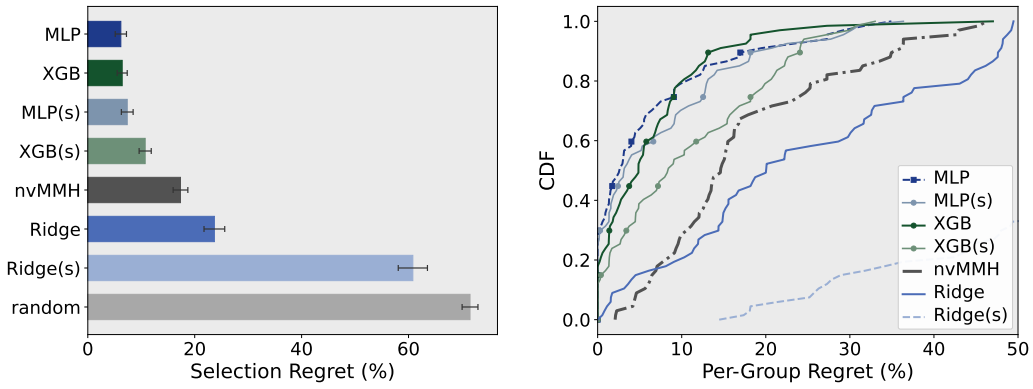
Gradient-boosted model. We train XGBoost models that assign a scalar score to each candidate. Numeric features are used directly and categorical features use fixed categories shared across training and inference. We consider three objectives: `rank:ndcg`, `rank:pairwise`, and squared-error regression on within-group normalized throughput. The ranking objectives use uncertainty-aware relevance labels that group configurations whose measured performance differs by less than measurement noise.

Feed-forward model. We also train multilayer perceptrons that score individual problem–candidate pairs. We standardize numeric features using training-set statistics and one-hot encode categorical features. We consider MSE on normalized throughput, RankNet pairwise loss, and LambdaRank. Pairwise objectives compare candidates only within the same GEMM group; we cap the number of sampled pairs per group while retaining comparisons involving high-performing candidates.

5 Evaluation

We run our experiments on an HPC node equipped with 4 GH200 GPUs running SUSE Linux Enterprise Server 15 SP6. We use CUTLASS from the commit `3476ddb7` using CUDA 13.1. The environment runs Python 3.12.3 with nvidia-matmul-heuristics-0.1.0.27 as the baseline; we are unable to compare against other related baselines (Swann et al. (2025); Yu et al. (2023)) since they are not implemented for the Hopper architecture. For each group g we define *selection regret* $R_g = 1 - T(\hat{c}_g) \,/\, T(c_g^\star)$, where $\hat{c}_g$ is the model's top-scoring config and $c_g^\star$ is the fastest measured config in that group. We train two model families, an MLP and XGBoost, under both the full hardware-aware representation and a structural-only ablation. To test whether the representation alone is sufficient, we also evaluate simpler linear and analytical selectors. Additional training-objective comparisons, feature ablations, and the relationship between validation NDCG and held-out regret are provided in Appendix H.1.

5.1 Evaluation on the Exhaustive Dataset



(a) Mean selection regret.

(b) CDF of per-group selection regret.

Figure 2: Selection regret of different model classes on the evaluation dataset.

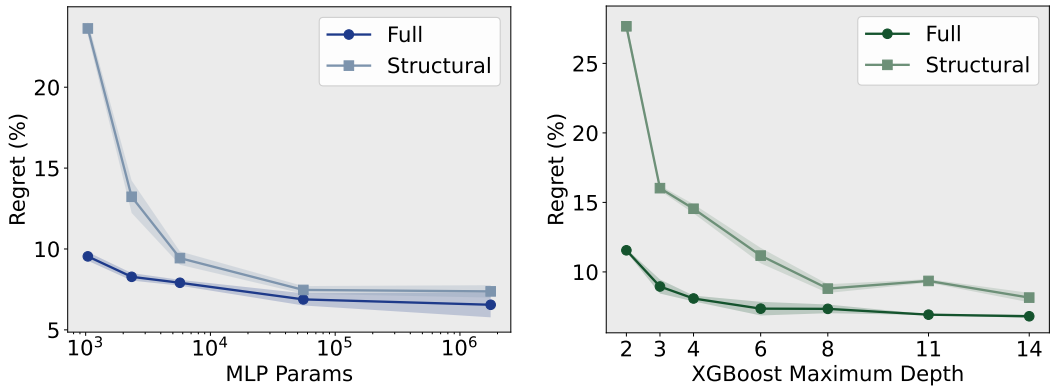
We consider an MLP with 1.76 M trainable parameters and an XGBoost ensemble of 916 trees with maximum depth 7. On the held out evaluation dataset (Figure 2a), the best hardware-aware MLP reduces mean regret to 6.2%, compared with 17.3% for the top-1 nvMMH recommendation and 71.5% for a random valid candidate. Hardware-aware XGBoost MSE follows closely at 6.4%. Hardware-aware features reduce mean regret on the best MLP from 7.4% to 6.2%; for XGBoost, they reduce it from 10.7% to 6.4%. The larger XGBoost gain indicates that the derived features substantially simplify the mapping from configuration parameters to performance.

A ridge regressor using the hardware-aware features reduces mean regret to 23.7%, compared with 60.8% using structural features alone, showing that the representation carries substantial selection signal even under a linear model. However, the linear selector remains well behind nvMMH (17.3%) and the nonlinear MLP/XGBoost models (6.2% or 6.4%), indicating that selection benefits from explicitly modeling the non-linear, problem-dependent interactions between the hardware-aware quantities. Detailed per-method selection statistics are reported in Appendix H.2.

Figure 2b shows that MLP and XGBoost place most of their mass at low regret: hardware-aware MLP and XGBoost have medians of 2.6% and 4.9%, compared with 14.6% for nvMMH, and hardware-aware models sit above their structural counterparts throughout the curve. Additional near-oracle and regime-specific evaluation results are provided in Appendices H.4 and H.5.

5.2 Model Size Ablations

We next ask whether the benefit of hardware-aware features persists as model capacity changes. We fix the MSE objective and sweep MLP widths from 16×8 to $1024\times1024\times512\times256$, and XGBoost `max_depth` over $\{2, 3, 4, 6, 8, 11, 14\}$, using three seeds per configuration and both feature sets. Figure 3a and Figure 3b report held-out mean selection regret.



(a) MLP of different parameter counts.

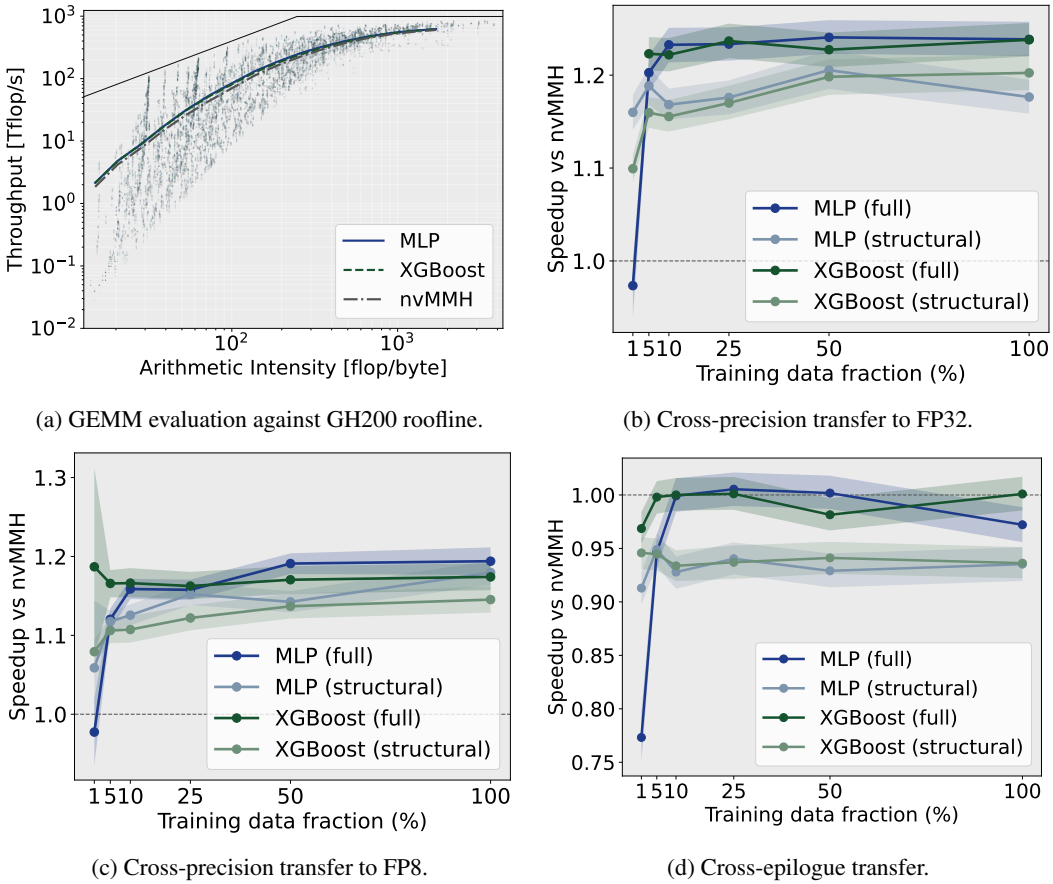
(b) XGB of different depths.

Figure 3: Selection regret on model capacity study across parameter ranges.

Hardware-aware features substantially reduce the model capacity required for accurate kernel selection. At the smallest MLP sizes, the structural model incurs up to 14 % higher regret, whereas the hardware-aware representation already captures much of the performance structure of the search space. As capacity increases, structural models gradually recover this gap, and only the largest MLP ($\sim$1.7 M parameters) matches the full representation. This effect is even more pronounced for tree models. XGBoost benefits from the hardware-aware representation throughout the entire depth sweep and retains an advantage even at the largest tested depth. In this sense, the hardware-aware features do not merely improve final accuracy; they make kernel-selection itself easier to learn.

5.3 Generalization and Broad GEMM Evaluation

We evaluate whether the same selectors generalize at scale to $8{,}000$ previously unseen BF16 GEMMs. Figure 4a places the selected kernels against the GH200 roofline. The hardware-aware MLP and XGBoost selectors produce nearly identical performance envelopes despite their very different model architectures, and both consistently outperform nvMMH across the full arithmetic-intensity range. The improvement persists from memory-bound GEMMs through the transition region and into compute-bound problems, indicating that the learned representation captures performance effects beyond any single bottleneck regime. More importantly, the improvement is broad rather than driven by a small number of outliers: MLP outperforms nvMMH on 79.6% of the $8{,}000$ problems, and XGBoost on 77.4%. Additional details on workload construction, scheduler controls, roofline methodology, and performance breakdown are provided in Appendix H.6.



(a) GEMM evaluation against GH200 roofline.

(b) Cross-precision transfer to FP32.

(c) Cross-precision transfer to FP8.

(d) Cross-epilogue transfer.

Figure 4: Model Generalization and Transfer experiments.

5.4 Transfer Learning Experiments

We evaluate whether a selector trained on BF16 can be adapted efficiently to new precisions. We transfer the BF16 models to FP32 and FP8 E4M3 GEMMs using nested subsets of the corresponding target-dtype training corpus, and evaluate on held-out problems against nvMMH.

Figures 4b and 4c show that only modest target-dtype supervision is required to recover strong performance. On FP32, hardware-aware selectors already reach approximately $1.20\times$–$1.22\times$ geometric-mean speedup over nvMMH using only 5% of the available target shapes, and improve to approximately $1.24\times$ with the full target corpus. The corresponding structural models generally trail the hardware-aware representation, particularly in the low-data regime. FP8 E4M3 presents a noisier, more difficult transfer setting, but the same trend persists. Hardware-aware MLPs reach $1.12\times$ speedup with 5% of the target shapes and $1.19\times$ with the full corpus, while XGBoost remains around $1.15\times$–$1.17\times$ across the sweep. Together, these results show that the selectors retain useful structure across changes in numerical precision, and that exposing candidate-induced hardware behavior improves adaptation when target-domain measurements are limited.

We repeat the same finetuning protocol for FP16 epilogue-fusion GEMMs, fusing ReLU, GeLU, and/or bias, warm-starting from the unfused BF16 checkpoints. Figure 4d summarizes the results on held-out fusion GEMMs. With only 10% of the fusion corpus, hardware-aware MLP and XGBoost already match nvMMH ($1.00\times$ and $1.00\times$ geometric-mean speedup, respectively) and outperform the structural ablation ($0.93\times$). Additional transfer methodology, coverage statistics, initialization details, and per-method results are provided in Appendix H.7.

5.5 DeepBench Case Study

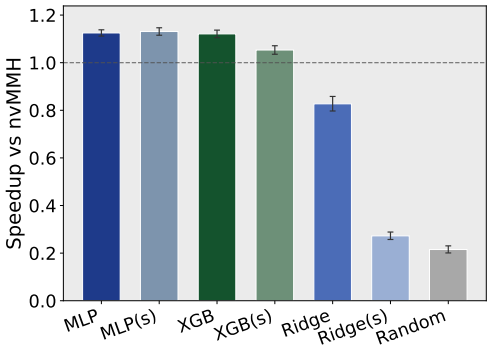


Figure 5: DeepBench Speedup vs. nvMMH

We additionally evaluate on 151 dense GEMM shapes from the public DeepBench catalogue (Baidu (2016)), providing an externally defined set of speech- and language-modeling workloads independently defined from our synthetic shape distributions. We compare each selector's rank-1 choice against nvMMH's best schedule realization. As shown in Figure 5, the hardware-aware MLP and XGBoost selectors achieve $1.12\times$–$1.13\times$ geometric-mean speedup over nvMMH with 100% execution coverage. In contrast, the ridge and random baselines remain below parity, indicating that the gains persist on realistic deep-learning GEMM shapes. These results show that the learned selector's advantage is not confined to our synthetic evaluation distribution, but carries over to independently specified deep-learning workloads with substantially different and more irregular problem shapes.

6 Related Work

Learned cost models guide iterative search over tensor-program spaces (Li et al. (2020); Zheng et al. (2020); Choi et al. (2022); Zhang et al. (2026a); Bai et al. (2025)) in frameworks like AutoTVM (Chen et al. (2018); Wu et al. (2023)) and Ansor (Zheng et al. (2020)). While corpora like TenSet (Zheng et al. (2021)) and optimization methods like AdaTune (Li et al. (2020)) or TLP (Zhai et al. (2023)) improve search efficiency, our model scores the entire valid CUTLASS configuration catalogue in a single batch to directly select the optimal implementation. While device-transfer frameworks (Ryu & Sung (2021); Zhai et al. (2023); Bai et al. (2025)) rely on complex learned program representations, instead we show that explicit, mechanism-level hardware features provide a strong inductive bias that improves transfer accuracy with fewer target-domain measurements.

While predictive models like ISAAC (Tillet & Cox (2017)), CUTLASS-tailor (Yu et al. (2023)), and related low-bit selectors (Guo et al. (2024; 2026)) show that GEMM execution cost is learnable from configuration descriptors (Liu & Halim (2024)), our model uses analytical hardware-behavior features to directly rank and select optimal configurations. Manual analytical models (Ran et al. (2026); Jarmusch & Chandrasekaran (2026)) and heuristics (Barclay et al. (2025); NVIDIA (2025); Swann et al. (2025); Zhang et al. (2026b)) avoid training data by hardcoding hardware-mapping rules, but we learn these complex interactions from behavioral features instead.

7 Conclusion

We presented a hardware-aware approach to learned CUTLASS kernel selection. Rather than asking a model to infer GPU behavior from raw configuration parameters, we augment each candidate with statically computable estimates of induced execution behavior. Using 4.9 million measured CUTLASS kernels, the resulting representation achieves mean selection regret of 6.2% with an MLP and 6.4% with XGBoost, compared with 17.3% for the top-1 nvMMH recommendation. Its benefit across both model families indicates that the improvement comes from the representation rather than a particular predictor architecture.

More broadly, our results suggest that learned performance models benefit from representing what a configuration does to the hardware, rather than only what the configuration is. Raw parameters often hide effects whose meaning changes with workload and execution regime, while fully analytical models require brittle architecture-specific rules. Hardware-aware representations provide an intermediate abstraction: architectural knowledge exposes the relevant mechanisms, while data learns their interactions. As accelerator configuration spaces grow, such behavioral representations may provide a reusable basis for performance models across precisions, operators, and architectures. The central implication is that better performance models may come less from larger models than from better representations aligned with hardware execution.

AI USE STATEMENT

In this work, we used generative AI tools to assist with the implementation and debugging of benchmarking and analysis code, interpret intermediate experimental results, and assist with language editing of the manuscript. We did not use generative AI tools to generate synthetic experimental data, formulate or prove mathematical claims, or fabricate experimental measurements or results. Tasks involving surveys, interviews, transcription of research material, and qualitative or thematic data analysis were not applicable to this work.

All AI-assisted research ideas, methodological suggestions, code, analyses, and manuscript text were reviewed by the authors. AI-assisted code was inspected and validated through compilation, testing, and comparison against measured data before being used in the experimental pipeline. Literature suggestions and technical claims were checked against the corresponding papers or documentation. Experimental results were obtained from the authors' own measurement infrastructure, and their interpretation and the final scientific conclusions were determined by the authors. We take responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide the information required to reproduce our results throughout the paper and appendix. Section 4.1 and Appendix G describe the construction of the training dataset, candidate-sampling procedure, measurement protocol, and software environment. Section 3 and Appendix C define the structural and hardware-aware features used by the selectors, while the model architectures, objectives, preprocessing, and training procedure are described in Section 4 and Appendix F. The metrics and evaluation details are documented in Section 5 and Appendix H. We provide the generated datasets, benchmark and feature-extraction code, training and evaluation scripts, configuration files, and exact software revisions used in the experiments as supplementary material as an anonymized repository; upon publication, we will open-source the source code and model weights.

REFERENCES

Yang Bai, Mingjun Li, Wendong Xu, and Bei Yu. A learned performance model with transfer learning across gpus on tensorized instructions. *IEEE Transactions on Parallel and Distributed Systems*, 36(9):1904–1919, 2025.

Baidu. Deepbench: Benchmarking deep learning operations on different hardware. `https://github.com/baidu-research/DeepBench`, 2016.

Harrison Barclay, Ian Tramble, Michał Kukuła, Michel Migdal, Nicholai Tukanov, and Leopold Cambier. Improving gemm kernel auto-tuning efficiency on nvidia gpus with heuristics and cutlass 4.2 — nvidia technical blog, 9 2025. URL `https://developer.nvidia.com/blog/improving-gemm-kernel-auto-tuning-efficiency-on-nvidia-gpus-with-heuristics-and-cutlass-4-2/`.

Ganesh Bikshandi and Jay Shah. Developing cuda kernels for accelerated matrix multiplication on nvidia hopper architecture using the cutlass library. *Colfax Research*, 2023.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.

Junkyeong Choi, Hyucksung Kwon, Woongkyu Lee, Jungwook Choi, and Jieun Lim. Learning from distinctive candidates to optimize reduced-precision convolution program on tensor cores. *arXiv preprint arXiv:2202.06819*, 2022.

Hong Guo, Nianhui Guo, Christoph Meinel, and Haojin Yang. Low-bit cutlass gemm template autotuning using neural network. In *2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 394–401, 2024. doi: 10.1109/ISPA63168.2024.00057.

Hong Guo, Nianhui Guo, Christoph Meinel, and Haojin Yang. Leveraging large-scale data for efficient low-bit cutlass gemm optimization via neural networks. *Big Data Mining and Analytics*, 9(2):632–652, 2026. doi: 10.26599/BDMA.2025.9020065.

Aaron Jarmusch and Sunita Chandrasekaran. Microbenchmark-driven analytical performance modeling across modern gpu architectures, 2026. URL https://arxiv.org/abs/2605.04178.

Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: adaptive tensor program compilation made efficient. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Xiaoteng Liu and Pavly Halim. Understanding gemm performance and energy on nvidia ada lovelace: A machine learning-based analytical approach, 2024. URL https://arxiv.org/abs/2411.16954.

Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. Benchmarking and dissecting the nvidia hopper gpu architecture. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 656–667, 2024. doi: 10.1109/IPDPS57955.2024.00064.

NVIDIA. Nvidia matmul heuristics — nvmatmulheuristics, 2025. URL https://docs.nvidia.com/cuda/nvidia-matmul-heuristics/.

Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D. Owens. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, pp. 429–431, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700156. doi: 10.1145/3572848.3577479. URL https://doi.org/10.1145/3572848.3577479.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. Kernelband: Steering llm-based kernel optimization via hardware-aware multi-armed bandits, 2026. URL https://arxiv.org/abs/2511.18868.

Jaehun Ryu and Hyojin Sung. Metatune: Meta-learning based cost model for fast and efficient auto-tuning frameworks. *arXiv preprint arXiv:2102.04199*, 2021.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: fast and accurate attention with asynchrony and low-precision. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Rupanshu Soi, Rohan Yadav, Fredrik Kjolstad, Alex Aiken, Maryam Mehri Dehnavi, Michael Garland, and Michael Bauer. Optimal software pipelining and warp specialization for tensor core gpus, 2025. URL https://arxiv.org/abs/2512.18134.

Ryan Swann, Muhammad Osama, Xiaohu Guo, Bryant Nelson, Lixun Zhang, Alex Brown, Yen Ong, Ali Yazdani, Sean Siddens, Ganesh Dasika, et al. tritonblas: Triton-based analytical approach for gemm kernel parameter selection. *arXiv preprint arXiv:2512.04226*, 2025.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Aditya Atluri, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. Cutlass, 1 2023. URL https://github.com/NVIDIA/cutlass/tree/v3.0.0.

Vijay Thakkar, Cris Cecka, Tejash Shah, Jay Shah, Paul VanKoughnett, and Ryo Asai. Cutlass 3.x: Orthogonal, reusable, and composable abstractions for gemm kernel design — nvidia technical blog, 7 2025. URL https://developer.nvidia.com/blog/cutlass-3-x-orthogonal-reusable-and-composable-abstractions-\for-gemm-kernel-design/.

Philippe Tillet and David Cox. Input-aware auto-tuning of compute-bound hpc kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351140. doi: 10.1145/3126908.3126939. URL https://doi.org/10.1145/3126908.3126939.

Xingfu Wu, Praveen Paramasivam, and Valerie Taylor. Autotuning apache tvm-based scientific applications using bayesian optimization. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, pp. 29–35, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707858. doi: 10.1145/3624062.3626079. URL https://doi.org/10.1145/3624062.3626079.

Yongseung Yu, Donghyun Son, Younghyun Lee, Sunghyun Park, Giha Ryu, Myeongjin Cho, Jiwon Seo, and Yongjun Park. Tailoring cutlass gemm using supervised learning. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pp. 465–474, 2023. doi: 10.1109/ICCD58817.2023.00077.

Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pp. 833–845, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575737. URL https://doi.org/10.1145/3575693.3575737.

Kaixuan Zhang, Chutong Ding, Shiyou Qian, Luping Wang, Jian Cao, Guangtao Xue, Cheng Huang, Guodong Yang, and Liping Zhang. Wavetune: Wave-aware bilinear modeling for efficient gpu kernel auto-tuning. *arXiv preprint arXiv:2604.10187*, 2026a.

Xiao Zhang, Yaoyao Ding, Bolin Sun, Yang Hu, Tatiana Shpeisman, and Gennady Pekhimenko. Hexcute: A compiler framework for automating layout synthesis in gpu programs, 2026b. URL https://arxiv.org/abs/2504.16214.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association. ISBN 978-1-939133-19-9.

Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

A Background

A.1 GPU Execution Model

An NVIDIA GPU consists of many streaming multiprocessors (SMs), each containing the compute units and on-chip storage used for execution. A kernel launches a grid of thread blocks, and each block is assigned to an SM independently. Threads within a block execute in groups of 32 called warps. Warps can cooperate through on-chip shared memory and synchronization, while registers are private to individual threads. The output of a kernel is therefore determined not only by the arithmetic performed, but also by how work, data, and synchronization are distributed across threads, warps, blocks, and the memory hierarchy.

For GEMM, the main data movement path is from global memory through shared memory to registers. A thread block typically loads tiles of the input matrices into shared memory, computes on those tiles using registers as accumulators, and eventually stores its output tile back to global memory. Shared memory provides lower-latency reuse within a block, while the L2 cache and device

memory serve data shared across blocks and data that does not fit on chip. The amount of work and data assigned to a block determines its arithmetic intensity, memory traffic, shared-memory footprint, register usage, and ability to keep the SM occupied.

Only a limited number of blocks can reside on an SM at once. This limit is determined by the block's register and shared-memory requirements, its thread count, and architectural limits on blocks and warps. Higher residency can help hide memory and instruction latency, but increasing the amount of work assigned to a block often increases resource usage and can reduce the number of simultaneously resident blocks. Consequently, the best GEMM configuration must balance reuse and computational efficiency against resource pressure.

A.2 Hopper Execution Mechanisms

Our primary evaluation targets NVIDIA Hopper GPUs, whose SM90a architecture introduces several mechanisms that substantially affect GEMM performance (Luo et al. (2024)). Hopper tensor-core operations are issued using warpgroup matrix multiply-accumulate (WGMMA) instructions, comprising four consecutive warps, or 128 threads, cooperating to issue an asynchronous matrix operation. The instruction shape and warpgroup organization impose further constraints on the tile shapes that a kernel can use. Hopper also provides the Tensor Memory Accelerator (TMA), which asynchronously transfers multidimensional tensor regions between global memory and shared memory, saving several individual thread address computations and data loads.

TMA and WGMMA enable warp-specialized pipelines (Soi et al. (2025)), where different warps assume different duties. Producer warps issue asynchronous TMA transfers, while consumer warpgroups perform WGMMA on previously loaded tiles. The shared-memory buffer is divided into a multi-stage pipeline, allowing memory movement to run ahead of computation. Increasing the number of stages can hide transfer latency but increase shared-memory consumption.

Hopper additionally supports thread-block clusters, which can cooperate through distributed shared memory, or through TMA multicast which can deliver the same data to multiple blocks. Clusters can reduce memory traffic but impose scheduling and residency constraints. Whether clustering is beneficial therefore depends on the problem shape, tile decomposition, and resource requirements of the candidate kernel.

These mechanisms make the relationship between a CUTLASS configuration and hardware performance highly indirect. A configuration parameter such as pipeline depth, schedule, or cluster shape changes several hardware-level properties simultaneously. The resulting effects cannot generally be inferred from the parameter value alone.

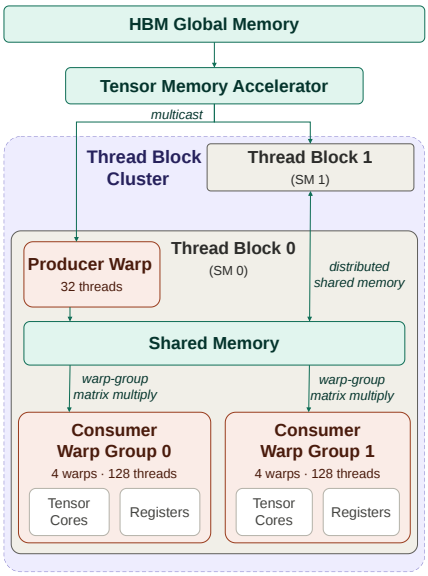


Figure 6: H100 Programming Model

A.3 GEMM Tiling and CUTLASS

We consider the general matrix multiplication $C = \alpha AB + \beta C$, where $A \in \mathbb{R}^{M\times K}$, $B \in \mathbb{R}^{K\times N}$, and $C \in \mathbb{R}^{M\times N}$. The operation performs $(2MNK)$ floating-point operations. A GPU implementation partitions C into output tiles. A thread block computes one or more such tiles by iterating over the reduction dimension in chunks.

A candidate configuration is characterized primarily by a tile shape (T_M, T_N, T_K). The first two dimensions determine the tile computed by a block, while T_K determines the amount of the reduction processed in one mainloop iteration. Each iteration loads an $T_M \times T_K$ tile of A and a $T_K \times T_N$ tile of B, performs a matrix multiply-accumulate, and advances to the next K-tile. The number of iterations is thus $\lceil K/T_K \rceil$. The tile shape therefore affects the number of output tiles, boundary

tile waste, effective arithmetic intensity, memory traffic, shared-memory and register usage, and the number of mainloop iterations.

CUTLASS builds GEMM kernels by composing reusable mainloop and epilogue components. The mainloop loads input tiles and performs the matrix multiplication, while the epilogue applies operations such as scaling, bias addition, activation, residual addition, type conversion, and the final store. CUTLASS exposes all aforementioned configurations as template parameters, and these choices are coupled – for e.g., a larger tile may increase data reuse but increase register and shared-memory usage. More pipeline stages can make the pipeline more efficient, but consume additional resources and reduce occupancy. A schedule may require a particular tile divisibility or warpgroup arrangement. Clusters may reduce redundant loads while limiting which blocks can be resident together. Epilogues add their own computation, memory traffic, and resource requirements, and can change which mainloop configuration is best. Not every combination of choices produces a valid kernel. CUTLASS implicitly imposes architectural and implementation constraints on several of the parameters. The valid candidate set is therefore a tiny constrained subset of the cartesian product of the parameters. The configuration does not directly specify performance – making selecting an efficient candidate from this valid set highly non-trivial.

A.4 A Generated Kernel

The snippet below is a representative CUTLASS kernel: BF16 operands with FP32 accumulation, tile $128 \times 128 \times 64$, a 1×1 cluster, four mainloop pipeline stages, the plain TMA warp-specialized mainloop, a TMA warp-specialized epilogue, persistent tile scheduling, and the `TN` operand layout (A row-major, B column-major). The configuration name is recorded in the generated comment header.

```
// ... unrelated generated boilerplate omitted (includes, CUDA/CUTLASS checks) ...

// Config: cutlass3x_sm90_tensorop_bf16_bf16_f32_bf16_bf16_
//         128x128x64_1x1x1_4_tnn_align64_warpspecialized_epi_tma

namespace Kernel_0 {

    using           ElementA    = cutlass::bfloat16_t;
    using           LayoutATag  = cutlass::layout::RowMajor;
    constexpr int   AlignmentA  = 64;

    using           ElementB    = cutlass::bfloat16_t;
    using           LayoutBTag  = cutlass::layout::ColumnMajor;
    constexpr int   AlignmentB  = 64;

    using           ElementD    = cutlass::bfloat16_t;
    using           ElementC    = cutlass::bfloat16_t;
    using           LayoutCTag  = cutlass::layout::ColumnMajor;
    using           LayoutDTag  = cutlass::layout::ColumnMajor;
    constexpr int   AlignmentD  = 64;
    constexpr int   AlignmentC  = 64;

    using ElementAccumulator    = float;
    using ElementCompute        = float;
    using ArchTag               = cutlass::arch::Sm90;
    using OperatorClass         = cutlass::arch::OpClassTensorOp;

    using MmaTileShape          = Shape<_128, _128, _64>;
    using ClusterShape          = Shape<_1, _1, _1>;
    using StageCount            = cutlass::gemm::collective::StageCount<4>;
    using TileSchedulerType     = cutlass::gemm::PersistentScheduler;

    using MainloopScheduleType  = cutlass::gemm::KernelTmaWarpSpecialized;
    using EpilogueScheduleType  = cutlass::epilogue::TmaWarpSpecialized;
    using EpilogueTile          = cutlass::epilogue::collective::EpilogueTileAuto;

    using CollectiveEpilogue = typename cutlass::epilogue::collective::CollectiveBuilder<
        ArchTag, OperatorClass,
        MmaTileShape, ClusterShape,
        EpilogueTile,
        ElementAccumulator, ElementAccumulator,
        ElementC, LayoutCTag, AlignmentC,
        ElementD, LayoutDTag, AlignmentD,
        EpilogueScheduleType
    >::CollectiveOp;

    using CollectiveMainloop = typename cutlass::gemm::collective::CollectiveBuilder<
```

```
        ArchTag, OperatorClass,
        ElementA, LayoutATag, AlignmentA,
        ElementB, LayoutBTag, AlignmentB,
        ElementAccumulator,
        MmaTileShape, ClusterShape,
        StageCount,
        MainloopScheduleType
    >::CollectiveOp;

    using GemmKernel = cutlass::gemm::kernel::GemmUniversal<
        Shape<int,int,int,int>,
        CollectiveMainloop,
        CollectiveEpilogue,
        TileSchedulerType>;

    using Gemm = cutlass::gemm::device::GemmUniversalAdapter<GemmKernel>;
}
```

Reading top to bottom: `ElementA/B/C` and `ElementAccumulator` fix the operand and accumulation types; `LayoutA/B` are the searched operand layouts while `LayoutC/D` are always column-major in our generator (row-major C is obtained offline via the transpose–swap identity $C^\top = B^\top A^\top$). `AlignmentA/B/C` are derived as 128/sizeof(element) bytes of vectorization. `ArchTag` and `OperatorClass` pin the kernel to SM90 tensor-core WGMMA. `MmaTileShape` and `ClusterShape` are the thread-block tile $T_M \times T_N \times T_K$ and cluster $C_M \times C_N \times C_K$; `StageCount` sets the software-pipelined mainloop depth. `MainloopScheduleType`, `EpilogueScheduleType`, and `TileSchedulerType` select the Hopper TMA warp-specialized mainloop variant, epilogue implementation, and persistent vs. Stream-K tile scheduling. The two `CollectiveBuilder` aliases materialize epilogue and mainloop collectives; `GemmKernel` and `Gemm` wrap them into a universal device GEMM. The setup excerpt shows the runtime path our benchmark uses: packed CuTe strides, `GemmUniversalMode::kGemm`, alpha/beta epilogue scalars, a hardware-info struct, then `can_implement` and `initialize` before timed `run`. Every generated source has essentially the same structure; the search space is formed by varying a small set of template/configuration choices while leaving the GEMM semantics unchanged. Table 3 lists the configuration space.

A.5 Configuration Space

For a fixed GEMM problem shape, BF16 precision, and operand-layout tag, the generator varies tile geometry, cluster shape, pipeline depth, mainloop and epilogue schedules, and the tile scheduler. Operand element types, accumulator type, architecture tag, operator class, C/D layout, alignments, and epilogue-tile sizing are not independent search axes: they are fixed by the precision tuple or left to CUTLASS defaults (`EpilogueTileAuto`). Rasterization order, swizzle patterns, and classic split-K are likewise not exposed by our API; Stream-K is the only searched alternative to persistent scheduling, and CUTLASS resolves remaining lowering details internally.

The raw Cartesian product over the searched axes alone contains 2,138,400 combinations. Static validity filtering leaves 242,971 BF16 configurations.

A.6 Validity Constraints

The Cartesian product contains many combinations that cannot instantiate as valid SM90 kernels. Enumeration is therefore followed by our set of static predicates; we also statically remove wasteful candidates with $T_N > 2M$ or $T_N > 2N$. Table 4 lists the constraints active for BF16.

B Profiling Analysis for Feature Construction

This section gives the detailed profiling analysis underlying the feature-design methodology in Section 3. The purpose of profiling is to identify execution effects that distinguish fast and slow CUTLASS configurations and can subsequently be approximated statically.

Table 3: Configuration knobs exposed by the BF16 SM90 generator (`generate_search_space`).

Knob	Values / domain	Meaning
`tile_m`	$\{64, 128, 192, 256\}$	Thread-block M tile extent
`tile_n`	$\{16, 32, 48, 64, 80, 96, 128, 192, 256\}$	Thread-block N tile extent
`tile_k`	$\{32, 64, 128, 256, 512\}$	Thread-block K tile / pipeline atom
`stages`	$\{2, \ldots, 12\}$	Mainloop pipeline stage count
`cluster_m`, `cluster_n`	$\{1, 2, 4, 8, 16\}$, $C_M C_N \leq 16$	CTA cluster shape in M and N
`cluster_k`	fixed 1	K-clustering not searched
`kernel_schedule`	WS / Pingpong / Cooperative	TMA warp-specialized mainloop policy
`epilogue_schedule`	NoSmem / TMA / TMA-Cooperative	Epilogue store path
`scheduler`	Persistent / Stream-K	Tile scheduling across the output grid
`layout`	TN, TT, NN, NT	Operand layouts for A and B
Fixed (not searched independently)		
Operand / acc. types	BF16 in, FP32 acc. (BF16 sweep)	CUTLASS element aliases
`alignment_*`	128 bit / element size	TMA vectorization requirement
C/D layout	ColumnMajor	Row-major output via transpose trick
`ArchTag` / `OpClass`	Sm90 / TensorOp	Target architecture
`EpilogueTile`	`EpilogueTileAuto`	Sub-tile shape chosen by CUTLASS

Table 4: Static validity constraints for BF16 SM90 GEMM configurations.

Constraint	Reason
Stream-K $\Rightarrow$ cooperative mainloop	CUTLASS Stream-K scheduler requires the cooperative TMA mainloop.
Cooperative mainloop $\Rightarrow T_M \equiv 0 \pmod{128}$	Two consumer warpgroups split the M tile; each half must stay 64-row WGMMA aligned.
$S_{\text{mainloop}} + S_{\text{epi}} + 2\,\text{KiB} \leq 227\,\text{KiB}$	Hopper shared-memory capacity per CTA; S_{epi} depends on epilogue schedule and tile via `estimate_epilogue_smem_bytes`.
If A is column-major: $T_K \leq 256\, C_N$	TMA smem box K extent is multicast-split across the N cluster dimension.
If B is row-major: $T_K \leq 256\, C_M$	Symmetric K-box limit along the M cluster dimension for the transposed operand.

B.1 Profiling setup

We collected 408 Nsight Compute profiles of BF16 CUTLASS GEMM kernels on GH200 across 17 problem shapes. Profiles are associated with fixed (M, N, K, layout) groups, within which only the CUTLASS candidate configuration changes. For every profiled candidate, we pair the collected hardware counters with the throughput measured by our benchmark harness.

Because many hardware counters scale strongly with the amount of work in the GEMM, correlations computed over all profiles can be misleading. For example, larger problems can simultaneously execute more tensor-core instructions, generate more memory traffic, and achieve higher sustained throughput. To separate these problem-size effects from candidate quality, for each metric x we compute

$$\rho_g(x) = \text{Spearman}\big(\{x(c)\}_{c \in C_g}, \{T(c)\}_{c \in C_g}\big)\,,$$

where g is a fixed shape–layout group, C_g is the set of profiled candidates in that group, and $T(c)$ is measured benchmark throughput.

Table 5 reports the complete set of counters used in this analysis.

Table 5: Spearman correlation between profiled execution metrics and measured benchmark throughput.

Metric	Global ρ	Mean ρ_g	Median ρ_g
Throughput and pipeline activity			
tensor_active_pct	$+0.992$	$+0.392$	$+0.492$
dram_throughput_pct	$+0.817$	$+0.408$	$+0.500$
compute_throughput_pct	$+0.977$	$+0.235$	$+0.382$
tma_active_pct	$+0.792$	$+0.181$	$+0.289$
sm_active_pct	$+0.910$	-0.001	$+0.024$
Memory traffic and hierarchy			
dram_read_bytes	$+0.903$	-0.055	-0.093
dram_write_bytes	$+0.817$	$+0.000$	$+0.086$
l2_read_sectors	$+0.942$	-0.102	-0.238
l2_write_sectors	$+0.908$	-0.151	-0.184
l2_throughput_pct	$+0.868$	$+0.142$	$+0.314$
l2_hit_rate	-0.393	-0.058	-0.087
l2_read_hit_rate	$+0.003$	-0.086	-0.194
l2_write_hit_rate	$+0.243$	$+0.110$	$+0.000$
Occupancy and resource use			
achieved_occupancy_pct	$+0.388$	$+0.140$	$+0.128$
registers_per_thread	$+0.736$	-0.151	-0.315
warps_active	$+0.388$	$+0.112$	$+0.145$
eligible_warps_per_cycle	$+0.337$	-0.148	-0.191
Warp stalls			
stall_barrier_pct	$+0.894$	$+0.156$	$+0.143$
stall_long_scoreboard_pct	-0.679	$+0.193$	$+0.267$
stall_short_scoreboard_pct	-0.863	$+0.029$	$+0.029$
stall_gmma_pct	$+0.540$	-0.158	-0.217
stall_mio_pct	-0.100	$+0.217$	$+0.227$
stall_wait_pct	-0.519	-0.113	-0.143
stall_not_selected_pct	$+0.054$	-0.224	-0.238

B.2 Global correlation is often a problem-size effect

Several metrics appear to be excellent performance predictors when all profiles are pooled, yet lose almost all ranking power once the problem is fixed. For example, sm_active_pct has global $\rho = +0.910$ but mean within-group $\rho_g = -0.001$. Likewise, l2_read_sectors drops from $+0.942$ globally to -0.102 within groups, and dram_read_bytes drops from $+0.903$ to -0.055. These quantities primarily distinguish different amounts of work rather than good candidates from bad candidates for the same GEMM.

Two execution quantities remain substantially correlated with candidate quality after fixing the problem. dram_throughput_pct has mean within-group $\rho_g = +0.408$, and tensor_active_pct has mean $\rho_g = +0.392$; both have positive correlation in 83.7% of computable groups. These measurements suggest that successful candidates simultaneously sustain the memory system and keep the tensor-core pipeline active. Since both quantities require executing the kernel, we instead construct static proxies for the mechanisms that produce them: data reuse, traffic, cache-relative working sets, feed–compute balance, pipeline fill, and pipeline amortization.

B.3 Measured symptoms versus controllable causes

Several counters are useful diagnostically but unsuitable as direct selector features. Barrier stalls, for example, have global $\rho = +0.894$ but only mean within-group $\rho_g = +0.156$. Long-scoreboard stalls even change apparent interpretation: their global correlation is -0.679, while the mean within-group correlation is $+0.193$. Such counters describe the state of an asynchronous execution pipeline after scheduling, resource allocation, and overlap have already taken effect.

This distinction is also visible when relating configuration parameters to measured counters. Pipeline depth has mean within-group correlation -0.357 with long-scoreboard stalls (median

-0.439), consistent with additional stages being associated with fewer exposed memory dependencies. However, stage count itself has only weak correlation with throughput. We therefore represent the quantities that determine the consequences of staging, such as bytes per stage, total shared-memory demand, residency, pipeline fill, and stage amortization, rather than treating the stall counter or stage count as a complete performance signal.

B.4 Static Configuration Effects

We additionally apply the same within-group analysis to candidate parameters and statically derived quantities. Table 6 summarizes the strongest effects.

Table 6: Correlation between selected static candidate quantities and measured throughput.

Feature	Global ρ	Mean ρ_g	Median ρ_g
`tile_m`	$+0.626$	-0.261	-0.409
`tile_n`	$+0.827$	$+0.080$	$+0.091$
`tile_k`	-0.415	$+0.257$	$+0.272$
`k_iters`	$+0.826$	-0.260	-0.289
`reg_pressure_proxy`	$+0.827$	-0.222	-0.319
`cluster_size`	-0.337	-0.247	-0.223
`stages`	-0.001	-0.111	-0.145
`bytes_per_stage`	$+0.396$	$+0.054$	$+0.092$
`pipeline_fill_frac`	$+0.721$	$+0.233$	$+0.171$
`restream_factor`	$+0.399$	$+0.115$	$+0.258$

These results illustrate why raw template parameters alone are insufficient. For example, `stages` has almost zero global correlation and only weak negative within-group correlation with throughput, even though stage depth directly affects the asynchronous mainloop. Its performance effect depends on how much storage each stage requires, how many K-iterations are available to amortize pipeline startup, and whether the resulting resource footprint reduces residency. The representation therefore includes both the raw stage count and derived quantities describing these interactions.

Similarly, T_K has positive mean within-group correlation with throughput ($+0.257$), while the corresponding number of reduction iterations has correlation -0.260. However, T_K is not itself an arithmetic-intensity proxy: in the tile-level ratio

$$I_{\text{tile}} = \frac{2T_M T_N T_K}{b_A T_M T_K + b_B T_N T_K} = \frac{2T_M T_N}{b_A T_M + b_B T_N},$$

T_K cancels. Instead, changing T_K modifies reduction granularity, the number of mainloop iterations, per-stage storage, and the amount of computation available between pipeline synchronization points.

B.5 Regime Dependence

The effect of a configuration parameter need not be uniform over the shape space. To test this, we repeat the within-group analysis after partitioning the held-out problems by size, geometry, and arithmetic-intensity regime.

The strongest example is T_K. Its effect remains positive across coarse size regimes, with mean within-group correlations of $+0.217$, $+0.307$, and $+0.245$ for small, medium, and large problems respectively, and $+0.265$ for compute-heavy problems. The magnitude changes substantially with geometry: the mean correlation is $+0.549$ for wide GEMMs, $+0.116$ for square GEMMs, and $+0.021$ for tall GEMMs. Skinny-K problems yield $+0.268$.

These results show that the strength of its effect depends strongly on problem geometry and execution regime. More generally, this motivates exposing both candidate choices and the problem quantities that determine how those choices should be interpreted.

The same principle guides the complete feature map: a feature is retained when it is available before execution, distinguishes configurations or identifies the regime in which they operate, and has a plausible connection to an execution mechanism. Appendix C gives the resulting analytical definitions.

Table 7: Within-group correlation between T_K and throughput under selected problem regimes.

Regime	Mean ρ_g	Median ρ_g
Small	$+0.217$	$+0.262$
Medium	$+0.307$	$+0.412$
Large	$+0.245$	$+0.273$
Compute-heavy	$+0.265$	$+0.289$
Memory-heavy	$+0.238$	$+0.262$
Wide	$+0.549$	$+0.577$
Square	$+0.116$	$+0.237$
Tall	$+0.021$	-0.010
Skinny-K	$+0.268$	$+0.268$

C Feature Set Design

C.1 Structural Core

The structural core contains the quantities required to identify a candidate: problem dimensions, operand layouts and types, threadblock and instruction tile shapes, stage count, mainloop and epilogue schedules, cluster shape, tile scheduler, and operator and compute-engine identifiers. Architecture descriptors provide the capacities and rates used by the derived features, including the number of SMs, shared-memory and register-file capacity per SM, LLC capacity, HBM bandwidth, and the peak throughput of the selected compute engine.

These features are necessary but insufficient. A stage count of four does not itself state whether four stages are enough to overlap TMA and WGMMA, whether the associated buffers fit in shared memory, or whether the problem has enough K-iterations to amortize pipeline fill. The derived groups make these consequences explicit.

C.2 Work Decomposition and Wave Quantization

Let (T_M, T_N, T_K) be the candidate tile shape. The output grid and number of mainloop iterations are

$$n_M = \left\lceil \frac{M}{T_M} \right\rceil, \qquad n_N = \left\lceil \frac{N}{T_N} \right\rceil, \qquad n_K = \left\lceil \frac{K}{T_K} \right\rceil.$$

The candidate computes $n_M n_N$ output tiles. The edge tiles may execute masked work that is discarded; we encode this separately along each output dimension:

$$w_M = \frac{n_M T_M - M}{M}, \qquad w_N = \frac{n_N T_N - N}{N}.$$

To estimate wave quantization, we divide the number of output tiles by the number of concurrently resident blocks across the GPU. If S_{SM} is the SM count and b_{res} the estimated resident blocks per SM, then

$$q_{\mathrm{SM}} = \left\lceil \frac{n_M n_N}{S_{\mathrm{SM}} b_{\mathrm{res}}} \right\rceil$$

is the number of passes that the resident blocks make over the tile list. The corresponding wave efficiency is

$$\eta_{\mathrm{last}} = \frac{n_M n_N}{q_{\mathrm{SM}} S_{\mathrm{SM}} b_{\mathrm{res}}}.$$

A value near one indicates that there are several waves and/or the final wave fills the GPU, while a small value indicates a ragged tail. This directly motivates the Stream-K proxy:

$$\texttt{streamK_applicability} = \max(0, 1 - \eta_{\text{last}}),$$

together with a feature indicating whether the candidate actually uses Stream-K. Cluster features similarly encode the cluster dimensions, number of tiles available to fill each cluster dimension, and whether the cluster overshoots the grid.

C.3 Memory Behavior

We estimate problem arithmetic intensity using the bytes required by the input matrices and the complete epilogue:

$$I_{\text{problem}} = \frac{2MNK}{b_A MK + b_B KN + Q_{\text{epilogue}}},$$

where b_A and b_B are input element sizes and Q_{epilogue} includes output stores and any reads or writes induced by the output type, βC, residual, bias, activation, or conversion operations.

A complementary tile-level feature captures the amount of tensor-core work obtained from a loaded operand footprint:

$$I_{\text{mainloop}} = \frac{2T_M T_N}{T_M b_A + T_N b_B}.$$

Here T_K cancels. This is useful: T_K affects per-stage storage and pipeline depth, whereas T_M and T_N determine how much reuse the tile geometry obtains from each loaded operand row or column.

We estimate the re-streaming incurred if operand panels cannot remain resident in LLC:

$$r_{\text{stream}} = \frac{n_N b_A MK + n_M b_B KN}{b_A MK + b_B KN}.$$

The remaining cache features compare the complete operand working set and a single resident operand panel with LLC capacity. Together, these features distinguish a problem whose repeated operand reads are absorbed by LLC from one that must repeatedly fetch them from HBM.

C.4 Resource Pressure and Occupancy

The shared-memory buffer required for one mainloop stage is

$$B_{\text{stage}} = T_M T_K b_A + T_N T_K b_B.$$

For S pipeline stages, the mainloop requires SB_{stage} bytes. The epilogue additionally requires schedule-dependent storage B_{epi}, including output fragments and any fused epilogue state. For a warp-specialized schedule, mainloop and epilogue storage may overlap; cooperative and ping-pong schedules may require both simultaneously. We therefore compute the corresponding schedule-specific total shared-memory requirement B_{smem} and its fraction of per-SM capacity:

$$f_{\text{smem}} = \frac{B_{\text{smem}}}{C_{\text{smem}}}.$$

This bounds shared-memory-limited residency:

$$b_{\text{smem}} = \left\lfloor \frac{C_{\text{smem}}}{B_{\text{smem}}} \right\rfloor.$$

We approximate register pressure from the output elements accumulated per thread,

$$r_{\text{proxy}} = \frac{T_M T_N}{T_{\text{block}}},$$

and use the register-file capacity C_{reg} to derive an independent residency bound b_{reg}. The estimated resident-block count is

$$b_{\text{res}} = \min(b_{\text{smem}}, b_{\text{reg}}, b_{\text{arch}}).$$

Keeping the separate bounds matters, because increasing the stage count tightens the shared-memory bound but does not directly alter accumulator pressure. The model can therefore distinguish a kernel limited by registers from one that falls off a shared-memory size limitation.

C.5 Pipeline Balance

Hopper warp-specialized kernels overlap TMA loads from HBM to shared memory with WGMMA execution from shared memory. We estimate the producer and consumer times for one stage as

$$t_{\text{producer}} = \frac{n_{SM} B_{\text{stage}}}{\widehat{BW}_{\text{TMA}}} \qquad t_{\text{consumer}} = \frac{2 T_M T_N T_K}{n_{\text{consumer}} \widehat{P}_{\text{WGMMA}}}$$

where n_{consumer} is the number of consumer warpgroups actively executing the mainloop for the selected schedule, $\widehat{BW}_{\text{TMA}}$ is the bandwidth of the TMA operation, and $\widehat{P}_{\text{WGMMA}}$ is the throughput of the WGMMA operation. The producer-consumer ratio

$$r_{\text{pc}} = \frac{t_{\text{consumer}}}{t_{\text{producer}}}$$

is a static proxy for whether TMA can supply WGMMA at the required rate. Values far below one indicate that consumers are likely to wait for data; values far above one indicate that loads are completed faster than arithmetic consumes them.

Two additional features describe whether the pipeline depth is useful for the problem:

$$f_{\text{fill}} = \frac{\min(S, n_K)}{S}, \qquad a_{\text{stage}} = \frac{n_K}{S}.$$

The first falls below one when the contraction is too shallow to prime a deep pipeline. The second measures how often the pipeline fill cost is amortized. Finally, $S \cdot f_{\text{smem}}$ exposes the interaction between stage count and shared-memory pressure. This interaction captures both the stage-count cliff and the regime in which a larger T_K becomes harmful because its enlarged stage buffers force a shallower pipeline.

D Implementation Optimizations

To make the dataset practical to collect, we implement the following optimizations:

1. **Batched compilation.** Kernels that share a subset of configurations are compiled together into one shared library. A single exported entry point dispatches by kernel index inside the batch, amortizing template instantiation and `nvcc` startup cost across many configurations.
2. **Executable caching.** Each batch library is stored under a name derived from a hash of its member configurations, and kernels are reused instead of recompilation.
3. **Buffer caching.** Each GPU worker pre-allocates a fixed device memory pool and serves all kernels from slices of that pool rather than allocating fresh tensors per kernel. We replicate the matrices enough times from this pool until the working set exceeds L2 capacity.
4. **Node-local database storage.** Benchmark results and compile metadata are written to a registry database kept in node-local memory for the duration of the job, and checkpointed periodically to scratch and to durable home storage.

Table 8: Held-out evaluation shapes and valid kernel counts per layout.

Shape			Kernel count				
M	N	K	TN	TT	NN	NT	Total
2048	2048	2048	60,795	60,740	60,740	60,696	242,971
4096	4096	4096	60,795	60,740	60,740	60,696	242,971
64	64	64	32,190	32,135	32,135	32,091	128,551
128	128	128	60,795	60,740	60,740	60,696	242,971
256	256	256	60,795	60,740	60,740	60,696	242,971
512	512	512	60,795	60,740	60,740	60,696	242,971
32	128	4096	60,795	60,740	60,740	60,696	242,971
2048	128	4096	60,795	60,740	60,740	60,696	242,971
4096	128	4096	60,795	60,740	60,740	60,696	242,971
12,288	128	4096	60,795	60,740	60,740	60,696	242,971
256	4096	4096	60,795	60,740	60,740	60,696	242,971
256	12,288	4096	60,795	60,740	60,740	60,696	242,971
64	12,288	4096	37,290	37,235	37,235	37,191	148,951
2048	2048	128	60,795	60,740	60,740	60,696	242,971
4096	11,008	4096	60,795	60,740	60,740	60,696	242,971
256	256	8192	60,795	60,740	60,740	60,696	242,971
512	3072	768	60,795	60,740	60,740	60,696	242,971
Sum (68 groups)			981,405	980,470	980,470	979,722	3,922,067

5. **Parallel compilation and benchmarking.** Compilation and GPU benchmarking run concurrently: a pool of CPU workers builds shared libraries while one persistent process per GPU consumes a queue of newly compiled kernels. A monitor recovers from worker crashes and hung kernels by attributing the failure to the active measurement, recording it, and respawning the worker to continue with the remainder of the queue.

E Evaluation Dataset

Unlike the training corpus, where each problem is measured on a proxy shortlist of $\approx 2{,}000$ configurations, the evaluation corpus is *exhaustively* enumerated: for every held-out shape (M, N, K) in Table 8 and every layout tag $\ell \in \{\mathrm{TN}, \mathrm{TT}, \mathrm{NN}, \mathrm{NT}\}$, the candidate set $\mathcal{C}(M, N, K, \ell)$ is the full catalogue after static validity filtering (Section A) and shape-aware plan guards. No further proxy pruning is applied. The 17 shapes are listed exhaustively; together with four layouts they define 68 evaluation groups and 3,922,067 distinct (kernel, shape) benchmark tasks.

F Model Design

F.1 Problem and Candidate Sets

A GEMM instance is a tuple $p = (M, N, K, \ell)$, where ℓ denotes the operand layout tag. h be a hardware descriptor. For each p we maintain a *finite candidate set* $\mathcal{C}(p) \subset \mathcal{K}$, where $\mathcal{K}$ is the statically valid CUTLASS configuration catalogue (Section A). Every $c \in \mathcal{C}(p)$ is described by a feature vector $\mathbf{x}(p, c) = \phi(p, c) \in \mathbb{R}^d$ computed from problem geometry, tile and cluster parameters, schedule choices, and the analytical hardware proxies of Section 3; no feature depends on measured runtime counters. A learned selector is a scoring function $f_\theta : \mathbb{R}^d \to \mathbb{R}$. The deployed decision rule is always

$$\hat{c}(p) \;=\; \arg\max_{c \in \mathcal{C}(p)} f_\theta\big(\mathbf{x}(p, c, h)\big), \tag{1}$$

i.e. we reduce selection to within-problem ranking and do not use a predicted throughput value directly at inference time.

F.2 Measured Labels

For each successfully benchmarked pair (p, c) the autotuner records a mean throughput estimate $\widehat{T}(p, c)$ and an empirical standard deviation $\widehat{\sigma}(p, c)$ over repeated timed rounds. Group all rows that share the same p and index such a group by g. The *empirical group maximum* is

$$T_g^\star \;=\; \max_{c \in \mathcal{C}(g)} \widehat{T}(g, c), \qquad y(g, c) \;=\; \frac{\widehat{T}(g, c)}{T_g^\star} \in (0, 1], \tag{2}$$

so that $y(g, c) = 1$ for at least one measured candidate in the group and $y(g, c) < 1$ otherwise. On training problems $\mathcal{C}(g)$ is a proxy shortlist (Section 4.1); on evaluation problems it is a near-exhaustive sweep over the catalogue, so $T_g^\star$ approximates the in-space empirical oracle for that shape–layout.

F.3 Relevance grades (ranking supervision)

Ranking objectives do not use raw $\widehat{T}$ directly; they use discrete grades derived from it. Within group g, sort candidates by decreasing $\widehat{T}(g, c)$ and partition the sorted list into contiguous *bands* by merging adjacent entries i and $i{+}1$ whenever their throughput gap is smaller than twice the pooled measurement uncertainty,

$$\widehat{T}_i - \widehat{T}_{i+1} < 2\sqrt{\widehat{\sigma}_i^2 + \widehat{\sigma}_{i+1}^2}. \tag{3}$$

Let $b(c) \in \{0, 1, \ldots\}$ be the band index of c (0 for the top band). We assign integer grades $\gamma(g, c) = G_{\max} - b(c)$ with $G_{\max} = 31$. For listwise tree training we further compress grades to a top-focused relevance scale with B bands ($B{=}2$ by default):

$$\mathrm{rel}(g, c) \;=\; \mathrm{clip}\big(\gamma(g, c) - (G_{\max} - B),\, 0,\, B\big). \tag{4}$$

Thus only the highest-throughput band(s) carry non-zero relevance, concentrating learning on near-optimal orderings.

F.4 Model A (gradient-boosted ranker)

Model A is an additive ensemble of regression trees $f_\theta = \sum_m h_m$ fitted with XGBoost. Numeric features enter splits directly; categorical schedule and layout fields use native categorical handling with a fixed level ordering shared across train, validation, and deployment. Let $\mathcal{G}_{\mathrm{tr}}$ denote the set of training groups and list all rows in a group-contiguous order with group-id vector $\mathbf{g}$. We train with one of three objectives:

- **Listwise ranking** (default): minimise a surrogate for NDCG at cut-off $|\mathcal{C}(g)|$ using targets $\mathrm{rel}(g, c)$ and the standard $\mathrm{rank}:\mathrm{ndcg}$ objective;
- **Pairwise ranking**: $\mathrm{rank}:\mathrm{pairwise}$ on the same relevance labels;
- **Pointwise regression** (ablation): minimise $\sum_i \big(f_\theta(\mathbf{x}_i) - y_i\big)^2$ with $\mathrm{reg}:\mathrm{squarederror}$, treating y_i as a continuous target independent of group structure.

Only the first two objectives align the training loss with the argmax decision rule in equation 1; the regression ablation tests whether accurate throughput prediction implies good selection.

F.5 Model B (feed-forward ranker)

Model B is a multilayer perceptron $f_\theta : \mathbb{R}^{d'} \to \mathbb{R}$ applied to a transformed feature matrix. Let $\mathbf{x}^{\mathrm{num}}$ denote numeric columns and $\mathbf{x}^{\mathrm{cat}}$ categorical indicators obtained by one-hot encoding with the same frozen level sets as in Model A. Training applies per-column standardisation to $\mathbf{x}^{\mathrm{num}}$ (zero mean, unit variance, with $\mathrm{NaN}/\pm\infty$ mapped to zero before scaling); categoricals are left unscaled.

We consider three training losses:

$$\mathcal{L}_{\mathrm{MSE}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\big(f_\theta(\mathbf{x}'_i) - y_i\big)^2, \tag{5}$$

$$\mathcal{L}_{\mathrm{RankNet}}(\theta) = \frac{1}{|\mathcal{P}|}\sum_{(i,j)\in\mathcal{P}} \log\big(1+\exp\big(f_\theta(\mathbf{x}'_j) - f_\theta(\mathbf{x}'_i)\big)\big), \tag{6}$$

$$\mathcal{L}_{\mathrm{LambdaRank}}(\theta) = \frac{1}{|\mathcal{P}|}\sum_{(i,j)\in\mathcal{P}} \Delta\mathrm{NDCG}_{ij}\ \log\big(1+\exp\big(f_\theta(\mathbf{x}'_j) - f_\theta(\mathbf{x}'_i)\big)\big), \tag{7}$$

where $\mathbf{x}'_i$ is the scaled/encoded feature vector, and $\mathcal{P}$ is a set of ordered pairs within a group such that $y_i > y_j$ (higher-throughput index i, lower j). For LambdaRank we use gain values $g_i = y_i$ (linear mode) or $g_i = 2^{y_i} - 1$ (exponential mode), ideal discounted cumulative gain $\mathrm{IDCG} = \sum_{r=1}^{n} g_{\pi(r)}/\log_2(1+r)$ for the throughput-sorted permutation π, and pair weights

$$\Delta\mathrm{NDCG}_{ij} \;=\; \frac{\big|g_i - g_j\big|\,\big|D(r_i) - D(r_j)\big|}{\mathrm{IDCG}}, \qquad D(r) = \frac{1}{\log_2(1+r)}, \tag{8}$$

with ranks r_i, r_j computed from the current model scores (detached from the gradient graph when evaluating D). Pairs are sampled per group with a hard cap; pairs touching any of the top eight y-values are always retained (*needle guarantee*) before random subsampling of the remainder. At deployment, numeric standardisation is folded into the exported inference graph so that equation 1 can be evaluated on raw feature rows.

F.6 Linear Ridge Baseline

The ridge baseline tests how much of the learned selectors' advantage comes from the hardware-aware representation and *globally linear* feature combinations, without tree splits or hidden layers. It uses the same candidate feature matrix as Models A and B: numeric columns from $\phi(p, c, h)$ are standardised with training-set mean and variance (with non-finite values mapped to zero before scaling), and categorical schedule and layout fields are one-hot encoded with the same frozen level sets. Let $\mathbf{x}'_i$ denote the resulting vector for row i.

The scorer is ordinary least squares with optional ℓ_2 shrinkage:

$$f_{\mathbf{w},b}(\mathbf{x}') \;=\; \mathbf{w}^\top\mathbf{x}' + b, \qquad \min_{\mathbf{w},b}\ \sum_{i\in\mathcal{P}_{\mathrm{tr}}}\big(y_i - f_{\mathbf{w},b}(\mathbf{x}'_i)\big)^2 + \alpha\,\|\mathbf{w}\|_2^2. \tag{9}$$

We solve equation 9 in closed form after centring $\mathbf{x}'$ and y (intercept handled separately); at inference the deployed rule is still the argmax in equation 1. Unlike the ranking objectives above, training optimises squared error on y directly; validation and test nonetheless score the *selected* kernel through regret equation 10.

F.7 Selection Regret (The Target Metric)

For any scorer f_θ the *regret* of group g is

$$R(g; f_\theta) \;=\; 1 - y\big(g, \hat{c}(g)\big) \;=\; 1 - \frac{\widehat{T}\big(g, \hat{c}(g)\big)}{T^\star_g}, \qquad \hat{c}(g) = \arg\max_{c\in\mathcal{C}(g)} f_\theta\big(\mathbf{x}(g,c)\big). \tag{10}$$

$R(g) \in [0, 1)$ with $R(g) = 0$ iff the selector picks a measured maximiser in $\mathcal{C}(g)$. Aggregate reports include the mean $\bar{R} = \frac{1}{|\mathcal{G}|}\sum_g R(g)$, median, 95th percentile, and maximum across groups, as well as

$$\mathrm{Top1}(f_\theta) = \frac{1}{|\mathcal{G}|}\sum_g \mathbf{1}\big[\mathrm{rank}(\hat{c}(g)) = 1\big], \tag{11}$$

$$\mathrm{Top5}(f_\theta) = \frac{1}{|\mathcal{G}|}\sum_g \mathbf{1}\big[\mathrm{rank}(\hat{c}(g)) \le 5\big], \tag{12}$$

$$\mathrm{Within5\%}(f_\theta) = \frac{1}{|\mathcal{G}|}\sum_g \mathbf{1}\big[R(g; f_\theta) \le 0.05\big], \tag{13}$$

Table 9: Hyperparameter search space for Model A (gradient-boosted trees).

Knob	Range / values	Notes
η (learning rate)	$[5 \times 10^{-3}, 0.3]$ log-uniform	
max_depth	$\{4, \dots, 12\}$	
min_child_weight	$[1, 30]$ log-uniform	
subsample	$[0.5, 1.0]$	row subsampling
colsample_bytree	$[0.5, 1.0]$	column subsampling
λ (L2)	$[0.01, 20]$ log-uniform	
α (L1)	$[10^{-4}, 10]$ log-uniform	
γ (min split loss)	$[0, 5]$	
$n_{\text{estimators}}$	$\{300, \dots, 1200\}$	boosting rounds
loss	$\{\text{ndcg}, \text{pairwise}, \text{mse}\}$	optional categorical
rel_top_bands	$\{2, \dots, 5\}$	ranking losses only
CV folds K	5	shape-grouped

Table 10: Hyperparameter search space for Model B (MLP).

Knob	Range / values	Notes
loss	$\{\text{LambdaRank}, \text{RankNet}, \text{mse}\}$	optional categorical
learning rate	$[10^{-4}, 5 \times 10^{-3}]$ log-uniform	Adam
hidden topology	10 preset tuples	depths 2–4, widths 32–1024
dropout	$[0, 0.4]$	after each ReLU
weight decay	$\{0, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}\}$	
epochs	$\{20, \dots, 150\}$	
batch size	$\{1024, 2048, 4096, 8192\}$	MSE only
groups per batch	$\{32, 64, 128, 256\}$	ranking only
gradient clip (max norm)	$\{0, 0.5, 1, 5, 10\}$	ranking only
max pairs per group	$\{2\text{k}, \dots, 64\text{k}\}$	ranking only
gain transform	$\{\text{linear}, \text{exp}\}$	LambdaRank only
CV folds K	3 during search	shape-grouped
train seeds	2 per fold	averaged before reporting CV
pruner	median, 10 startup trials	early-stops weak MLP trials

where $\text{rank}(\cdot)$ is the rank by measured $\widehat{T}$ within the group (minimum-rank tie breaking). Regret is the quantity we minimise during hyperparameter search and of which we maximise interpretability.

F.8 Hyperparameter Search

We tune θ with Optuna's Tree-structured Parzen Estimator (TPE), minimising CV in equation 14. Unless an objective is pinned for an ablation arm, each trial also selects the training loss (ranking versus pointwise regression) as a categorical hyperparameter so TPE can allocate budget to the better-surrogate family. Trials are executed in parallel on multiple GPUs against a shared SQLite study so workers contribute to one global search. Table 9 and Table 10 list the searched ranges; conditional knobs apply only when the trial's loss is a ranking objective.

For Model B the search budget is deliberately lighter per trial (3-fold $\times$ two seeds); the winning configuration is re-validated offline at 5-fold cross-validation with three independent initialisation seeds before the final production fit. After search completes, the best trial's hyperparameters are *locked* and the model is refit once on all of $\mathcal{P}_{\text{tr}}$.

F.9 Shape-Grouped Cross-Validation

Hyperparameters are chosen by K-fold cross-validation on $\mathcal{P}_{\text{tr}}$ only. Folds partition *base shapes* (M, N, K): all four layouts of a shape belong to the same fold, so validation never sees a training shape under any layout. Let $\mathcal{S}$ be the set of distinct base shapes in $\mathcal{P}_{\text{tr}}$ and $\{\mathcal{S}^{(k)}\}_{k=1}^{K}$ a partition of

$\mathcal{S}$. For fold k, train $f_\theta^{(k)}$ on all groups whose base shape lies in $\mathcal{S} \setminus \mathcal{S}^{(k)}$ using the chosen training objective ($\mathcal{L}_{\mathrm{MSE}}$, $\mathcal{L}_{\mathrm{RankNet}}$, $\mathcal{L}_{\mathrm{LambdaRank}}$, or the tree ranker analogue), then evaluate on validation groups $\mathcal{G}^{(k)}$ by computing equation 10 via equation 1. The CV score is

$$\mathrm{CV}(f_\theta) \;=\; \frac{1}{K}\sum_{k=1}^{K} \frac{1}{|\mathcal{G}^{(k)}|} \sum_{g \in \mathcal{G}^{(k)}} R\big(g; f_\theta^{(k)}\big). \tag{14}$$

Bayesian optimisation (Optuna TPE) minimises CV over tree depth, learning rate, regularisation, ensemble size, hidden widths, dropout, pair caps, and—when not fixed—the choice among ranking versus regression objectives. $\mathcal{P}_{\mathrm{ev}}$ is never used in equation 14.

F.10 Final Fit and Test Protocol

After hyperparameters are fixed, we refit f_θ once on *all* training groups in $\mathcal{P}_{\mathrm{tr}}$ (no held-out fold). The reported generalisation result is the evaluation of equation 10–equation 1 on the 68 oracle groups in $\mathcal{P}_{\mathrm{ev}}$, with breakdowns by problem regime (square, tall, wide, skinny-K, large). Baselines computed on the same measured data include uniform random selection, whose expected regret is $\mathbb{E}_g[1-\bar{y}(g)]$ with $\bar{y}(g) = \frac{1}{|\mathcal{C}(g)|}\sum_c y(g,c)$, and a single fixed configuration chosen by best mean y over groups where it appears in at least 80% of evaluation groups.

F.11 Relation between Training Loss and Validation Metric

Measured throughput always defines the supervision signal (y and γ), but the default training objectives are ranking losses that penalise incorrect *orderings* within $\mathcal{C}(g)$ rather than absolute error in $\widehat{T}$. Cross-validation and final testing, by contrast, always score the *deployed* rule equation 1 through regret equation 10. This deliberate mismatch—optimise surrogate ranking loss, validate selection regret—matches the compiler use case: only the identity of the argmax matters, and mis-ordering among top candidates is far more costly than mis-predicting throughput on clearly suboptimal tiles.

G Sampling Kernels - Analysis

Exhaustive measurement is impractical for training. Therefore we study how the measurement budget should be allocated across candidates. The goal of this experiment is to quantify how effectively a sampling policy exposes high-performing kernels within a fixed measurement budget.

We use the exhaustively measured BF16 evaluation groups as ground truth. For each group g, let C_g denote the complete valid candidate catalogue, $T(c)$ the measured throughput of candidate c, and

$$T_g^\star = \max_{c \in C_g} T(c)$$

the exhaustive oracle throughput. Given a sampling policy that selects a subset $S_B(g) \subseteq C_g$ of B candidates, we define the *sampling regret*

$$R_B(g) = 1 - \frac{\max_{c \in S_B(g)} T(c)}{T_g^\star}.$$

This measures the quality lost purely because the measurement subset failed to contain the best available kernels. For stochastic policies, we repeat each sampling experiment 100 times per group and report mean results.

G.1 Sparsity of the High-Performance Tail

We first quantify how many kernels in the valid catalogue are actually competitive. For each group, we compute the fraction of candidates whose throughput lies within ϵ of the exhaustive oracle:

$$q_\epsilon(g) = \frac{\big|\{c \in C_g : T(c) \geq (1-\epsilon)T_g^\star\}\big|}{|C_g|}.$$

Table 11: Fraction of valid kernels close to the exhaustive oracle.

Threshold	Mean fraction	Median fraction	Minimum fraction
Within 1%	0.43%	0.17%	0.17%
Within 5%	2.41%	1.00%	0.33%
Within 10%	10.91%	4.42%	1.17%

Across the evaluation groups, only 0.43% of candidates are within 1% of the oracle on average, 2.4% are within 5%, and 10.9% are within 10%. The corresponding median fractions are 0.17%, 1.0%, and 4.4%, respectively. In the most selective groups, fewer than 0.33% of valid kernels lie within 5% of the optimum.

These results show that high-performing kernels occupy a sparse tail of the catalogue. A uniformly sampled training subset can therefore contain thousands of measured candidates while still failing to observe any configuration close to the true optimum.

G.2 Sampling Policies

We compare the following candidate-selection policies.

Uniform. Candidates are sampled uniformly at random from the complete valid catalogue.

Static top-B. Candidates are ranked by the final static sampling score from Section 4.1, and the highest-ranked B candidates are selected deterministically.

Biased-diverse sampling. This is the policy used to construct the training set. A fraction 75% of the budget is sampled from the high-ranked portion of the static score, while the remaining 25% is sampled uniformly from the rest of the catalogue. The exploratory quarter is not overhead for oracle discovery alone: the throughput predictor must also observe poorly ranked configurations during training, so a purely heuristic-biased corpus would under-represent the long tail of bad kernels.

We evaluate budgets

$$B \in \{100, 250, 500, 1000, 2000, 3000, 5000, 10000\}.$$

The values 1000, 2000, and 3000 correspond directly to the candidate budgets used in our training collection.

G.3 Sampling Efficiency

Figure 7 reports sampling regret as a function of measurement budget. Uniform sampling improves steadily with larger budgets, but converges slowly because the near-optimal tail is sparse. At a budget of 1000 candidates, uniform sampling leaves 11.3% mean regret; at 2000 candidates the regret is 8.7%, and at 3000 candidates it remains 7.1%.

The static top-B policy improves over uniform sampling at small budgets, reaching 0.6% regret at $B = 2000$. However, its deterministic bias can exclude competitive kernels that receive a poor heuristic score. The biased-diverse policy combines both effects and achieves 1.5%, 0.9%, and 0.5% mean regret at budgets of 1000, 2000, and 3000, respectively.

At the default training budget of 2000 candidates, biased-diverse sampling reduces sampling regret by 89.4% relative to uniform sampling (from 8.5% to 0.9%). Static top-B attains marginally lower regret (0.6%) at this budget, but it is unsuitable as a training policy because it never deliberately measures configurations the heuristic ranks poorly. Oracle-discovery metrics and training-corpus construction therefore pull in opposite directions under a fixed budget.

Figure 7: Sampling regret versus candidate measurement budget on the exhaustive BF16 evaluation groups.

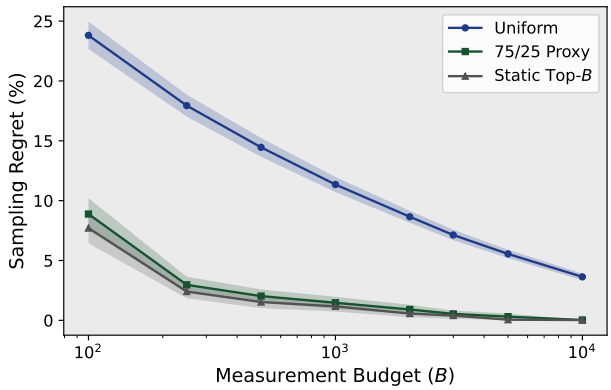
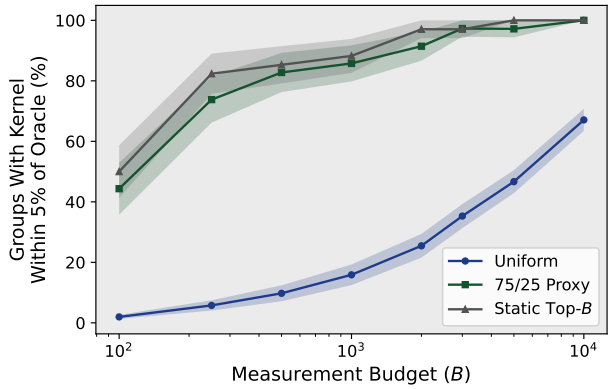


Figure 8: Probability that the sampled candidate pool contains a near-optimal kernel.



G.4 Probability of Observing a Near-Optimal Kernel

Sampling regret summarizes the quality of the best observed configuration, but it is also useful to ask how often a sampling policy discovers at least one near-optimal kernel. We therefore measure

$$P_\epsilon(B) = \Pr\left[\max_{c \in S_B(g)} T(c) \geq (1-\epsilon)T_g^\star\right]$$

for $\epsilon \in \{0.01, 0.05, 0.10\}$.

Figure 8 reports $P_{0.05}(B)$ as a function of measurement budget. At $B = 2000$, uniform sampling observes at least one kernel within 5% of the oracle in 25.6% of groups, compared with 97.1% for static top-B and 91.3% for biased-diverse sampling. At $B = 3000$, the corresponding fractions are 35.3%, 97.1%, and 97.3%. Uniform sampling improves with budget but remains far below the heuristic-biased policies until $B \geq 3000$, where biased-diverse sampling essentially matches static top-B.

G.5 Effect of the Exploration Fraction

The biased-diverse policy trades off exploitation of the static score against broader exploration of the remaining catalogue. We ablate this mixture at a fixed budget of $B = 2000$. Let α denote the fraction of the budget drawn from the heuristic-preferred region. We test

$$\alpha \in \{0, 0.25, 0.50, 0.75, 1.0\}.$$

Table 12: Ablation of the exploitation fraction at $B = 2000$.

Heuristic-biased fraction α	Mean sampling regret	Groups within 5% of oracle
0.00	8.5%	25.6%
0.25	1.4%	87.0%
0.50	1.1%	88.9%
0.75	0.9%	91.3%
1.00	0.6%	97.1%

Pure exploration ($\alpha = 0$) reduces to broad random sampling and yields 8.5% mean regret. Pure exploitation ($\alpha = 1$) achieves 0.6% regret and 97.1% within-5% coverage, the best values *for oracle discovery* on this evaluation set. Increasing α monotonically improves both metrics, yet $\alpha = 1$ is not a viable training policy: the model would rarely see measured examples of low-throughput kernels and could not learn to discriminate bad configurations from good ones.

This retrospective analysis validates the $\alpha = 0.75$ policy used for corpus construction. At $B = 2000$ this yields 0.9% mean regret and 91.3% within-5% coverage—a modest cost relative to $\alpha = 1$—while guaranteeing that one quarter of every measurement budget samples outside the heuristic-preferred region.

G.6 Summary

The exhaustive study confirms that the candidate-selection problem is dominated by a sparse high-performance tail: only 2.4% of valid kernels lie within 5% of the oracle on average. Consequently, uniform sampling requires substantially larger measurement budgets to expose competitive configurations. At the default budget of 2000 candidates per shape–layout group, our biased-diverse policy reduces sampling regret from 8.5% to 0.9% and increases the probability of observing a kernel within 5% of the oracle from 25.6% to 91.3%.

These results validate the candidate-sampling strategy used for the training corpus. Most measurements are directed toward configurations with favorable static efficiency estimates, while the fixed 25% exploratory fraction ensures the predictor is trained on bad as well as good kernels—a requirement that pure top-B or $\alpha = 1$ sampling cannot satisfy under the same budget.

H Additional Training and Evaluation Results

This section provides additional analysis of the model-training objectives, feature ablations, and the held-out exhaustive evaluation discussed in Section 5.

H.1 Training Objective Comparison

We train both the MLP and XGBoost models using three objectives. For the MLP, we compare mean-squared error (MSE), RankNet, and LambdaRank; for XGBoost, we compare squared-error regression, pairwise ranking, and NDCG listwise ranking. Each objective is evaluated with both the full hardware-aware representation and the structural-only feature set.

Figure 9 and Figure 10 show validation NDCG@10 throughout training. All objectives converge stably. With the full feature set, MSE achieves the highest validation NDCG@10 for both MLP (0.968) and XGBoost (0.960). With structural features, MSE again performs best, reaching 0.969 for MLP and 0.929 for XGBoost.

Validation NDCG@10 is not perfectly aligned with held-out selection regret. In particular, MLP RankNet achieves slightly lower validation NDCG than MSE, but attains the lowest mean regret on the exhaustive evaluation set. This suggests that validation ranking quality is useful for monitoring training, but does not fully capture the cost of errors among the highest-performing candidates.

Figure 11 compares the full and structural representations across all objectives. Hardware-aware features consistently reduce held-out regret, although the magnitude of the improvement depends on

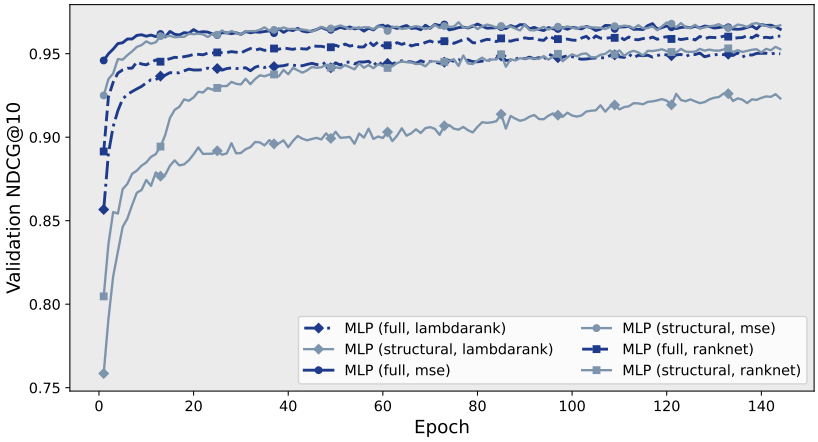


Figure 9: MLP validation NDCG@10.

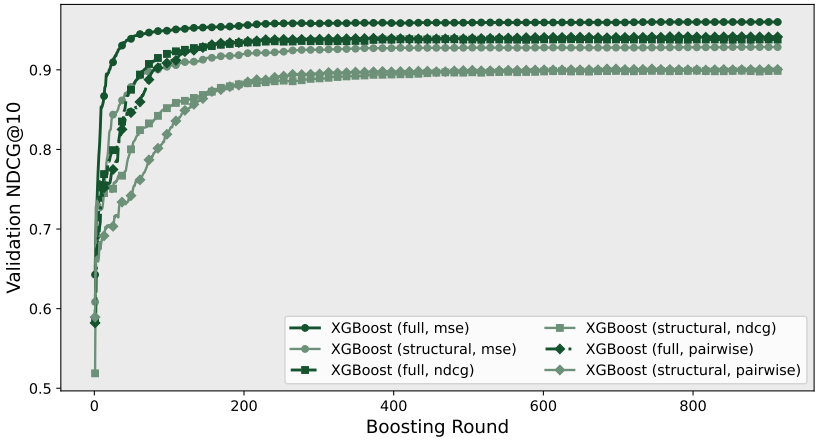


Figure 10: XGBoost validation NDCG@10.

the model family. The gain is relatively modest for the MLP, while XGBoost benefits substantially more from the derived hardware features.

H.2 Detailed Exhaustive Evaluation

Table 13 reports detailed selection quality on the evaluation dataset.

The learned selectors are substantially more likely to recover near-oracle configurations. MLP RankNet selects a kernel within 5% of the oracle on 63.2% of groups, while XGBoost MSE does so on 52.9%. By comparison, the top-1 nvMMH recommendation is within 5% of the oracle on only 10.3% of groups.

We additionally report an oracle over nvMMH's top-K candidate list. Even when selecting the fastest measured candidate from that shortlist, only 25.0% of groups fall within 5% of oracle throughput. This indicates that a substantial fraction of nvMMH's error arises before the final choice among its recommended candidates.

H.3 Why is a learned model necessary?

We next ask whether the hardware-aware representation can support strong kernel selection without a nonlinear learned model. We compare against two simpler alternatives: a validation-selected

Table 13: Selection quality on 68 exhaustively measured shape–layout groups.

Method	Mean	Median	Within 1%	Within 5%	Top-1
MLP (full)	6.2%	2.6%	36.8%	63.2%	27.9%
MLP (structural)	7.4%	3.1%	32.4%	55.9%	26.5%
XGBoost (full)	6.4%	4.9%	25.0%	52.9%	19.1%
XGBoost (structural)	10.7%	8.3%	17.6%	39.7%	13.2%
nvMMH (best-of-K)	12.0%	10.2%	0.0%	25.0%	0.0%
nvMMH (top-1)	17.3%	14.6%	0.0%	10.3%	0.0%
Ridge (full)	23.7%	20.0%	4.4%	16.2%	1.5%
Ridge (structural)	60.8%	63.6%	0.0%	0.0%	0.0%
Analytical score	63.7%	63.8%	0.0%	0.0%	0.0%
Random	71.5%	—	—	—	—

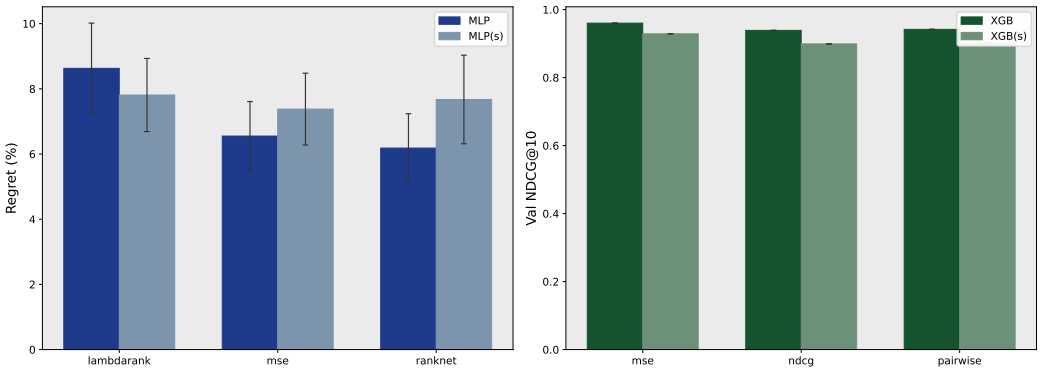


Figure 11: Comparison of full hardware-aware and structural-only features across training objectives.

analytical score combining a small set of mechanistic hardware terms, and a ridge regressor that linearly scores candidates using the same full or structural feature representations as the learned selectors (Table 13).

The analytical score performs poorly, reaching 63.7% mean regret and never selecting a configuration within 5% of the oracle across the 68 exhaustive evaluation groups. This is despite several of its constituent hardware-aware features showing clear within-group correlation with throughput on the training set. Thus, identifying individually informative mechanisms is not sufficient: a compact fixed composition of those mechanisms does not recover the high-performance tail of the configuration space.

The ridge baseline isolates the value of the representation under a fixed linear model family. Using structural features alone yields 60.8% mean regret, while adding the hardware-aware features reduces regret to 23.7%. This large improvement shows that the proposed representation exposes substantial selection signal even to a simple linear predictor. Nevertheless, the linear hardware-aware model remains worse than nvMMH (17.3%) and far behind the nonlinear MLP and XGBoost selectors (6.2% and 6.4%).

Together, these results separate representation quality from model expressivity. The hardware-aware features make the ranking problem substantially easier, but a single global linear mapping is insufficient for strong selection. The remaining gap indicates that accurate ranking depends on nonlinear and problem-dependent interactions between work decomposition, memory behavior, resource pressure, and pipeline behavior, which the MLP and XGBoost models are able to capture.

H.4 Near-oracle selection rate.

Figure 12 reports the fraction of evaluation groups for which the selected kernel is within 5% of the exhaustive oracle. MLP RankNet with hardware-aware features reaches this threshold on 63.2%

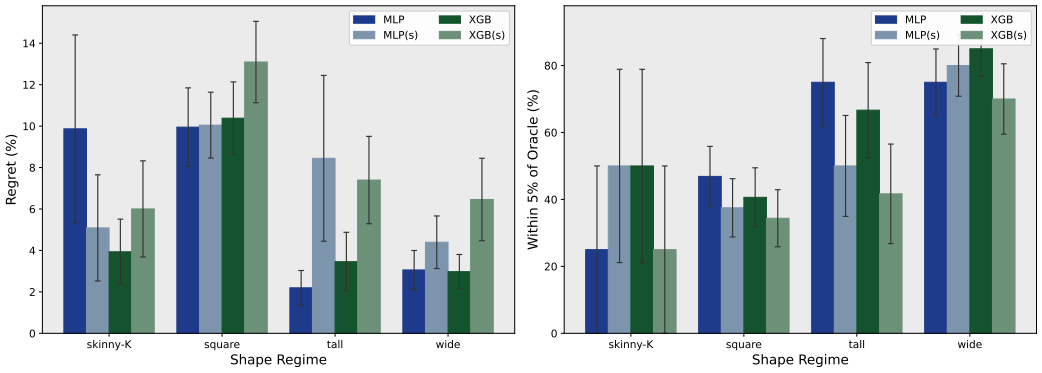


Figure 13: Selection regret and within-5% rate by regime for the best full and structural variants of each model family.

Table 14: Broad GEMM evaluation on 8,000 held-out problems.

Model	Features	Geo-mean [Tflop/s]	Geo-mean % roof	vs. nvMMH	Win %
MLP MSE	Full	42.9	11.9%	1.14×	79.6%
MLP MSE	Structural only	41.2	11.5%	1.10×	73.9%
XGBoost MSE	Full	42.4	11.8%	1.13×	77.4%
XGBoost MSE	Structural only	41.0	11.4%	1.09×	70.1%
nvMMH	—	37.6	10.5%	—	—

of groups, compared with 55.9% for the structural MLP, 52.9% for hardware-aware XGBoost, and 39.7% for structural XGBoost. nvMMH reaches the same threshold on only 10.3% of groups with its top-1 recommendation, or 25.0% when selecting the best measured kernel from its top-K shortlist. This metric complements mean regret by showing how frequently each selector returns a practically near-optimal configuration rather than relying on a small number of large or small errors.

H.5 Performance by GEMM Regime

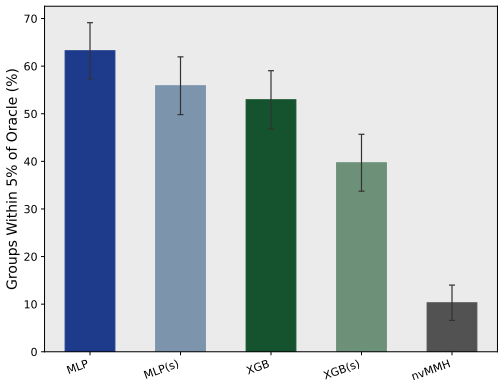


Figure 12: Fraction of exhaustive evaluation groups selected within 5% of oracle throughput.

Selection quality depends on problem geometry. nvMMH performs worse across all geometric regimes, with mean regret between approximately 12% and 19%. The same qualitative pattern appears for both full and structural feature sets, although the hardware-aware representation consistently reduces regret.

Figure 13 compares the best full and structural variants of each model family. The hardware-aware representation improves both mean regret and the fraction of near-oracle selections, with the largest gain for XGBoost.

H.6 Broad GEMM Evaluation Details

The broad evaluation uses 2,000 (M, N, K) triples held out from the production BF16 autotuning databases and evaluates each under all four operand layouts (TN, TT, NN, NT), yielding 8,000 problems. Half of the shapes are fully 32-aligned; the remainder are 8-aligned but not 32-aligned. The latter set tests generalization to off-grid dimensions that are largely absent from the regular training sweep. The performance results are in Table 14.

We compare four learned selectors—MLP and XGBoost, each with the full hardware-aware representation and the structural-only ablation—against nvMMH. Each learned model returns a single rank-1 CUTLASS configuration. nvMMH emits a rank-1 tile and cluster recommendation but does not completely specify the CUTLASS mainloop, epilogue, and scheduler configuration. We therefore materialize all valid variants consistent with the nvMMH recommendation and report the best measured variant. This gives nvMMH an optimistic comparison relative to the learned selectors, which each receive only one selected kernel.

All candidates are compiled and benchmarked on the same GH200 node using the same warmup, iteration-count, and operand-rotation protocol as the training measurements. To isolate kernel ranking from tile-scheduler choices, every method uses the same rank-1 rasterization, swizzle, and split-K settings during benchmarking. nvMMH retains its recommended tile geometry, while the learned selectors retain their independently selected tile, schedule, cluster, and pipeline configuration.

For the roofline analysis in Figure 4a, we use a GH200 peak BF16 tensor-core throughput of $989.5\,\mathrm{Tflop/s}$ and memory bandwidth of $4\,\mathrm{TB/s}$. Arithmetic intensity is computed from the mathematical GEMM work and minimum operand traffic. Each light point corresponds to one measured GEMM, while the bold curves show binned median throughput.

All methods achieve 100% execution coverage on the $8{,}000$ problems. The hardware-aware MLP reaches $42.9\,\mathrm{Tflop/s}$ geometric-mean throughput compared with $37.6\,\mathrm{Tflop/s}$ for nvMMH, corresponding to a $1.14\times$ improvement. Hardware-aware XGBoost reaches $42.4\,\mathrm{Tflop/s}$, or $1.13\times$ nvMMH. The structural variants remain competitive at 41.2 and $41.0\,\mathrm{Tflop/s}$, but both are consistently below their hardware-aware counterparts.

Measured per problem, the full-feature MLP outperforms nvMMH on 79.6% of cases and full-feature XGBoost on 77.4%. Even the structural variants win on 73.9% and 70.1%, respectively. Together with the exhaustive-oracle evaluation, this shows that the learned selectors' lower selection regret translates directly into higher realized throughput over a substantially broader problem distribution.

H.7 Transfer Learning Details

For each target dtype, (FP32 and FP8 E4M3), we construct nested subsets containing $\{1, 5, 10, 25, 50, 100\}\%$ of the 593 target-dtype base training shapes. The same (M, N, K) subsets are used for FP32 and FP8, while the measured candidate kernels differ according to dtype-specific validity and compilation behavior. The fractions therefore measure target-domain shape coverage rather than equal numbers of kernel measurements. Table 15 reports aggregate statistics at 5% of the training shapes (30 of 593 base shapes).

We reuse the model architectures selected for the BF16 study. MLP transfer initializes from the corresponding BF16 checkpoint, while XGBoost is warm-started from the BF16 model by adding 200 trees. We evaluate both the complete hardware-aware representation and the structural ablation. Evaluation uses a disjoint held-out set and the same benchmarking protocol as Section 5.3. Learned selectors emit one rank-1 kernel, while nvMMH is evaluated using its best valid schedule realization.

At full target-dtype coverage, FP32 transfer reaches $1.24\times$ geometric-mean speedup for both hardware-aware MLP and XGBoost, with win rates of 74.6% and 78.4%, respectively. Structural variants remain strong but consistently lower, reaching $1.18\times$ and $1.20\times$. FP8 remains more challenging, particularly in execution coverage, but the hardware-aware MLP and XGBoost still achieve $1.19\times$ and $1.17\times$ geometric-mean speedup over nvMMH.

The low-data regime is particularly informative. At 5% of the target shape pool, the hardware-aware FP32 models already exceed nvMMH by roughly 20%–22%, despite observing only 30 target-dtype GEMM geometries. The corresponding FP8 models also exceed nvMMH, demonstrating that much of the useful structure learned from BF16 survives the precision change and can be adapted with comparatively little target-domain data.

Fusion transfer. We apply the same finetuning protocol to FP16 epilogue-fusion GEMMs. Selectors are warm-started from the unfused BF16 checkpoints and finetuned on nested subsets of the 593-shape fusion training corpus, using the same $\{1, 5, 10, 25, 50, 100\}\%$ fractions as the precision-

Table 15: Cross-precision transfer at 5% of the target training shapes.

Dtype	Method	Coverage	Geo-mean speedup	Win %
FP32	MLP (full)	77.1%	1.20×	58.7%
	XGBoost (full)	97.4%	1.22×	76.3%
	MLP (struct.)	80.8%	1.19×	62.3%
	XGBoost (struct.)	98.8%	1.16×	69.4%
FP8 E4M3	MLP (full)	100.0%	1.12×	69.8%
	XGBoost (full)	68.2%	1.17×	47.3%
	MLP (struct.)	100.0%	1.12×	69.7%
	XGBoost (struct.)	71.2%	1.11×	40.7%

Table 16: DeepBench throughput speedup vs. nvMMH.

Model	Features	vs. nvMMH	Win %
MLP MSE	Full	1.12×	96.0%
MLP MSE	Structural only	1.13×	92.7%
XGBoost MSE	Full	1.12×	84.8%
XGBoost MSE	Structural only	1.05×	59.6%
Ridge	Full	0.83×	40.4%
Ridge	Structural only	0.27×	0.0%
Random	—	0.22×	0.0%
nvMMH	—	—	—

Table 17: DeepBench speedup vs. nvMMH by catalogue split.

Split	n	MLP (full)	XGBoost (full)
Training	107	1.09×	1.07×
Inference server	39	1.25×	1.29×
Inference device	5	1.08×	1.04×

transfer study. We evaluate $1{,}000$ held-out problems with in-vocabulary fusion kinds assigned at evaluation time.

At full fusion training coverage, hardware-aware XGBoost matches nvMMH ($1.00\times$ geometric-mean speedup), while the hardware-aware MLP reaches $0.97\times$. Structural variants remain below at $0.94\times$. All configurations achieve 100% execution coverage.

The low-data fusion regime mirrors the precision-transfer trend. With only 10% of the fusion corpus (59 shapes), hardware-aware MLP and XGBoost already reach $1.00\times$ geometric-mean speedup over nvMMH, while structural models remain lower. The hardware-aware MLP peaks at $1.01\times$ with 25% coverage before settling near parity.

H.8 DeepBench Study Details

Our catalogue contains 151 unique (M, N, K, layout) problems after deduplication and constraining $M, N, K \geq 32$. The evaluation set contains 107 training, 39 inference-server, and 5 inference-device problems. Table 16 summarizes the results. All learned selectors achieve 100% execution coverage on the retained problems. Hardware-aware MLP and XGBoost improve geometric-mean throughput by 12%–13% over nvMMH and win on 85%–96% of problems. The structural MLP matches the full model ($1.13\times$), consistent with the observation that very large MLP models with structural features extrapolate rather robustly. Ridge and random baselines remain well below nvMMH, confirming that the gains require learned nonlinear ranking. The improvement is largest on inference-server GEMMs ($1.24\times$–$1.29\times$ for MLP and XGBoost over nvMMH), which contain the batched speech and language-modeling layers DeepBench was designed to stress. Training-split problems show a smaller but still positive margin ($\approx 1.07\times$–$1.09\times$).