Contour: Extensible Mapping Surfaces for Agentic Accelerator Kernel Generation

Ruijie Gao*, Jirong Yang†, Mengtian Hong‡, Youyou Lu*, Barry Lyu*, Haoran Jin*, and Nathan Bleier*
*University of Michigan †The University of Texas at Austin ‡University of Glasgow
*{ruijieg,yoyolu,barrylyu,allenjin,nbleier}@umich.edu †polarisyjr@utexas.edu ‡2960081H@student.gla.ac.uk

Abstract

Emerging accelerators rarely ship with mature compiler backends, so recent work asks LLM agents to write native kernels directly from ISA documentation and simulator feedback. On two such accelerators, we find that this direct workflow returns a correct kernel on only 17 of 38 workload instances, and that most of its inference budget is spent re-deriving machine semantics and repairing illegal code rather than optimizing the workload.

We present Contour, which consists of two parts. Contour-Explore replaces native-code generation with search over a mapping surface: a typed set of scheduling knobs and their legal domains, generated automatically from the accelerator's ISA description and the workload graph. A compiler-owned resolver completes partial assignments into legal plans and emits native code, so every evaluated candidate is verified against the compiler's default. Kernel writing becomes knob assignment, and kernel optimization becomes design-space exploration that any policy can drive through the same intent interface, including hill climbing, evolutionary search, and LLM agents. Compared with KernelCraft, a state-of-the-art agentic kernel generator, an LLM agent through Contour-Explore returns a verified kernel on every Gemmini and PLENA instance, produces 2.86× and 1.13× faster kernels on the instances where both succeed, and uses 26.3% and 83.4% fewer tokens per verified kernel. KernelCraft returns a correct kernel on only 60% of the Gemmini instances and 5 of the 18 PLENA instances. The agent's kernels are also 1.17× faster than those from the expert-written tiling heuristics in Gemmini's software library and 1.46× faster than those from PLENA's compiler, and AccelOpt finds 1.50× faster kernels on Gemmini and 1.20× faster kernels on four PLENA attention instances when it searches through Contour-Explore instead of editing kernel programs.

A generated surface is bounded by its vocabulary. The second component—Contour-Ext—lifts this bound: when search stalls, an LLM agent reads the stalled program, hardware facts, and search history, then extends the backend itself by implementing a new knob and its lowering. Contour-Ext accepts the extension only if the default path is byte-identical, every new value is numerically correct and changes the program, and the enlarged surface yields a measured gain. On VTA, Contour-Ext extends the AutoTVM surface of TVM, and AutoTVM's XGBTuner then finds 4.14× faster kernels; on PLENA, the extended surface makes the kernels found by hill climbing 1.20× faster.

1 Introduction

Programmable tensor accelerators often expose target-specific instruction set architectures, explicitly managed on-chip memories, and specialized data-movement and control mechanisms [6, 9, 16, 18]. Supporting such a target requires a tensor compiler backend

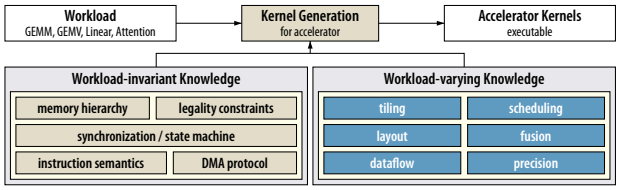


Figure 1: Kernel generation combines reusable machine-interface knowledge with workload-dependent mapping decisions.

to lower tensor operations to these interfaces, including instruction selection, memory placement, state configuration, and legal instruction sequencing [10, 13]. However, most proposed tensor accelerators lack reusable tensor compiler backends for generating implementations beyond supported kernels or templates, and rapid hardware iteration makes separately engineered backends difficult to maintain [11–13, 15]. When a reusable backend is unavailable, developers often fall back on manually written low-level kernels or target-specific compilation templates [15, 19].

Recent work has explored large language model (LLM)-based kernel generation and optimization across GPUs and specialized tensor accelerators [11, 21, 29]. These systems construct and revise kernel programs, guided by compilation, correctness tests, and performance feedback [10, 17, 19, 23, 28]. For example, KernelCraft generates and repairs ISA-level kernels from machine specifications, using compilation, execution, and correctness feedback [19], and AccelOpt starts from a correct kernel and improves performance using beam search, profiling feedback, and optimization memory distilled from slow–fast kernel pairs [28]. We refer to workflows in which agents generate or revise complete kernel programs as *program-level agentic generation*.

Across workloads, these workflows involve two classes of information with different reuse properties (Figure 1). Workload-invariant machine knowledge defines the target interface, including instruction semantics, the memory hierarchy, direct memory access (DMA) protocols, configuration-state transitions, synchronization, and legality constraints. Workload-dependent mapping decisions determine how a particular computation uses that interface, including tiling, scheduling, layout, fusion, dataflow, and precision. Both are required to produce an executable kernel, but machine knowledge can be reused across workloads, whereas mapping decisions generally vary across workload instances. Without a reusable backend, the agent handles both within the same per-workload loop, so each invocation spends inference and validation feedback not only on workload-specific optimization, but also on instruction legality, configuration-state transitions, data-movement protocols, and synchronization.

Recent advances make this coupling optional. TensorLift recovers executable tensor-level machine semantics from RTL [8] and

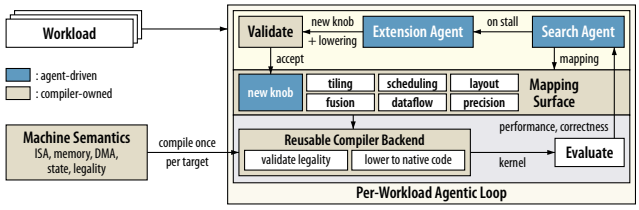


Figure 2: Compiler-mediated agentic kernel generation compiles machine semantics once while searching workload-dependent mappings per workload, and extends the mapping surface with a validated knob when search stalls.

represents them in the Tensor Accelerator ISA Definition Language (TAIDL) [14], from which a compiler backend can be generated automatically [15]. Workload-invariant machine knowledge can therefore be compiled once and reused across agent invocations.

A generated backend takes over machine-specific code generation. Two problems remain. First, search agents need explicit control over mapping decisions: where a compiler resolves implicit mapping choices, program-level workflows treat that completion as a black box and lack a shared contract for submitting partial mapping decisions and inspecting their resolution. Second, the compiler defines which mapping decisions an agent can express, and this set bounds every search policy: a mapping that no published decision expresses cannot be reached, whereas a program-level agent can write it directly. We address the first problem by separating mapping search from compiler-owned completion and native code generation, and the second by letting an agent extend the compiler when search stalls (Figure 2).

We present Contour, an agentic framework in which search agents choose mapping decisions and the compiler generates the kernel. Contour consists of two parts: Contour-Explore searches a mapping surface, and Contour-Ext extends it.

Contour-Explore derives a workload-conditioned mapping surface from formal, RTL-lifted machine semantics and publishes each mapping decision as a typed knob with its legal values. The agent submits a mapping intent: required or preferred knob values, with other decisions left to the compiler. For example, it may select an algorithm, prefer a tile size, and leave loop organization unspecified. A symbolic resolver completes a legal mapping for the backend to lower to native code; completing an empty intent yields a verified default kernel, Candidate-0, which initializes search. During search, feedback reports the realized mapping values, correctness, and performance to guide the agent's next mapping hypothesis. A *search agent*, the component that proposes mapping intents, can implement an LLM-based policy or a conventional search policy, such as hill climbing, evolutionary search, or random sampling; all policies share the same mapping surface, symbolic completion, code generation, and candidate-validation path.

Contour-Ext extends the mapping surface when search in Contour-Explore stalls. An LLM-based *extension agent* analyzes the stalled kernel and the search history, then implements a new knob and its lowering in the compiler. Contour-Ext keeps the knob only if it passes correctness and effectiveness checks, after which Contour-Explore republishes the surface and search continues. The agent's ability to write low-level code is thereby applied to the compiler instead of to a single kernel: an accepted knob is checked once and then remains available to later workloads and to every search policy. Contour-Ext requires only a knob registry, a lowering, legality checks, and an evaluator from its host compiler, so it also applies to existing compilers such as TVM [4].

This paper makes the following contributions:

- We characterize how program-level agentic generation repeatedly reconstructs workload-invariant machine semantics on accelerators without a reusable kernel-generation backend: when we replicate KernelCraft on PLENA, repairing programs and reasoning about machine semantics take 4.1× the completion tokens spent on optimizing correct programs.
- We design Contour-Explore, a workload-conditioned mapping-search interface that publishes typed decisions derived from the hardware specification and the workload. It supports partial intents with required, preferred, and automatic decisions, completes them into legal plans, and reports the realized mapping values as feedback to guide subsequent search.
- We design Contour-Ext, which extends the mapping surface when search stalls with knobs that pass correctness and effectiveness checks. On 16 VTA instances, Contour-Ext extends the AutoTVM surface of TVM, and AutoTVM's XGBTuner finds 4.14× faster kernels on the extended surface than on the original one. On 18 PLENA instances, the extended surface makes the kernels found by hill climbing 1.20× faster.
- We evaluate Contour-Explore against KernelCraft, AccelOpt, and the software shipped with each accelerator: the tiled matrix-multiply routine in Gemmini's official software library and the compiler of PLENA. An LLM agent searching through Contour-Explore returns a correct kernel on every Gemmini and PLENA instance, whereas KernelCraft fails on 21 of the 38; where both succeed, the agent's kernels are faster and cost fewer tokens. The agent's kernels are also 1.17× faster than the Gemmini library routine and 1.46× faster than those of the PLENA compiler. AccelOpt also finds faster kernels when it searches through Contour-Explore than when it edits kernel programs.

2 Preliminaries

2.1 Emerging Accelerator Architectures

We study three representative accelerator platforms: Gemmini [9], an established state-of-the-art tensor-accelerator generator; PLENA [26], a recently introduced open-source research accelerator for long-context agentic LLM inference; and VTA [18], an open-source deep-learning accelerator. All three expose target-specific instructions and explicitly managed on-chip memories, so mapping a tensor computation requires coordinating its computational decomposition with data placement and movement.

Gemmini. Gemmini is an open-source, full-stack DNN accelerator generator integrated with RISC-V systems. Its central compute structure is a configurable spatial array whose processing elements perform multiply–accumulate operations using weight- or output-stationary dataflow. The array reads operands from a banked, explicitly managed scratchpad and writes partial and final results

to a higher-precision accumulator. A Gemmini mapping therefore determines how matrices are tiled, staged through the scratchpad, accumulated, and executed under the selected dataflow.

PLENA. PLENA is designed for long-context agentic LLM inference and organizes its ISA around Matrix, Vector, Scalar, HBM, and Control instruction classes. Its Matrix Unit uses an output-stationary flattened systolic array: operands stream along a long reduction dimension while sub-arrays retain partial sums, which are combined by a cross-array reduction. PLENA separates its on-chip storage into a Vector SRAM for activations and vector intermediates and a Matrix SRAM for weights and KV tensors. The latter supports transposed and non-transposed reads without explicitly rearranging the stored data, directly supporting computations such as QK^{T}. The Vector Unit complements matrix multiplication with elementwise, reduction, and nonlinear operations needed by normalization and softmax.

PLENA composes these capabilities at tile granularity. Independent HBM prefetch paths can fill the Matrix and Vector SRAMs while other units execute, and fine-grained scheduling can interleave matrix multiplication, vector reduction, scalar control, and data movement within a fused attention pipeline. The architecture also supports mixed-precision MX formats across its storage and compute paths. Its programmable mapping surface consequently includes matrix and vector tiling, cross-array reduction, tensor orientation, SRAM placement and residency, prefetching and buffering, numerical precision, fusion, and overlap across execution units. Together, these are hardware-level mapping decisions that the compilation approaches below must ultimately realize.

VTA. The Versatile Tensor Accelerator (VTA) [18] is the open-source deep-learning accelerator of the TVM compiler stack [4]. Its GEMM core multiplies int8 tiles held in explicitly managed on-chip buffers and accumulates in int32. Its load, compute, and store modules run concurrently and synchronize through dependency queues, so data movement can overlap computation. TVM generates VTA code from a schedule, which fixes the tiling, loop order, and on-chip buffer placement of a tensor operator; its VTA backend maps the scheduled loops to VTA load, GEMM, ALU, and store instructions.

2.2 Machine Semantics and Generated Backends

We use *compiler backend* to mean the target-specific component that translates a tensor computation graph into accelerator assembly. Prior work provides the machine semantics that a backend needs in machine-readable form. The Tensor Accelerator ISA Definition Language (TAIDL) describes accelerator state, instruction semantics, and legality constraints using tensor operations [14]. TensorLift generates TAIDL from RTL: it lifts bit-accurate architectural-state updates into tensor-level descriptions of computation, data movement, and control, and validates them against the low-level model with SMT proofs and simulation [8]. A compiler backend can then be generated from the TAIDL specification [15]; on seven Gemmini workloads, TensorLift with such a backend achieves a 1.014× geometric-mean speedup over the official handwritten kernels [8].

These systems provide reusable machine semantics and native code generation, but do not expose workload-level mapping decisions to an external search policy or accept partially specified mapping intents. Contour introduces a mapping-search interface that publishes the decisions available for each workload and accepts required, preferred, and automatic choices. Its symbolic resolver completes partial intents into legal plans and reports their realized mapping values as feedback. The mapping surface, intent interface, resolver, and feedback-directed search workflow are introduced by this work.

Table 1: Tokens generated by KernelCraft on PLENA.

Token use	Completion	Reasoning
Repairing illegal or incorrect programs	5.79M	4.34M
Reasoning about machine semantics	3.99M	3.90M
Repairing or machine semantics	**8.28M**	**6.75M**
Optimizing a correct program	**2.04M**	**1.42M**
Ratio	4.1×	4.8×

2.3 Program-Level Agentic Generation

KernelCraft studies a different response to the absence of a reusable backend: a tool-using LLM agent directly generates and repairs a close-to-metal kernel for each workload [19]. Each task supplies a concrete workload instance, the target ISA and hardware documentation, and memory-layout information. A platform-specific execution harness gives the agent access to the target compiler, simulator, correctness oracle, and documentation retrieval. Together, these inputs and services form the agent's context. The required output is an ISA-level program that explicitly realizes both the computation and its use of the target machine.

Generation proceeds as a multi-turn diagnosis-and-repair loop. In each iteration, the agent produces a candidate assembly program, compiles and executes it, and receives syntax diagnostics, functional differences, and execution latency. When additional machine information is needed, the agent queries the supplied ISA and hardware documentation through `grep_docs`. It uses this feedback first to repair correctness and then to optimize performance, repeating the loop until its interaction budget is exhausted.

Because the generated artifact is already close to the target ISA, the agent is responsible for more than workload mapping. Within the same program, it must select instructions, calculate addresses, place values in the memory hierarchy, schedule data movement and computation, manage configuration state, and expose parallelism. KernelCraft shows that frontier agents can sometimes produce correct, performance-competitive kernels for unfamiliar accelerators. Program-level agents can also optimize an existing kernel. AccelOpt organizes this process around three agents: a planner, executor, and summarizer [28]. In its workflow, the planner proposes optimizations using profiling feedback and accumulated experience; the executor implements these plans as kernel programs; and the summarizer distills reusable optimization insights from slow–fast kernel pairs for subsequent exploration. In its Trainium implementation, the agents receive hardware constraints and NKI programming guidance through platform-specific prompts.

2.4 Motivation

KernelCraft's program-level generation workflow treats each workload task in isolation. Although the target ISA, memory organization, and instruction semantics remain unchanged across tasks, the workflow does not carry the machine-specific knowledge recovered

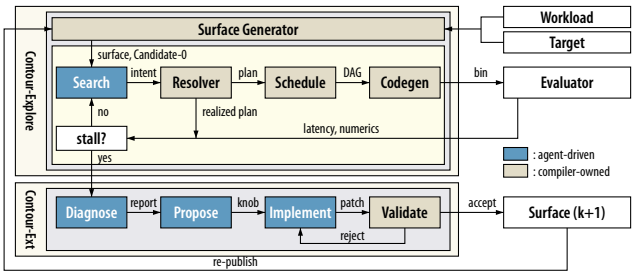


Figure 3: CONTOUR overview: Contour-Explore searches the published mapping surface, and Contour-Ext extends it when search stalls.

for one task into the next. Consequently, the target details required for native-code generation remain part of every per-workload interaction.

When we replicate KernelCraft's experiments on PLENA with its benchmark workloads, including GEMM, GEMV, normalization, softmax, FFN, and attention, and classify each model turn by tool verdict and by content, most completion tokens go to repairing programs and to reasoning about *machine semantics*: instruction semantics, operand conventions, memory layout, precision formats, legality, and timing. These two uses take 4.1× the completion tokens and 4.8× the reasoning tokens spent on optimizing correct programs (Table 1). Every run spends 9.7–50.9% of its completion tokens on this workload-invariant knowledge.

This observation motivates our compiler-mediated agentic design: a reusable compiler captures target-invariant code-generation knowledge and generates every candidate from the validated machine semantics, so that candidates are legal by construction and the agent can spend its tokens on workload-specific mapping decisions instead of program repair.

3 Design

CONTOUR consists of two parts (Figure 3). **Contour-Explore** connects a search agent to a reusable compiler through a workload-specific mapping interface. The agent chooses how to map a tensor computation; the compiler completes these choices and generates target code. An independent evaluator checks the output and measures performance. A controller returns these results to the agent and retains the fastest verified candidate. The same interface supports neural and conventional search policies. **Contour-Ext** extends this interface when search stalls. An extension agent implements a new knob and its lowering in the compiler, and Contour-Ext keeps the new knob only if it passes correctness and effectiveness checks. Sections 3.1–3.4 describe Contour-Explore, and Section 3.5 describes Contour-Ext.

3.1 Constructing the Mapping Surface

The input is a tensor computation in the high-level optimizer (HLO) intermediate representation [20] and a target described by TAIDL. A *workload contract* fixes the computation, input data, target, execution boundary, and numerical acceptance policy for every candidate. Within this contract, a (1) *mapping surface* lists the decisions that search may change, such as tiling, operand residency, or execution

Algorithm 1 Mapping search with explicit preference completion

(1) Surface (2) Hard checks (3) Preferences
(4) Region completion (5) Compile/evaluate (6) Selection

Require: workload W, target semantics T, search budget
(1) $S \leftarrow \textsc{DeriveSurface}(W, T)$
$p_0 \leftarrow \textsc{Resolve}(\varnothing, W, T)$
(5) $b \leftarrow \textsc{CompileAndEvaluate}(p_0, W)$
Require b to pass verification; $F \leftarrow \{b\}$
while proposal and evaluation budgets permit another batch **do**
  $I \leftarrow \textsc{Propose}(W, S, F)$
  if target is Gemmini **then**
    $p \leftarrow$ catalog plan selected by I
    $P \leftarrow \{p\}$ if p is fresh, else $\varnothing$
  else
    Check region I and completion of its hard subset
    (4) Sample I; complete intents with RESOLVE
    Deduplicate plans; expand a short neighborhood
    Filter static-cost outliers; select two diverse plans
    $P \leftarrow$ selected plans, or $\varnothing$ on rejection
  end if
  for all $p \in P$ **do**
    (5) $r \leftarrow \textsc{CompileAndEvaluate}(p, W)$
    if (6) $r.verified$ and r is faster than b **then**
      $b \leftarrow r$
    end if
  end for
  $F \leftarrow$ mapping outcomes, incumbent, remaining budget
end while
return b

function RESOLVE(I, W, T)
  (2) $A \leftarrow$ hard assignments in I
  $p \leftarrow$ complete A with defaults; check joint constraints
  Reject if completion fails
  for all preferences (k, v) in (3) fixed order **do**
    $q \leftarrow$ complete and check $A \cup \{k = v\}$
    if q is legal **then**
      $A \leftarrow A \cup \{k = v\}; p \leftarrow q$
    else
      Record the relaxation and its reason
    end if
  end for
  Construct plan sections and schedule DAG; reject cycles
  return plan with realized decisions and resolution records
end function

overlap. Each decision, or *knob*, has a type and a finite domain of supported values.

The surface generator constructs a workload-specific mapping surface using a TAIDL hardware description and a shared mapping layer. TAIDL specifies architectural state, storage capacities, instruction semantics, and legality constraints. The shared mapping layer defines the knob vocabulary and default policies, which are compiler design choices reused across accelerators. For each workload, the surface generator uses the hardware description and these shared policies to derive the available knob values, and the resolver checks their joint legality.

The shared mapping layer defines each knob category with four parts: a derivation rule that creates knob instances and candidate domains from the workload graph and TAIDL facts, a default policy, a legality predicate for its values, and a plan field that lowering reads. We call these four parts a *knob definition*. A *compiler version* is the compiler together with its registered knob definitions. The surface generator publishes one surface for each compiler version. Contour-Ext (Section 3.5) registers new knob definitions through

the same interface, so added knobs and built-in knobs use the same derivation, publication, and completion steps.

On PLENA, the surface generator first selects the knob categories applicable to the operator family. The eight categories cover algorithm, matrix and vector decomposition, placement, memory pipeline, execution pipeline, precision, and loop structure. For each applicable knob, it tests each supported value by requiring that value and completing the remaining decisions with the default policies. It retains values whose completion passes the constraints and omits knobs with only one retained value. Those fixed choices remain compiler-owned.

These per-knob tests do not make the domains independently combinable. The resolver checks joint constraints again when it completes an intent. For example, the attention algorithm determines whether scores are materialized; that choice changes SRAM demand and the legal block sizes. A conditional surface therefore publishes individual domains without enumerating their cross-product. Gemmini instead uses a finite catalog of complete mappings that pass the workload's shape, capacity, and scheduling checks. Both forms expose legal choices without exposing physical instructions or addresses.

3.2 Completing Mapping Intents

An *intent* assigns values to exposed knobs. Contour supports ❷ *hard* assignments that must hold and ❸ *preferred* assignments that may be relaxed. The resolver rejects unknown knobs and values outside the published domains, and fills automatic or omitted choices using shared default policies. Gemmini selects a complete catalog mapping and treats all selected values as hard assignments; PLENA also supports partial intents with preferences.

For PLENA, completion first applies all hard assignments together. The resolver fills the remaining choices with the shared default policies and checks shape, capacity, reduction, output-coverage, and dependency constraints. If this completion fails, it rejects the intent. It then processes preferences in a fixed order: category order followed by knob-name order. For each preference, it adds the requested assignment to the accepted assignments and attempts completion again. A successful check retains the assignment; a failure drops it and records the requested value, realized value, and reason. Preferences are incorporated incrementally in a fixed order, retaining each choice only when its completion remains legal alongside previously accepted assignments.

Mapping completion produces a *resolved plan*: a complete legal mapping with eight semantic sections for the computation graph, matrix and vector decomposition, placement, memory, precision, pipeline, and loops. It also constructs an engine-schedule directed acyclic graph (DAG) specifying operations and dependencies, rejecting cyclic schedules. Internal resolution records identify defaulted choices and honored or relaxed preferences. The resolved decisions guide subsequent lowering of the tensor computation to target code.

3.3 Feedback-Directed Search

Search begins by completing the empty intent. The resulting default plan, Candidate-0, uses the shared default policies under the target and workload constraints. It requires no model inference. The controller compiles, executes, and verifies this plan before using it as the initial *incumbent*, the fastest verified candidate so far. The agent receives the workload, surface, default decisions and performance, and remaining evaluation budget. PLENA additionally reports a bottleneck profile containing engine activity, HBM traffic, and SRAM occupancy.

Gemmini: selecting a legal point. The agent sees the catalog's mapping decisions, with performance hidden for untried mappings. It submits one complete mapping per evaluation. The controller rejects duplicate submissions and returns verification, measured cycles, and structural properties for each newly evaluated mapping. The agent can then refine a tile choice or explore different buffering and store schedules. Only candidates evaluated within the run enter final selection.

PLENA: proposing a region. A ❹ *search region* describes how the controller should generate candidate intents. Hard assignments remain fixed; preferred assignments guide sampling; explicitly automatic knobs are left to compiler completion. Other unassigned knobs remain available for sampling. Each region includes at least one *diversity axis*, identifying a knob along which alternative plans are desired, and leaves at least two knobs free of hard assignments. It may also name an evaluated plan as a center and specify an initial radius in changed decisions. Diversity guides sampling and selection without adding legality constraints.

The controller constructs a center using preferred values, then the named plan's remaining values, or domain midpoints when no plan is named. Explicitly automatic knobs remain omitted from candidate intents. Without a named plan, the controller samples up to 256 assignments: preferred knobs take their requested values with probability 0.75 and otherwise draw uniformly; other free knobs draw uniformly. With a named plan, sampling changes one to the radius's number of knobs, prioritizing diversity axes. Each sampled assignment becomes a candidate intent whose sampled values are preferences and whose hard assignments are preserved from the region.

The controller removes rejected intents and plans that duplicate either previous evaluations or other plans in the pool. Deduplication uses the eight semantic sections. If a neighborhood supplies fewer than two fresh plans, it expands the radius up to eight; a pool still smaller than two is rejected. It filters the worst quarter by estimated static cost, preserving the center and at least two plans. This estimate filters candidates rather than choosing the measured winner.

Two plans enter evaluation. The first is the center when available; otherwise, it is the plan most distant from previously evaluated plans. The second prioritizes a new value on the first diversity axis, then maximizes distance from the first plan and evaluated plans. Distance is normalized Hamming distance over the eight semantic sections. Both plans obey the hard assignments, but their realized preferences can differ after completion.

Algorithm 1 combines this target-specific proposal path with the common search loop. Feedback reports each plan's identity, verification, measured performance, and realized decisions relative to the default. A rejected region instead receives a mapping-level explanation, such as conflicting hard assignments or exhausted alternatives. The agent can revise its proposal, but only the controller can update the incumbent. The controller bounds both proposals

and evaluations. Rejected regions consume requests without consuming evaluations, so repeated rejection cannot prolong search indefinitely. A new batch requires sufficient remaining evaluation budget; CANDIDATE-0 is outside this search budget.

Conventional search policies. Hill climbing, genetic search, and random sampling implement the PROPOSE step in Algorithm 1 over the published mapping decisions. Hill climbing explores neighboring assignments, genetic search mutates and recombines typed intents, and random search samples assignments from the exposed domains. CONTOUR completes these proposals into legal mappings and generates kernels for verification and measurement. Measured latency provides the objective value for hill climbing and genetic selection; random sampling proceeds independently of these values. Across all policies, the controller retains the fastest verified candidate. Thus, conventional policies search mapping choices without implementing kernel generation or legality checks.

Stall detection. Search *stalls* when the incumbent stops improving: the relative latency improvement over a window of recent evaluations falls below a threshold, or the policy can reach no unevaluated program. An exploration round also ends when an LLM search agent produces no new evaluation for a fixed number of turns, or when the round's evaluation budget is spent. The incumbent at the end of the round is the *stalled plan*, p_s. Contour-Ext (Section 3.5) starts from p_s.

3.4 Code Generation and Candidate Validation

Gemmini realization. CONTOUR exposes macro tiling, buffering, store scheduling, and batch-region choices through a legal mapping catalog. Macro-tile dimensions are multiples of the primitive tile, bounded by workload dimensions and scratchpad and accumulator capacities. The selected mapping specifies the choices that the subsequent schedule transformations must realize; choices with only one applicable value remain fixed.

The existing generated backend (5) lowers the HLO computation to a primitive schedule with concrete transfer and matrix-operation addresses and sizes. CONTOUR then applies schedule transformations to implement accumulator initialization, the selected buffering strategy, and store order. The schedule parser checks the transformed schedule before code generation. This additional transformation stage connects the searchable mapping choices to their executable realization.

PLENA realization. The resolver translates the completed mapping into typed bridge options that CONTOUR passes to code generation, making the resolved computation, placement, data movement, and execution overlap explicit inputs to lowering. Instruction selection, physical memory management, and synchronization realize these decisions as target code. From the same options, the resolver derives an engine-schedule DAG for symbolic legality checks and cost estimation. The mapping interface thus gives search control over the plan realized by the backend. A schedule layer reads the plan fields and selects operator decompositions, placement, tensor storage, and loop nests before instruction selection. Instruction selection maps abstract tile operations to target instructions by semantic signatures derived from TAIDL. Lowering either implements every value in a resolved plan or reports that it cannot; it does not fall back to another value. On request, lowering tags each instruction with the schedule-layer step that produced it.

Algorithm 2 Extending the mapping surface when search stalls

(1) Diagnosis (2) Proposal (3) Implementation
(4) Correctness checks (5) Effectiveness checks

Require: workload W, target semantics T, compiler C, default plan p_0, stalled plan p_s, earlier rejections H

(1) Profile p_s and report its bottleneck B
while the proposal budget permits **do**
  (2) $\kappa \leftarrow$ PROPOSEKNOB(B, H) ▷ values and lowering
  if no knob is proposed **then break**
  (3) $C' \leftarrow C$ with κ registered and its lowering implemented
  $S' \leftarrow$ DERIVESURFACE(W, T, C')
  (4) Reject κ if any of:
    S' exposes no new knob
    a new value leaves the program unchanged
    the default value changes the program of p_0 or p_s
    a new value exceeds the output-error bound
  (5) $p^* \leftarrow$ hill climbing on S' from p_s
  (5) Reject κ if any of:
    p^* does not use κ
    the program of p^* is reachable without κ
    p^* is not faster than p_s by the gain threshold
    a new value exceeds the error bound on scaled or outlier inputs
  if κ is not rejected **then return** (C', p^*)
  $H \leftarrow H \cup \{(\kappa, \text{rejection reason})\}$ ▷ revise or propose anew
end while
return (C, p_s)

Independent validation. The evaluator executes each generated program and compares its complete output with an independent numerical oracle under the workload contract. The acceptance policy remains fixed across mapping and precision choices. The evaluator returns correctness and performance to the controller, together with structural counters when available. A candidate replaces the incumbent only when it (6) passes verification and strictly improves the performance objective. Otherwise, the previous verified candidate remains the result. The agent cannot change the acceptance criteria or the final selection rule. The evaluator identifies each program by a hash of its assembly and does not execute the same program twice. CONTOUR keeps the hashes of all programs measured in the run.

3.5 Extending the Mapping Surface

Contour-Ext adds a knob to the surface when Contour-Explore stalls. It uses four parts of Contour-Explore: the knob registry, the lowering, the legality checks, and the evaluator. Algorithm 2 shows one invocation.

Analysis of the stalled plan. Contour-Explore passes Contour-Ext the stalled plan p_s, CANDIDATE-0 (p_0), the search history, the hashes of measured programs, and earlier rejected knobs (H). In (1) diagnosis, an LLM-based extension agent examines p_s without editing code. It reads the assembly, with each instruction tagged by the lowering step that produced it, and an emulator profile of memory traffic and unit occupancy. It ends with a bottleneck report B that cites measurements and code locations.

Implementation of a new knob. In (2) proposal, the agent specifies one knob that addresses B and differs from every entry in H: its values with the default first, the lowering change for each value, and the operator families it applies to. In (3) implementation, the agent registers the knob with a derivation rule and a legality predicate, and implements its values in the schedule layer. It cannot change the computation, instruction selection, or the evaluator.

Before submission, it can compile and emulate plans through the Contour-Explore resolver and evaluator.

Correctness and effectiveness. Contour-Ext accepts a knob only if it passes two groups of checks. The ❹ correctness checks require that the default value reproduces the programs of the previous compiler version at p_0 and p_s, that each new value changes the program, and that each new value keeps the output error within the numerical bound of the target. The ❺ effectiveness checks run hill climbing on the extended surface S' from p_s. Its best plan p^* must use the new knob, its program must not be reachable without the knob, and it must be faster than p_s by a gain threshold. The new values must also meet the numerical bound at p^* on inputs with large magnitudes and outliers. A failed check returns its reason to the agent, which revises the patch or proposes another knob; each rejected knob is added to H.

Return to Contour-Explore. An accepted knob becomes part of a new compiler version. Contour-Explore publishes the extended surface and resumes search from p^*; its search policies use the new knob in the same way as any other knob. When Contour-Ext returns no knob, Contour-Explore continues with the previous compiler version.

4 Evaluation

4.1 Setup

We use Gemmini to evaluate Contour-Explore (Section 4.2), VTA with TVM and AutoTVM to evaluate Contour-Ext (Section 4.3), and PLENA to evaluate the full system (Section 4.4).

Accelerator platforms. We evaluate Contour on Gemmini, VTA, and PLENA. For Gemmini, we use the `GemminiRocketConfig` provided by Chipyard [3, 9], whose default Gemmini configuration sets `DIM=16` and instantiates a 16×16 systolic array. We build the bitstream of this SoC with Vivado 2022.2, generate its software image with PetaLinux 2022.2, and deploy the resulting system on a Xilinx ZC706 FPGA evaluation board. We obtain Gemmini performance measurements from this physical implementation.

For VTA, we use the TVM Versatile Tensor Accelerator [18] and measure cycles with tsim, which simulates its Chisel RTL with Verilator. The evaluator compares each output element by element with a CPU reference.

For PLENA, we use the authors' official RTL release and the transaction-level mode of their official simulator [2, 26]. The simulator executes compiled assembly against an explicit memory layout and produces both numerical outputs and simulated latency.

Models and workloads. All main experiments use DeepSeek-V4.1-Flash [7], with the same model settings across the compared LLM methods; the extension agent of Contour-Ext uses the same model. On Gemmini and PLENA, we adopt the workload definitions and input shapes from KernelCraft [19]. Each workload–input-shape combination is a *workload instance*. Within each workload family, C1, C3, and C5 denote the smallest, middle-sized, and largest workloads, respectively. The evaluation covers 20 Gemmini instances and 18 PLENA instances: FFN, SwiGLU, FlashAttention, MHA, GQA, and MQA, each with shapes C1, C3, and C5. These multi-stage workloads exercise fusion and attention computations beyond standalone GEMM and GEMV. On VTA, we use 16 int8 instances with int32 accumulation, shift, and clipping: FFN (two dense layers with ReLU), GEMM, and GEMV, each with shapes C1–C5 from smallest to largest, and the fully connected layer of ResNet-18 (R18). AutoTVM [5] tunes the knobs of a hand-written TVM schedule template. The TVM 0.12 VTA dense template exposes tile sizes for the batch, input-channel, and output-channel dimensions and the number of virtual threads over output channels; we call the mappings expressible by these knobs the *AutoTVM surface*. The *Contour-Ext surface* of an instance adds the knobs that Contour-Ext accepts on it. Hill climbing runs on the AutoTVM surface until it stalls; Contour-Ext then extends the surface from that stall, and hill climbing continues on the Contour-Ext surface. We also run the AutoTVM XGBTuner, GATuner, and RandomTuner on both surfaces with up to 512 measurements per run and report the median of three seeds.

Search agents and baselines. Our *naive agent* follows the feedback-directed loop in Section 3: an LLM proposes a mapping intent, receives correctness and performance feedback, and uses that feedback to propose the next intent. We denote this configuration Naive + Contour-Explore. Contour-Explore retains the fastest verified kernel found during the run. We also connect hill climbing and a genetic algorithm (GA) to the same interface. These policies share the mapping surface, symbolic completion, code generation, and evaluator.

KernelCraft generates kernels without a supplied starting kernel and provides the ISA-level generation baseline. Section 5.1 evaluates the AccelOpt optimization harness through Contour-Explore and at the program level. We also report the software shipped with each accelerator: Gemmini's library matrix-multiply routine (`tiled_matmul_auto` in `gemmini.h`) [9], which we call the Gemmini library, and the PLENA compiler [2], which compiles each instance once without search.

Search budgets and reporting. Contour-Explore search permits up to 80 candidate evaluations per PLENA instance and 12 unique candidate evaluations per Gemmini instance. On PLENA, hill climbing and GA use all 80 evaluations. The naive agent can stop earlier: its run is also bounded in the number of LLM turns and requests, and those that repair a rejected intent or yield no new candidate consume this bound without an evaluation. We report the evaluation counts. Methods with three runs report the median latency for each instance. Token usage and evaluation counts are averaged across the runs. The naive agent uses three runs per instance. On PLENA, GA uses three runs and hill climbing uses one; on Gemmini, hill climbing and GA use 1,000 seeds per instance. KernelCraft uses three runs per instance. To extend the PLENA surfaces, hill climbing runs in iterations of 16 evaluations and invokes Contour-Ext when the relative improvement over the last six iterations is below 0.2%. The effectiveness checks run 40 evaluations of hill climbing and require a gain of at least 1%. The extension of an instance ends after three consecutive rejected invocations, eight accepted knobs, 4,000 evaluations, or 48 hours.

Metrics. We report geometric-mean speedup over a reference kernel, which is the Gemmini library on Gemmini and Candidate-0 on VTA and PLENA, and verified coverage, the fraction of instances for which a method returns a verified kernel. *Tokens per verified kernel* equal the total prompt and completion tokens across all runs divided by the number of runs that return a verified kernel.

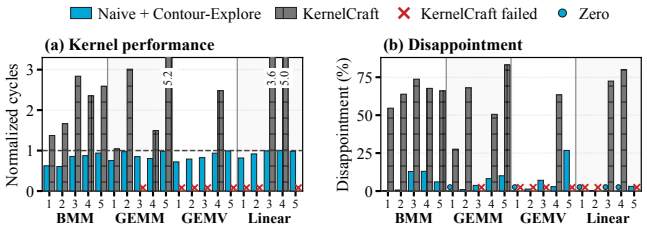


Figure 4: Gemmini results on 20 instances: (a) cycles normalized to the Gemmini library (dashed line); (b) disappointment. Crosses: KernelCraft failures. BMM: BatchMatMul.

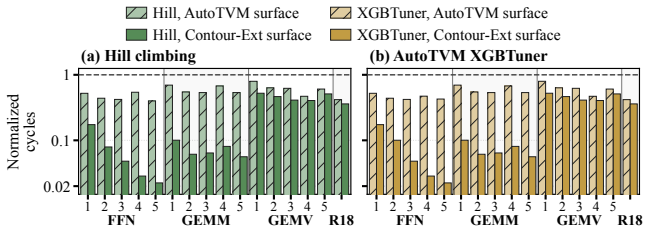


Figure 5: VTA results on 16 instances: cycles normalized to Candidate-0 (log scale). XGBTuner reports the median of three seeds.

Failed runs contribute their token usage to the numerator. Each run contributes at most one kernel to the denominator.

We define *disappointment* on an instance as $d = 1 - t_{\text{best}}/t_{\text{method}}$, where t_{method} is the latency returned by the method and t_{best} is the best latency known for the instance. On Gemmini, it is the lowest latency that the naive agent, hill climbing, or GA finds in any run, so a method outside these three can have negative disappointment; on PLENA, it is the lowest latency among the compared methods and the longer search runs on the merged Contour-Ext surface (Section 4.4). A method that returns no verified kernel has $d = 100\%$. We report mean disappointment across instances.

4.2 Contour-Explore on Gemmini

Comparison. We compare the kernels found by the naive agent, hill climbing, and GA through Contour-Explore with the Gemmini library and with KernelCraft across 20 instances from BatchMatMul, GEMM, GEMV, and Linear. Every kernel uses the same workload inputs, correctness oracle, and execution boundary. The naive agent, hill climbing, GA, and KernelCraft have the same evaluation budget of 12 evaluations per instance. We also compare verified coverage and tokens per verified kernel against KernelCraft.

Search through Contour-Explore against the hand-written library. Figure 4a normalizes cycles to the Gemmini library, the kernel that a Gemmini user obtains without search. With 12 evaluations, the naive agent returns a faster kernel than the library on 19 of the 20 instances, and its kernels are 1.17× faster in geometric mean. The gain holds in every workload family: 1.30× on BatchMatMul, 1.18× on GEMV, 1.15× on GEMM, and 1.06× on Linear, where the library is 0.4% faster on Linear C4. Under the same budget, hill climbing and GA are 1.17× and 1.10× faster than the library. Mean disappointment is 4.9% for both the naive agent and hill climbing, 10.7% for GA, and 18.2% for the library: every policy is closer on average than the library to the best kernel found on the surface.

Verified coverage and kernel quality. KernelCraft returns a verified kernel on 12 of the 20 instances (60%). On the other eight, none of the programs that it generates within its budget compiles and passes the correctness check; we count these instances as failures. The naive agent through Contour-Explore returns a verified kernel on all 20 instances in each run. On the 12 instances with a verified KernelCraft result, the naive agent's kernels are 2.86× faster than KernelCraft's in geometric mean, and KernelCraft's kernels are slower than the Gemmini library on all 12, by 2.44× in geometric mean. This performance comparison is conditioned on KernelCraft success; the coverage comparison includes the entire suite. The naive agent returns a faster kernel than KernelCraft on every instance; mean disappointment is 78.6% for KernelCraft, including its eight failures at 100% (Figure 4b).

Cost per verified kernel. Table 3 reports 564K tokens per verified kernel for the naive agent through Contour-Explore, compared with 765K for KernelCraft, a 26.3% reduction. The denominator accounts for the KernelCraft runs that consume tokens without returning a verified kernel. Coverage from the compiler default and verified incumbent selection converts search expenditure into a usable result on every instance.

4.3 Contour-Ext on VTA

Hill climbing on the AutoTVM surface stalls at a 1.84× geometric-mean speedup over Candidate-0, which on VTA is the kernel compiled from the default configuration of the TVM template. All 32 Contour-Ext invocations, two per instance, are accepted; they use 3.60M tokens in total, or 112K tokens per invocation. On the resulting Contour-Ext surface, hill climbing reaches 7.77×, and its final cycles are 4.22× lower than at the stall in geometric mean (Figure 5a). The largest speedups occur on FFN and GEMM, at 18.83× and 14.10× over Candidate-0; GEMV and R18 reach 2.17× and 2.78×. The most frequent added knobs move operand loads out of inner loops, followed by knobs for loop order and tiling. The added knobs also improve search procedures that were not used to find them. XGBTuner reaches 1.85× over Candidate-0 on the AutoTVM surface and 7.66× on the Contour-Ext surface, so its kernels are 4.14× faster on the Contour-Ext surface (Figure 5b), and GATuner and RandomTuner also reach fewer cycles on the Contour-Ext surface on all 16 instances. On FFN, most tuner measurements on the AutoTVM surface fail because many tile combinations are illegal; for XGBTuner with seed 1, the failure rate is 99.6% on FFN C4.

4.4 Full System on PLENA

On PLENA, we first measure how far Contour-Ext raises the performance that search can reach on the surface, and then evaluate Contour-Explore on the final surface against KernelCraft.

Contour-Ext raises the reachable performance. We call the surface that the surface generator publishes for an instance its *base surface*. On the base surface, hill climbing reaches a 1.048× geometric-mean speedup over Candidate-0 after 3,000 evaluations per instance, and it improves Candidate-0 by less than 3% on 14 of the 18 instances (Figure 6). Contour-Ext is invoked 119 times across the 18 instances and accepts 47 knobs, using 2.30B tokens in total, or 19.3M tokens per invocation. With the same 3,000 evaluations, hill

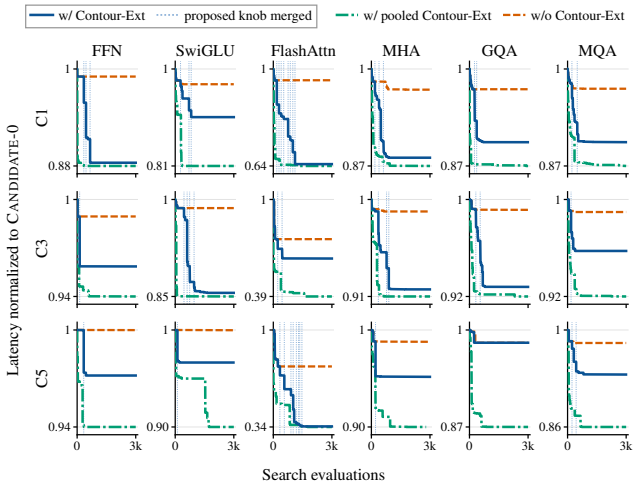


Figure 6: Best latency so far on 18 PLENA instances, normalized to CANDIDATE-0: base surface (w/o Contour-Ext), per-instance Contour-Ext surface (w/ Contour-Ext; dotted lines mark accepted knobs), and merged Contour-Ext surface (pooled).

climbing on each instance's Contour-Ext surface reaches 1.191×. FlashAttention gains the most, from 1.229× on the base surface to 1.935×.

A knob definition derives its values from the workload, so a knob accepted on one instance can be published on the others. We merge the knobs accepted on the 18 instances into one surface with 30 knob definitions after removing duplicates; CANDIDATE-0 is the same program on all surfaces. On this *merged Contour-Ext surface*, hill climbing reaches 1.262× after 3,000 evaluations (median of three seeds), which is 1.20× faster than its result on the base surface, and 1.157× after the first 80 evaluations. The merged surface reaches at least the speedup of the instance's own Contour-Ext surface on 17 of the 18 instances. On GQA C5, where Contour-Ext accepted no knob, it raises the speedup from 1.017× to 1.135×.

Contour-Explore on the merged surface. We run the naive agent, hill climbing, and GA through Contour-Explore on the merged Contour-Ext surface with the 80-evaluation budget and compare them with KernelCraft and the PLENA compiler (Figure 7, Table 2). Disappointment is measured against the best known latency on the merged surface, found by the longer runs in Figure 6, which reach 1.269× speed-up over CANDIDATE-0.

Search efficiency. The naive agent improves CANDIDATE-0 on all 18 instances and reaches a 1.243× geometric-mean speedup with a mean of 67.9 evaluations per instance. Its mean disappointment is 2.1%, and its disappointment is below 3% on 15 instances; on MHA C1, it finds a kernel 0.4% faster than the best one from the longer runs. Hill climbing and GA reach 1.157× and 1.139×, with mean disappointment of 8.3% and 10.1%; the naive agent's kernels are 7.4% and 9.1% faster than theirs in geometric mean. Among the three policies, the naive agent returns the lowest latency on 15 instances. The largest gaps occur on FlashAttention C3 and C5, where hill climbing has 30.2% and 31.9% disappointment after 80 evaluations and the naive agent has 4.1% and 1.6%.

Comparison with KernelCraft. KernelCraft returns a verified kernel on 5 of the 18 instances and fails on the other 13 by the

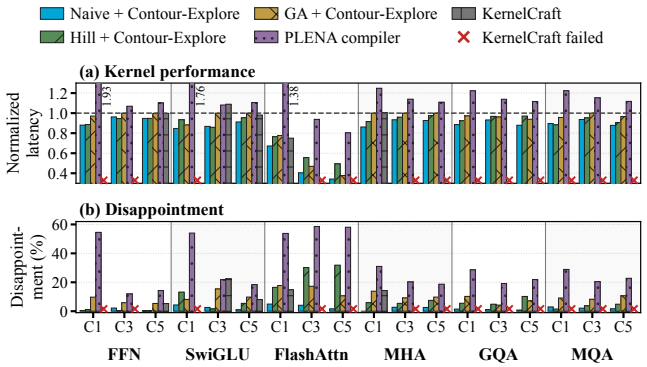


Figure 7: PLENA results on the merged Contour-Ext surface (18 instances): (a) latency normalized to CANDIDATE-0 (dashed line); (b) disappointment. Crosses: KernelCraft failures. Hill: first 80 evaluations of one run.

Table 2: Contour-Explore search policies on the merged Contour-Ext surface (18 PLENA instances). Evaluations: mean per instance, with a budget of 80.

Policy	Speedup	Disappointment	Evaluations
Naive	1.243×	2.1%	67.9
Hill	1.157×	8.3%	80.0
GA	1.139×	10.1%	80.0

Table 3: Tokens per verified kernel. Both methods use three runs per instance.

Platform	Naive + Contour-Explore	KernelCraft	Reduction
Gemmini (20)	564K	765K	26.3%
PLENA (18)	1.393M	8.375M	83.4%

criterion of Section 4.2, while every policy through Contour-Explore returns one on all 18. On these five instances, KernelCraft reaches 1.045× over CANDIDATE-0 with 13.0% mean disappointment, and the naive agent reaches 1.181× with 1.8%; the naive agent is faster on all five. Pooling tokens over all runs, the naive agent uses 1.393M tokens per verified kernel and KernelCraft uses 8.375M, an 83.4% reduction (Table 3).

Comparison with the PLENA compiler. The speedups above are measured over CANDIDATE-0, and CANDIDATE-0 is itself faster than the PLENA compiler on 16 of the 18 instances: the compiler's programs reach 0.849× the speed of CANDIDATE-0 in geometric mean. The two exceptions are FlashAttention C3 and C5, where the compiler skips key blocks that the causal mask excludes entirely and CANDIDATE-0 computes them. Every policy through Contour-Explore returns a faster kernel than the compiler on all 18 instances, including these two, and the naive agent's kernels are 1.46× faster in geometric mean (Figure 7).

5 Discussion

5.1 Sensitivity to Model Capability and Optimization Harness

We examine whether changing the model improves program-level generation and search through Contour-Explore. The Gemmini ablation compares Flash and Pro within the DeepSeek-V4 family;

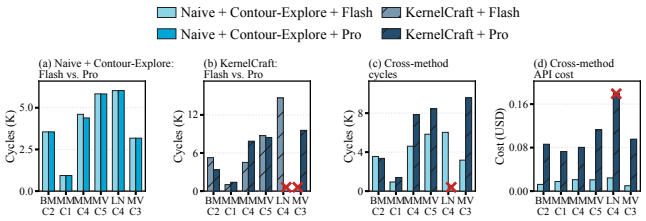


Figure 8: Gemmini Flash/Pro ablation: (a) Naive + Contour-Explore; (b) KernelCraft; (c, d) cycles and API cost for KernelCraft-Pro versus Naive + Contour-Explore + Flash. Crosses denote failures. BM/MM/MV/LN: BatchMatMul/GEMM/GEMV/Linear.

Table 4: PLENA coverage on 18 instances; geometric-mean speedup over Candidate-0 on the 15 instances solved by KernelCraft + Sol. Higher is better.

Method	Verified	Speedup ↑
Candidate-0	18/18	1.000×
KernelCraft + Sol	15/18	0.894×
Naive + Contour-Explore + Sol	18/18	1.262×

the PLENA experiments extend the comparison to GPT-6.1-Sol (Sol), separating generation success from the performance of verified kernels.

Gemmini: limited gains from Flash to Pro. We retain six instances selected from the Flash results to span easy, intermediate, and success-boundary cases. Each model–method pair reports seed 1, with candidates replayed under the same board timing harness (Figure 8). For Naive + Contour-Explore, five instances differ by at most 0.1%; Pro improves GEMM C4 by 4.7%. Across all six, Pro lowers the geometric-mean cycle count by 0.8% while increasing API cost by 17.0×. KernelCraft's response is mixed: Pro solves GEMV C3, where Flash fails, but fails on Linear C4, which Flash solves. Among the five instances solved by KernelCraft-Pro, Naive + Contour-Explore + Flash is 1.60× faster in geometric mean. These results establish limited benefit from this model upgrade on the tested Gemmini mappings.

PLENA: generation success and kernel quality. Following the evaluation protocol used above, we compare KernelCraft and Naive + Contour-Explore with Sol on the 18 PLENA instances. KernelCraft + Sol verifies 15 instances, versus 5 for DeepSeek-Flash (median of three runs); Naive + Contour-Explore + Sol verifies all 18.

Table 4 compares performance on the 15 instances solved by KernelCraft + Sol. Even after excluding its failures, KernelCraft's geometric-mean speedup over Candidate-0 is only 0.894×. Naive + Contour-Explore with the same model achieves 1.262× speedup on those same instances. Sol therefore helps program-level generation solve previously unsuccessful workloads, while the generated kernels collectively remain slower than the compiler default and the mappings found through Contour-Explore.

AccelOpt through Contour-Explore. AccelOpt is an LLM optimization harness with a planner, executor, and summarizer [28]; we build on its public repository at commit `47e29a7` [1]. Program-level AccelOpt edits the kernel program, while AccelOpt through Contour-Explore proposes mapping intents through the interface of Section 4.2; both start from Candidate-0. On the 20 Gemmini

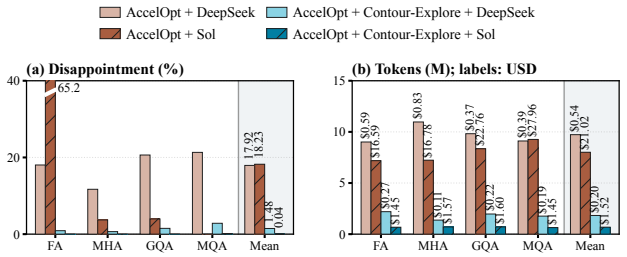


Figure 9: AccelOpt on four PLENA attention instances (C3): (a) disappointment; (b) tokens per run (bars) and API cost in USD (labels). FA: FlashAttention.

instances with DeepSeek-Flash, AccelOpt through Contour-Explore returns a faster kernel than program-level AccelOpt on every instance. Its kernels are 1.50× faster in geometric mean, mean disappointment falls from 36.8% to 6.3%, and it uses 588K instead of 620K tokens per instance. Program-level AccelOpt returns a faster kernel than the Gemmini library on 4 of the 20 instances, and AccelOpt through Contour-Explore on 17. Contour-Explore uses three runs and program-level AccelOpt uses one.

PLENA: model choice within AccelOpt. We next compare DeepSeek-Flash and Sol within the AccelOpt harness on four attention workloads: FlashAttention, MHA, GQA, and MQA, each at shape C3, with one run per configuration. AccelOpt + Sol tests whether a more capable model can complete the program-level route without a compiler: it optimizes the kernels that KernelCraft + Sol generates, whereas the other configurations start from Candidate-0. Within AccelOpt + Contour-Explore, replacing DeepSeek-Flash with Sol improves all four instances (Figure 9): mean disappointment falls from 1.48% to 0.04%, mean tokens fall from 1.83M to 0.68M (62.7% fewer), and geometric-mean speedup over Candidate-0 rises from 1.223× to 1.241×. AccelOpt + Sol raises the geometric-mean speedup of its KernelCraft-generated kernels over Candidate-0 from 0.771× before optimization to 0.934× afterward, but retains 18.23% mean disappointment and uses 8.01M tokens per instance. **With the same Sol model, AccelOpt + Contour-Explore achieves 32.8% higher geometric-mean performance than this program-level route at just 7.2% of AccelOpt's API cost ($1.52 versus $21.02 per instance).** The benefit also holds for DeepSeek: Contour-Explore reduces mean disappointment from 17.92% to 1.48%, raises the geometric-mean speedup over Candidate-0 from 1.018× to 1.223× (1.20× faster kernels), and uses 1.83M instead of 9.73M tokens per instance.

Model capability thus remains useful on PLENA, both for generating correct programs and for improving mapping search. The mapping interface preserves that opportunity while providing a verified default that stronger models can improve upon.

5.2 Ablation of Contour-Ext

We remove one mechanism of Contour-Ext at a time and repeat the extension loop of Section 4.4 on four PLENA instances, FlashAttention C1, SwiGLU C3, MHA C1, and FFN C1, with one run per configuration (Table 5). Without diagnosis, the extension agent starts at the proposal phase and has no profiling tool. Without checks, Contour-Ext keeps every patch that applies and compiles Candidate-0, and skips the correctness and effectiveness checks.

Table 5: Ablation of Contour-Ext on four PLENA instances: accepted knobs out of invocations and geometric-mean speedup over CANDIDATE-0. †Excluding the incorrect knob.

Configuration	Accepted	Speedup
Full Contour-Ext	18/33	1.239×
A No diagnosis	9/25	1.143×
A No failure feedback	8/22	1.143×
B No checks	15/30	1.195×†
C Leave-one-out	–	1.215×
Base surface	–	1.024×

Without failure feedback, the history H of rejected knobs is empty at every invocation.

A **Proposals need evidence.** Diagnosis and failure feedback both tell the extension agent where to look: the profile of the stalled program shows what limits it, and H shows what has already failed. Without either, fewer proposals pass the checks, the loop ends earlier, and the final kernels are slower (Table 5); on FFN C1, no knob is accepted at all. Without H, the agent also proposes rejected knobs again more often. To find useful knobs, the agent therefore needs measured bottlenecks and past failures in addition to the lowering code.

B **Without checks, an incorrect knob reaches the final kernel.** Without the checks, Contour-Ext accepts a knob that skips writing an intermediate result of FlashAttention C1 back to memory, although the program later reads it. The program still passes the evaluator, because that memory happens to hold valid values, and it is faster, so hill climbing selects it for the final kernel. When memory is filled with NaN before execution, its outputs contain NaN. Passing the evaluator during search does not show that a knob is correct, so new knobs must be checked on additional inputs before they enter the surface.

C **Accepted knobs transfer across workloads.** When the knobs that only the target instance contributed are removed from the merged surface, the remaining knobs, all proposed for other instances, still recover most of the gain of the instance's own extension loop. On the two attention instances, they recover all of it. The knobs that Contour-Ext accepts are therefore reusable compiler capabilities that apply beyond the workload they were proposed for, consistent with the merged surface in Section 4.4 matching or exceeding each instance's own surface on 17 of the 18 instances.

6 Related Work

KernelCraft generates ISA assembly for emerging accelerators, AutoComp optimizes ISA-level C, and AccelOpt optimizes Trainium NKI kernels using profiles and optimization memory [10, 19, 28]; GPU kernel agents and benchmarks take the same program-level approach [17, 21, 23]. In these workflows, the agent expresses its mapping choices through generated or modified kernel code, which at the ISA level includes instruction calls, address calculations, and data-movement schedules. In Contour-Explore, the agent sets knob values and the compiler generates the kernel; the search policies of these harnesses can use the same interface, as our AccelOpt integration shows (Section 5.1).

AutoTVM tunes the knobs of developer-written schedule templates [5], and Ansor and MetaSchedule generate schedule spaces from fixed rules [24, 30]. Timeloop constructs per-layer mapspaces from hand-written architecture descriptions and scores mappings analytically [22]. In these systems, the space is fixed before search, and a mapping outside it is unreachable for any search policy. Contour-Explore derives its mapping surface from the RTL-lifted machine semantics and measures every mapping as a generated kernel; when search stalls, Contour-Ext adds knobs and their lowering to the compiler, extending the surface itself (Section 4.3).

Concurrent to our work, CAKE [27] proposes co-designing a GPU compiler with kernel agents: agents write a typed, hardware-explicit schedule IR, and the compiler returns localized diagnostics and evolves its verifier rules and IR primitives when candidates expose missing capabilities. Our work differs in three respects. First, CAKE targets NVIDIA GPUs, for which large libraries of production kernels exist, and its methodology builds on them: the IR is distilled from production kernels such as those in FlashInfer and CUTLASS. Emerging accelerators have no such corpus. Second, CAKE has no formal machine semantics; its primitives come from production kernels and hardware documentation and are validated by tests over its kernel corpus. CONTOUR instead derives its knobs from the formal, RTL-lifted machine semantics of the target, and Contour-Ext adds a knob only after it passes legality checks against these semantics and the numerical oracle. Third, CAKE's candidates are written by LLM agents, whereas the mapping surface of Contour-Explore can be searched by LLM agents and by conventional policies such as hill climbing and GA alike.

7 Conclusion

We present CONTOUR, a compiler-mediated framework for kernel generation on emerging accelerators. Contour-Explore connects search policies to a reusable compiler through a workload-specific mapping surface: the compiler completes partial mapping intents into legal plans, generates target code, and returns verified correctness and performance feedback. Contour-Ext extends this surface when search stalls: an extension agent implements a new knob and its lowering, and Contour-Ext keeps the knob only if it passes correctness and effectiveness checks.

The naive agent searching through Contour-Explore returns a correct kernel on every instance, whereas KernelCraft returns one on only 60% of the 20 Gemmini instances and 5 of the 18 PLENA instances. On the instances where KernelCraft succeeds, the Contour-Explore agent's kernels are 2.86× (Gemmini) and 1.13× (PLENA) faster than KernelCraft's, and Contour-Explore spends 26.3% and 83.4% fewer tokens per correct kernel. The Contour-Explore agent's kernels are also 1.17× faster than Gemmini's library routine and 1.46× faster than those of the PLENA compiler. AccelOpt finds 1.50× faster kernels on Gemmini and 1.20× faster kernels on four PLENA attention instances when it searches through Contour-Explore instead of editing kernel programs. On 16 VTA instances, Contour-Ext extends the AutoTVM surface, and AutoTVM's XGB-Tuner finds 4.14× faster kernels on the extended surface; on PLENA, the extended surface makes the kernels found by hill climbing 1.20× faster. These results show that partial mapping specification with compiler completion lets neural and conventional search share one verified interface, and that this interface can be extended when search stalls.

A future direction is recursive improvement across data, models, and harnesses, inspired by MetaRSI [25]. The verified kernels, mapping decisions, and measured performance produced by Contour-Explore, together with the knobs accepted by Contour-Ext, could serve as post-training data. A post-trained model could then search more efficiently and propose further knobs; once these knobs pass the checks of Contour-Ext, they would expand subsequent searches and produce further training data.

References

[1] 2026. AccelOpt Source Code. https://github.com/zhang677/AccelOpt. https://github.com/zhang677/AccelOpt/commit/47e29a7c24704e3c3e534b5bcbf39cfac3f831e3 Commit 47e29a7, July 28, 2026: "Update reference".

[2] AICrossSim. 2026. PLENA Simulation System. GitHub repository. https://github.com/AICrossSim/PLENA_Simulator Accessed: 2026-07-31.

[3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. doi:10.1109/MM.2020.2996616

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 578–594.

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

[6] Yunji Chen, Huiying Lan, Zidong Du, Shaoli Liu, Jinhua Tao, Dong Han, Tao Luo, Qi Guo, Ling Li, Yuan Xie, and Tianshi Chen. 2019. An Instruction Set Architecture for Machine Learning. *ACM Trans. Comput. Syst.* 36, 3, Article 9 (Aug. 2019), 35 pages. doi:10.1145/3331469

[7] DeepSeek-AI. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. arXiv:2606.19348 [cs.CL] https://arxiv.org/abs/2606.19348

[8] Ruijie Gao, Haoran Jin, Jirong Yang, and Nathaniel Bleier. 2026. TensorLift: Automatic Extraction of Tensor-Level ISA Semantics from Accelerator RTL via MLIR Semantic Lifting. In *IEEE/ACM International Conference on Computer-Aided Design* (San Jose, CA, USA) *(ICCAD '26)*. Association for Computing Machinery, New York, NY, USA, 9 pages. doi:10.1145/3831252.3833946

[9] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2022. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *Proceedings of the 58th Annual ACM/IEEE Design Automation Conference* (San Francisco, California, United States) *(DAC '21)*. IEEE Press, 769–774. doi:10.1109/DAC18074.2021.9586216

[10] Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. 2025. Autocomp: A Powerful and Portable Code Optimizer for Tensor Accelerators. *arXiv preprint arXiv:2505.18574* (2025).

[11] Charles Hong, Sahil Bhatia, Altan Haan, Shengjun Kris Dong, Dima Nikiforov, Alvin Cheung, and Yakun Sophia Shao. 2024. LLM-Aided Compilation for Tensor Accelerators. In *2024 IEEE LLM Aided Design Workshop (LAD)*. 1–14. doi:10.1109/LAD62341.2024.10691748

[12] Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Gus Henry Smith, Thierry Tambe, Akash Gaonkar, Vishal Canumalla, Andrew Cheung, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. 2024. Application-level Validation of Accelerator Designs Using a Formal Software/Hardware Interface. *ACM Trans. Des. Autom. Electron. Syst.* 29, 2, Article 35 (Feb. 2024), 25 pages. doi:10.1145/3639051

[13] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. doi:10.1145/3519939.3523446

[14] Devansh Jain, Marco Frigo, Jai Arora, Akash Pardeshi, Zhihao Wang, Krut Patel, and Charith Mendis. 2025. TAIDL: Tensor Accelerator ISA Definition Language with Auto-generation of Scalable Test Oracles. In *Proceedings of the 2025 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 1316–1333. doi:10.1145/3725843.3756075

[15] Devansh Jain, Akash Pardeshi, Marco Frigo, Kaustubh Khulbe, Krut Patel, Saatvik Lochan, Jai Arora, and Charith Mendis. 2026. Automatically Generating ML Compiler Backends from Tensor Accelerator ISA Descriptions. *Proc. ACM Program. Lang.* OOPSLA2 (Oct. 2026). doi:10.1145/3839457

[16] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 http://arxiv.org/abs/1704.04760

[17] Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Ruichao Xiao, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, Zewei Jiang, Roman Levenstein, Dianshi Li, Liyuan Li, Ajit Mathews, Varna Puvvada, Feng Shi, Nathan Yan, Xiayu Yu, Uladzimir Pashkevich, Matt Steiner, Carole-Jean Wu, and Gaoxiang Liu. 2026. KernelEvolve: Scaling Agentic Kernel Coding for Heterogeneous AI Accelerators at Meta . In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 733–747. doi:10.1109/ISCA66397.2026.00063

[18] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. 2019. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro* 39, 5 (2019), 8–16.

[19] Jiayi Nie, Haoran Wu, Yao Lai, Zeyu Cao, Cheng Zhang, Binglei Lou, Erwei Wang, Jianyi Cheng, Timothy M. Jones, Robert D. Mullins, Rika Antonova, and Yiren Zhao. 2026. KernelCraft: Benchmarking for Agentic Close-to-Metal Kernel Generation on Emerging Hardware. In *Forty-third International Conference on Machine Learning*. https://openreview.net/forum?id=CG3TwM48M8

[20] OpenXLA. [n. d.]. StableHLO. https://openxla.org/stablehlo

[21] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: can LLMs write efficient GPU kernels?. In *Proceedings of the 42nd International Conference on Machine Learning* (Vancouver, Canada) *(ICML'25)*. JMLR.org, Article 1894, 60 pages.

[22] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 304–315.

[23] Taras Sereda, Tom St John, Burak Bartan, Natalie Serrino, Sachin Katti, and Zain Asgar. 2025. Kforge: Program synthesis for diverse ai hardware accelerators. *arXiv preprint arXiv:2511.13274* (2025).

[24] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems* 35 (2022), 35783–35796.

[25] Zihan Tan, Leixin Sun, Zitong Shi, Yitao Liu, Jiajun Wu, Nathaniel Brooks, Jiaru Qian, Xiaoran Shang, Suyuan Huang, Yi Ding, et al. 2026. MetaRSI/RSI2: A Meta-Recursive Self-Improving System for Recursive Self-Improving Systems Themselves. *arXiv preprint arXiv:2609.06396* (2026).

[26] Haoran Wu, Can Xiao, Jiayi Nie, Xuan Guo, Binglei Lou, Jeffrey T. H. Wong, Zhiwen Mo, Cheng Zhang, Przemyslaw Forys, Chengyang Ai, Timi Adeniran, Wayne Luk, Hongxiang Fan, Jianyi Cheng, Timothy M. Jones, Rika Antonova, Robert Mullins, and Aaron Zhao. 2026. Combating the Memory Walls: Optimization Pathways for Long-Context Agentic LLM Inference. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 100–115. doi:10.1109/ISCA66397.2026.00023

[27] Zihao Ye, Yingyi Huang, Hongyi Jin, Bohan Hou, Junru Shao, Zhongming Yu, Jinqi Chen, Meghan Cowan, Shiyi Cao, Shanli Xing, Hanfeng Chen, Vinod Grover, Tianqi Chen, and Luis Ceze. 2026. CAKE: Compiler-Agent Co-Design for Frontier Kernel Evolution. arXiv:2608.12629 [cs.LG] https://arxiv.org/abs/2608.12629

[28] Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. 2025. AccelOpt: A Self-Improving LLM Agentic System for AI Accelerator Kernel Optimization. arXiv:2511.15915 [cs.LG] https://arxiv.org/abs/2511.15915

[29] Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. 2025. CudaForge: An Agent Framework with Hardware Feedback for CUDA Kernel Optimization. arXiv:2511.01884 [cs.LG] https://arxiv.org/abs/2511.01884

[30] 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. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.