Rethinking Agentic Kernel Generation for Emerging Accelerators

Ruijie Gao*, Jirong Yang†, Barry Lyu*, Haoran Jin*, and Nathan Bleier*
*University of Michigan †The University of Texas at Austin
*{ruijieg,barrylyu,allenjin,nbleier}@umich.edu †polarisyjr@utexas.edu

***Abstract*—Emerging accelerators often lack mature compiler backends, motivating neural agents that generate and repair kernels from architectural documentation and simulator feedback. This approach repeatedly reconstructs workload-invariant machine semantics—including instruction behavior, legality constraints, synchronization rules, and memory protocols—for every workload. We argue that these semantics should be compiled once into a persistent symbolic artifact, while neural reasoning should focus on workload-dependent mapping decisions.**

We present ZOMBOSS, a compiler-mediated agentic framework for kernel generation that places neural search within a verified compiler boundary. ZOMBOSS compiles machine semantics and legality constraints into a reusable mapping interface, then uses a neural agent to optimize workload-dependent decisions within the validated mapping space. Across 20 Gemmini and 36 PLENA workload instances, ZOMBOSS returns a correct verified kernel on all 56 instances. Relative to the compiler default, ZOMBOSS achieves geometric-mean speedups of $3.34\times$ on Gemmini and $1.10\times$ on PLENA. Relative to direct agentic generation, it reduces inference tokens by 71.2% on Gemmini and 54.2% on PLENA.

These results show that a compiler-defined symbolic interface turns native kernel synthesis into verified design-space exploration: compiler infrastructure preserves legality and correctness, while neural guidance improves workload-specific performance with lower search cost and complete coverage.

I. INTRODUCTION

Programmable tensor accelerators often expose target-specific instruction set architectures, explicitly managed on-chip memories, and specialized data-movement and control mechanisms [4], [7], [15], [18]. Supporting such a target requires a compiler backend to lower tensor operations to these interfaces, including instruction selection, memory placement, state configuration, and legal instruction sequencing [8], [11]. However, most proposed tensor accelerators lack compiler backends with code-generation support, and rapid hardware iteration makes separately engineered backends difficult to maintain [9]–[11], [14]. Gemmini illustrates why programming-stack support matters for accelerator evaluation: system-level resource interactions and programming-stack inefficiencies can materially affect measured performance and efficiency [7]. When a reusable backend is unavailable, developers often fall back on manually written low-level kernels or target-specific compilation templates [14], [19].

Recent work has explored large language model (LLM)-based systems, including feedback-driven agents, for generating and optimizing hardware kernels across GPUs and specialized tensor accelerators [8], [9], [21], [23], [27]. Although GPU and accelerator kernel agents are often discussed together, they operate at different abstraction boundaries. GPU agents generate CUDA or GPU-DSL programs that a mature, reusable compiler stack lowers to executable device code [21], [27]. This compiler boundary fixes the machine-realization interface within which the agent explores workload-specific implementation and mapping choices; the agent does not reconstruct the GPU's native code-generation machinery for every workload. In contrast, when an emerging accelerator lacks a comparable backend, the agent must select a mapping and construct its native realization within the same loop. Thus, the success of GPU kernel agents does not eliminate the need for a compiler boundary; it presupposes one.

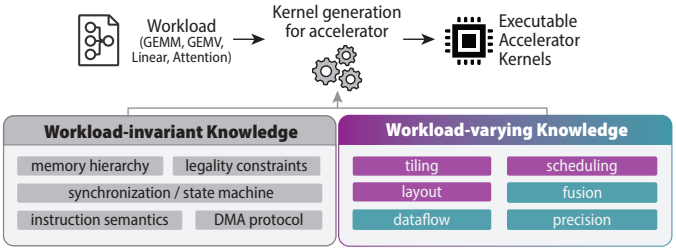


Fig. 1. Kernel generation combines reusable machine-interface knowledge with workload-dependent mapping decisions.

KernelCraft exemplifies agentic kernel generation for accelerators without a reusable compiler backend. Given a workload and machine specification, its agent uses compilation, simulation, and correctness feedback to iteratively generate and refine a low-level kernel for each workload [19]. Its results show that agents can produce functionally correct kernels for previously unseen accelerator instruction sets and can match or exceed template-based compiler baselines on some tasks. We refer to this per-workload, neural-agent-based workflow as *direct agentic kernel generation*. Viewed across workloads, it repeatedly combines two classes of information with different reuse properties, as illustrated in 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

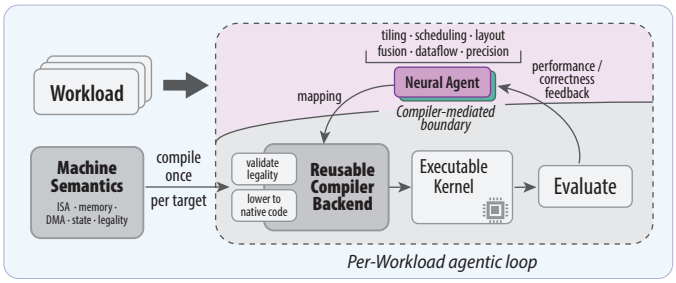


Fig. 2. Compiler-mediated agentic kernel generation compiles machine semantics once while searching workload-dependent mappings per workload.

across workload instances.

Direct agentic kernel generation handles both responsibilities within the same per-workload loop. The agent selects a mapping for the current workload while also constructing a native realization that obeys fixed machine rules. Consequently, 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. This coupling follows naturally from the no-backend setting: without a reusable implementation of the machine interface, the agent must account for both the machine and the workload.

Recent advances make this coupling optional. TensorLift recovers executable tensor-level machine semantics from RTL-derived models and explicit target profiles [6]. It represents these semantics using the Tensor Accelerator ISA Definition Language (TAIDL), which captures instruction behavior, architectural state, memory movement, and numerical operations in a machine-readable form [13]. The Accelerator Compiler Toolkit (ACT) consumes this specification to automatically generate a compiler backend [14]. Together, these systems provide an automated path from hardware semantics to reusable code generation, allowing workload-invariant machine knowledge to be compiled once rather than reconstructed within every agent invocation.

This technical shift makes the placement of neural reasoning an architectural choice rather than a constraint imposed by the no-backend setting. It also makes that choice experimentally accessible: the same representation of machine knowledge can support either direct native-code generation or neural search above a compiler backend. We therefore ask: should a per-workload agent continue to consume machine semantics and re-encode them in native code, or should the backend capture those semantics while the agent searches only workload-dependent mapping decisions? We argue for the latter. Our approach uses automatically generated backends to give emerging accelerators the structural advantage on which GPU agents already rely: a reusable, compiler-mediated abstraction boundary inside the agentic loop, as shown in Figure 2. Behind this boundary, the backend encodes target semantics, enforces legality constraints, and lowers selected mappings to native code; above it, the agent searches the workload-dependent mapping space.

To the best of our knowledge, this work is the first to identify the repeated reconstruction of workload-invariant machine semantics as a systems problem in direct agentic kernel generation and to formulate a compiler-mediated boundary that separates machine realization from workload-dependent mapping search.

To realize this boundary, we present ZOMBOSS, a compiler-mediated agentic framework that places neural search above an automatically generated backend. The backend exposes typed mapping decisions and owns symbolic completion, legality enforcement, and native code generation, while the agent uses verified correctness and performance feedback to search the mapping space.

This paper makes the following contributions:

- We identify the repeated reconstruction of workload-invariant machine semantics as a systems problem in direct agentic kernel generation, and distinguish reusable machine realization from workload-dependent mapping.
- We design ZOMBOSS, a compiler-mediated agentic framework that exposes typed mapping decisions, symbolically completes and legalizes agent proposals, and uses independently verified correctness and performance feedback to direct search.
- We develop two target-adaptive realizations of the compiler–agent interface for contrasting accelerator organizations: agent selection from compiler-enumerated legal mappings for Gemmini's compact systolic-array mapping space, and agent-directed region specification with conditional symbolic completion for PLENA's heterogeneous, multi-engine mapping space.
- We recast accelerator kernel generation from end-to-end native-program synthesis into compiler-bounded design-space exploration. The resulting symbolic programming interface opens hardware-aware kernel optimization to neural agents, heuristics, and conventional search algorithms through a common compiler-owned realization and trusted evaluation path.

We evaluate ZOMBOSS across 20 Gemmini and 36 PLENA workload instances. It returns a correct verified kernel on all 56 instances. Relative to the compiler default, it achieves geometric-mean speedups of $3.34\times$ on Gemmini and $1.10\times$ on PLENA. Relative to direct agentic generation, it reduces inference tokens by 71.2% on Gemmini and 54.2% on PLENA.

II. PRELIMINARIES

A. Emerging Accelerator Architectures

We study two representative accelerator platforms: Gemmini [7], an established state-of-the-art tensor-accelerator generator, and PLENA [25], a recently introduced open-source research accelerator for long-context agentic LLM inference. Both 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.

B. Automatic Compiler Backend Generation

We use *compiler backend* to mean the target-specific component that translates a tensor computation graph into accelerator assembly. For a tensor accelerator, this requires mapping tensor operators to legal instructions and realizing their data movement and memory placement. TensorLift, TAIDL, and ACT form an automated path for constructing this component from the hardware implementation.

TensorLift: RTL to tensor-level semantics. TensorLift removes the need to manually write the machine specification [6]. It lifts bit-accurate architectural-state updates extracted from RTL into structured tensor-level descriptions of computation, data movement, and control. The recovered semantics are validated against the low-level model using a combination of SMT proofs and simulation, connecting the compiler-visible interface back to the hardware implementation.

TAIDL: a reusable machine contract. TensorLift expresses this interface in the Tensor Accelerator ISA Definition Language (TAIDL), which describes accelerator state, instruction semantics, and legality constraints using tensor operations [13]. TAIDL therefore serves as a reusable contract between hardware and compiler generation. The published Gemmini artifact describes individual instructions for a functional oracle [12]; TensorLift additionally recovers macro-instruction semantics that let a compiler match an entire tensor operator, such as matrix multiplication or convolution [6].

TABLE I
REPEATED DOCUMENTATION RETRIEVAL ACROSS GEMMINI WORKLOAD TASKS.

Tool-trace metric	Value
Workload tasks	20
Tasks invoking `grep_docs`	17/20 (85.0%)
Tasks invoking `grep_docs` ≥ 2 times	15/20 (75.0%)
`grep_docs` calls / all tool calls	58/318 (18.2%)
Cross-family section re-retrieval	9/15 (60.0%)

ACT: tensor IR to native code. The Accelerator Compiler Toolkit (ACT) turns such an ISA specification into a target-specific compiler [14]. It combines equality saturation to match tensor computations with accelerator instructions and constraint solving to determine legal instruction ordering and memory placement. The resulting compiler is sound and complete relative to the supplied semantics and can enumerate equivalent programs for performance tuning.

Together, these systems show that the machine interface can be captured once and reused without sacrificing kernel performance. On seven Gemmini workloads, TensorLift+ACT achieves a $1.014\times$ geometric-mean speedup over the official handwritten kernels [6]. ACT also reports matching Gemmini library kernels when they are directly supported, outperforming hand-tuned Exo GEMM schedules by $1.10\times$ geometric mean with XLA autotuning, and outperforming compositions of hand-optimized library kernels by $1.48\times$ on composite workloads [14].

C. Direct Agentic Kernel Generation: KernelCraft

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.

D. Motivation

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

We reproduce KernelCraft's workflow on Gemmini [7] across 20 workload tasks drawn from four workload families. The target documentation and tool interface are fixed across the suite. As shown in Table I, 17 of the 20 tasks invoke `grep_docs`, and 15 invoke it multiple times. In total, the agent issues 58 documentation queries, accounting for 18.2% of all tool calls. Among successful matches, 15 workload-family–section accesses cover only six unique sections, yielding 60% cross-family re-retrieval. These traces make the repeated retrieval of target-invariant knowledge visible within the per-workload generation loop.

This observation motivates our compiler-mediated agentic design: a reusable compiler captures target-invariant code-generation knowledge, while the agent focuses on workload-specific mapping decisions.

III. Design

A. System Overview

Our framework turns per-workload kernel generation into feedback-directed mapping search above a reusable compiler backend. Given a tensor workload and a backend generated for the target, it searches for a high-performance mapping and returns an executable implementation that passes correctness validation. Machine realization remains inside the backend; neural reasoning is applied only to workload-dependent mapping choices. Figure 3 summarizes this organization.

The framework consists of six functional components. *The mapping-space generator* derives the mapping choices supported for the current workload, such as tiling, layout, placement, fusion, precision, and overlap. *The neural mapping agent* selects choices from this space using the workload and feedback from earlier candidates. *The symbolic mapping engine* converts the agent's proposal into one or more complete legal plans, filling unspecified decisions and enforcing target constraints. *The target code generator* lowers each plan to accelerator code. *The candidate evaluator* executes that code, verifies its output, and evaluates its performance using the target-specific objective. *The search orchestrator* connects

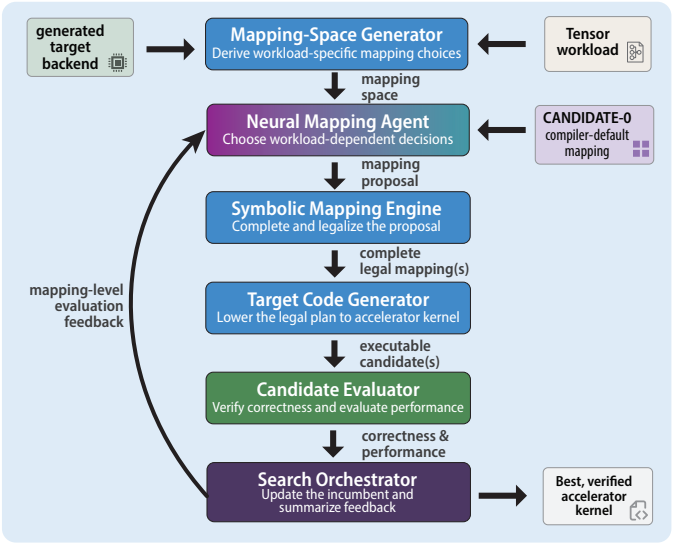


Fig. 3. Compiler-mediated mapping search separates workload-dependent neural decisions from compiler-owned completion and code generation, and closes the loop with independent correctness and performance feedback.

these components, maintains the best verified candidate, and constructs the feedback for the next agent decision.

These components form a closed search pipeline. The mapping-space generator first specializes the compiler's mapping interface to the input workload. The agent proposes a mapping, after which the symbolic mapping engine completes and legalizes it. The target code generator produces an executable candidate, and the evaluator returns both a correctness result and a performance measurement. The orchestrator updates the incumbent and feeds a compact result back to the agent, which either refines the current direction or explores another part of the mapping space. The loop begins from the compiler's default mapping and ends with the best verified candidate observed during search.

This organization is common to both accelerators we study, but the appropriate agent action differs with the structure of the target's mapping space. Gemmini's supported mapping space is small enough to enumerate, so its agent selects a complete mapping from a legal catalog. PLENA exposes interacting algorithm, memory, precision, and multi-engine scheduling decisions for which enumeration is impractical. Its agent instead describes a partial search region, and ACT completes that region into a small batch of legal plans.

B. The Compiler-Agent Mapping Contract

The framework connects its components through the five objects summarized in Table II. They form a typed progression from a workload definition to an evaluated program. In particular, an intent contains only choices that the compiler has explicitly made available through the mapping surface.

Workload contract. The workload contract fixes what is being optimized and how it is evaluated. CANDIDATE-0 and every searched mapping use the same operation, inputs, target, execution boundary, and correctness policy. Performance differences therefore come from mapping decisions rather than changes to the workload or evaluation conditions.

TABLE II
CORE OBJECTS IN COMPILER-MEDIATED MAPPING SEARCH.

Object	Role
Workload contract	Fixes the workload semantics, inputs, target, execution boundary, and correctness policy across all candidates.
Mapping surface	Publishes the typed, workload-applicable mapping decisions and their legal domains.
Agent intent	Selects a complete legal mapping or constrains a partial mapping region, depending on the target.
Resolved plan	Represents the complete legal mapping produced by the compiler for code generation.
Verified candidate	Associates an executable plan with its correctness result and measured performance.

Mapping surface. The mapping surface is the backend's public optimization contract. A decision appears on the surface only when it is supported by the generated backend and can affect the current workload. A matrix multiplication may expose tiling and accumulator initialization, whereas attention may additionally expose tensor residency, transposed views, softmax precision, and cross-engine overlap. The agent therefore searches meaningful mapping choices without manipulating physical instructions or addresses.

Intent and resolved plan. The agent's action is an intent grounded in the current surface. On an enumerable surface, an intent specifies every exposed decision. On a conditional surface, it can require some decisions, express preferences over others, and leave the remainder to ACT. In both cases, the result presented to code generation is a complete resolved plan. ACT fills omitted choices and ensures that the combined mapping satisfies target constraints. Consequently, an intent expresses *what mapping to seek*, while the resolved plan captures the complete legal mapping that the backend will realize.

Verified candidate. A resolved plan becomes a verified candidate after code generation, execution, and correctness validation. Its performance measurement can then be used for feedback and final selection.

Table III summarizes the resulting ownership boundary. The agent chooses the workload-dependent mappings that determine decomposition, reuse, precision, fusion, and overlap. The compiler completes these mappings, enforces their legality, and owns their native realization. The evaluator independently verifies correctness and measures performance.

C. Feedback-Directed Mapping Search

Algorithm 1 gives the target-independent search loop. The compiler first generates a workload-conditioned mapping surface and a default plan, CANDIDATE-0. CANDIDATE-0 is evaluated through the same path as searched plans, providing both the initial feedback and a verified fallback.

In each iteration, the agent sees the mapping surface, the default mapping, and summarized results from earlier candidates. Its proposal is passed to the symbolic mapping engine, which completes omitted choices, enforces target constraints, and returns only fresh legal plans. This completion step is target-specific: Gemmini produces one selected catalog plan, whereas PLENA produces two plans that test and diversify the requested region.

TABLE III
RESPONSIBILITY PARTITION IN THE FRAMEWORK.

Component	Functional responsibility
Agent	Chooses workload-dependent algorithm, tiling, layout, placement, fusion, precision, loop, and overlap decisions exposed by the surface.
Compiler	Generates the mapping surface, completes legal plans, and realizes them through instruction selection, memory management, synchronization, and target code generation.
Evaluator and orchestrator	Executes candidates, verifies correctness, evaluates performance, returns feedback, and selects the best verified result.

Algorithm 1 Compiler-mediated mapping search

Require: workload W
Ensure: best verified candidate b
$S \leftarrow \textsc{GenerateMappingSpace}(W)$
$p_0 \leftarrow \textsc{CompleteMapping}(\varnothing, S)$
$x_0 \leftarrow \textsc{GenerateCode}(p_0)$
$r_0 \leftarrow \textsc{EvaluateCandidate}(x_0, W)$
$b \leftarrow r_0$; $F \leftarrow \textsc{Summarize}(r_0)$
while CONTINUESEARCH **do**
  $I \leftarrow \textsc{AgentProposal}(W, S, F)$
  $P \leftarrow \textsc{CompleteAndLegalize}(I, S)$
  $R \leftarrow \emptyset$
  for all $p \in P$ **do**
    $x \leftarrow \textsc{GenerateCode}(p)$
    $r \leftarrow \textsc{EvaluateCandidate}(x, W)$
    $R \leftarrow R \cup \{r\}$
    if $r.verified \wedge \textsc{Better}(r, b)$ **then**
      $b \leftarrow r$
    end if
  end for
  $F \leftarrow \textsc{Summarize}(R, b)$
end while
return b

The evaluator returns correctness and performance for every generated candidate. Only verified results can update the incumbent. The feedback given to the agent contains these outcomes and the associated high-level mapping decisions, rather than native-code or simulator internals. The agent can then refine a promising mapping or explore another region. If a proposal cannot produce a legal fresh plan, the agent receives a mapping-level explanation and revises its next proposal. Final selection always returns the best verified candidate, not a result chosen directly by the agent.

D. Adapting Search Granularity to the Target

The abstraction boundary above does not require every target to use the same search granularity. It requires the agent to act on typed mapping decisions and the backend to own their native realization. We instantiate that boundary with selection over an enumerable catalog for Gemmini and symbolic completion of conditional regions for PLENA.

1) Gemmini: Enumerable Legal Mapping Selection: Gemmini uses a finite mapping surface for GEMV, GEMM, batch

matrix multiplication, and linear layers. Its exposed decisions are a macro tile, an optional compiler-supported tensor layout or logical view, and an accumulator strategy. A macro tile sets the M, N, and K tile sizes for one schedule region. ACT exposes only tiles compatible with the workload shape and the scratchpad and accumulator capacities. Layout choices preserve the workload's external tensor ABI; they do not ask the agent to implement a transpose or data-repacking kernel.

Tiling the K dimension produces a sequence of partial products. Before output conversion, each accumulator element satisfies

$$P_{m,n}^{(t)} = \sum_{k \in K_t} X_{m,k} W_{k,n},$$
$$A_{m,n} = I_{m,n} + \sum_{t=0}^{T_K - 1} P_{m,n}^{(t)}, \tag{1}$$

where I is the workload-defined initial value. Across the supported workloads, the compiler exposes three strategies for realizing this initialization, with each workload receiving only the applicable subset. The *explicit* strategy loads I before accumulating all K tiles. When $I = 0$, *zero-overwrite* initializes the accumulator with $P^{(0)}$ and accumulates the remaining tiles, avoiding a load of zeros. For a linear layer, *bias-broadcast* uses $I_{m,n} = bias_n$, loading the one-dimensional bias directly across the output rows. The backend translates the selected strategy into legal accumulator operations while preserving Equation 1.

For each workload, the compiler constructs a finite catalog containing the mappings that satisfy the exposed shape, capacity, and scheduling constraints. The agent can inspect the mapping decisions in this catalog, but it does not see the performance of an untried mapping.

A Gemmini iteration is therefore a selection step. The agent chooses one complete mapping from the legal catalog, and the evaluator returns its verification result, FPGA-measured cycle count, and structural properties. These properties help explain the measurement but do not score or rank the mapping. The next iteration can refine a promising tile and accumulator strategy or choose a mapping with different reuse and capacity characteristics.

2) PLENA: Conditional Symbolic Region Search: PLENA requires a different realization of the same boundary. A PLENA mapping can jointly choose an algorithm family, matrix and vector decompositions, tensor placement, SRAM residency, HBM prefetching, multi-engine overlap, precision, and loop structure. The valid domain of one choice often depends on another: for example, an online attention algorithm changes whether scores are materialized, which changes Vector SRAM demand and the legal block sizes. Each exposed decision has a finite typed domain, so the complete mapping space is finite in principle. In the PLENA workloads we evaluate, the resulting surfaces expose 8–18 free mapping knobs, and their corresponding intent spaces range from 11,664 to 27.75 billion combinations. Materializing such a space requires constructing this multiplicative cross-product and then applying joint legality and capacity checks to its members. Exhaustive enumeration is therefore infeasible as a per-workload search interface. Figure 4 summarizes the resulting conditional search path.

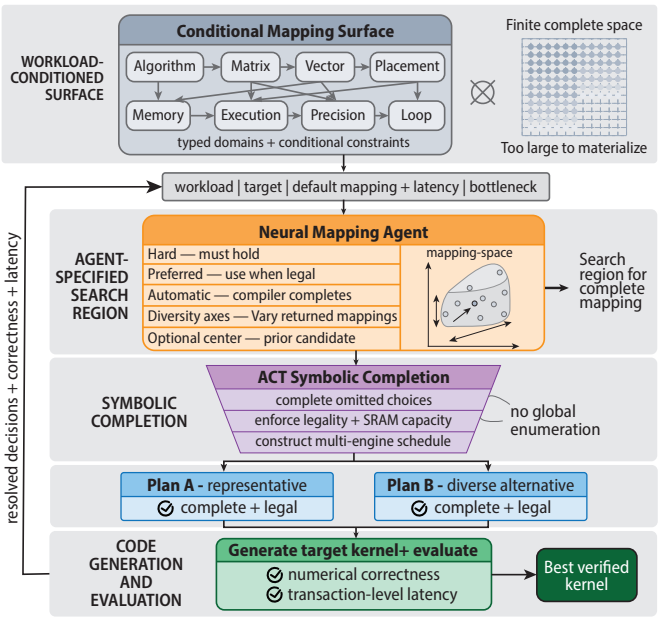


Fig. 4. PLENA conditional region search. The workload-conditioned surface compactly represents the available mapping decisions. The agent specifies a region, symbolic completion produces two complementary legal mappings, and evaluation feeds their realized decisions and performance into the next iteration.

ACT represents this space with a compact conditional surface for the current workload. The surface explicitly lists the supported decisions and their typed domains, rather than all complete mappings. It is organized into eight categories—algorithm, matrix, vector, placement, memory pipeline, execution pipeline, precision, and loop—and includes only decisions that apply to the workload. Thus, a GEMM surface omits softmax and head-grouping choices, while an attention surface can expose transposed Matrix-SRAM views, persistent and streamed operands, score materialization, and overlap across the HBM, Matrix, and Vector engines.

Before proposing a region, the agent observes the workload, target, conditional surface, default mapping and performance, and the current bottleneck. This provides enough context to choose a search direction without enumerating complete plans.

The agent describes a search region by assigning exposed decisions one of three roles. A *hard* decision must hold in every mapping completed from the region; ACT rejects the region when that value is incompatible with the other constraints. A *preferred* decision expresses the mapping that the agent expects to perform well, while allowing ACT to choose another value when the preference conflicts with legality or capacity. An *automatic* decision leaves the choice entirely to ACT.

Preferred decisions are the main mechanism through which neural guidance and symbolic completion cooperate. They let the agent state a performance hypothesis without having to make every interacting choice legal. For an HBM-bound attention mapping, for example, the agent can require online

TABLE IV
TARGET-SPECIFIC REALIZATIONS OF THE MAPPING BOUNDARY.

Property	Gemmini	PLENA
Representation	Enumerated legal catalog	Conditional surface
Agent action	Complete mapping	Constrained search region
Completion	One selected plan	Two completed plans
Iteration goal	Compare a legal point	Test and diversify a region
Primary metric	Board-measured cycles	Transactional latency

attention and a transposed Matrix-SRAM view, prefer larger tiles and persistent operands to improve reuse, and leave loop organization automatic. If the preferred tile and residency choices jointly exceed SRAM capacity, ACT can retain the legal parts of the proposal and select another value for the conflicting choice. The same conflict on a hard decision instead makes the region infeasible.

The agent can center the region on a promising mapping when refining an earlier result. It separately names *diversity axes*, which specify where the returned mappings should differ. Diversity controls how ACT samples the region; it does not impose an additional mapping constraint.

The symbolic mapping engine completes the free decisions and produces two legal mappings without materializing the global mapping space. One follows the agent's preferences as closely as the joint constraints allow; the other varies choices along the diversity axes. Evaluating both tests the agent's optimization hypothesis and a nearby alternative. Their realized decisions and performance direct the next iteration toward the better region or toward a different algorithm, placement, precision, or overlap policy.

E. Candidate Validation and Trusted Selection

The mapping interface prevents the agent from directly expressing illegal native code, but surface validation alone is not sufficient evidence that a candidate is correct. The framework therefore admits a plan to incumbent selection only after compiler legality checks, execution, numerical verification, and target-specific evidence collection.

Compiler legality. ACT first resolves all mapping decisions and checks their joint constraints. These checks cover tensor shapes, on-chip capacity, complete computation and output coverage, reductions, and cross-operation dependencies. The generated backend then owns instruction selection, memory management, synchronization, and target code generation. An invalid mapping is rejected before execution; the agent never repairs native code.

Numerical verification. Every candidate executes on the input bundle fixed by its workload contract, and the evaluator compares the complete output with an independent oracle. Data preparation required by a mapping remains inside the measured execution boundary. For PLENA mappings that vary precision, the workload contract fixes the acceptable numerical policy, so the agent cannot trade away correctness to improve performance.

Gemmini evidence and objective. Gemmini candidates additionally undergo a separate kernel-delimited dynamic trace. The trace establishes that the candidate consumes the workload inputs and produces the complete output through Gemmini execution rather than a host fallback.

Each evaluated mapping is compiled into a bare-metal executable and run on the Gemmini FPGA. The measurement harness uses `rdcycle` to delimit the complete kernel execution, including any view, repacking, or bias preparation required by the workload ABI and the Rocket-side command-issue overhead. The measured executable is the same program subjected to full-output verification and dynamic tracing, so correctness, provenance, and performance refer to the same mapping.

Gemmini's performance objective is the cycle count measured directly on the FPGA. This provides a direct hardware cycle measurement rather than an analytic or simulator-derived cost estimate. Feedback also reports factual structural properties of the evaluated schedule, including external-transfer bytes, Gemmini command counts, useful and padded MACs, operand reuse, and scratchpad and accumulator occupancy. These properties help the agent interpret why one mapping is faster than another.

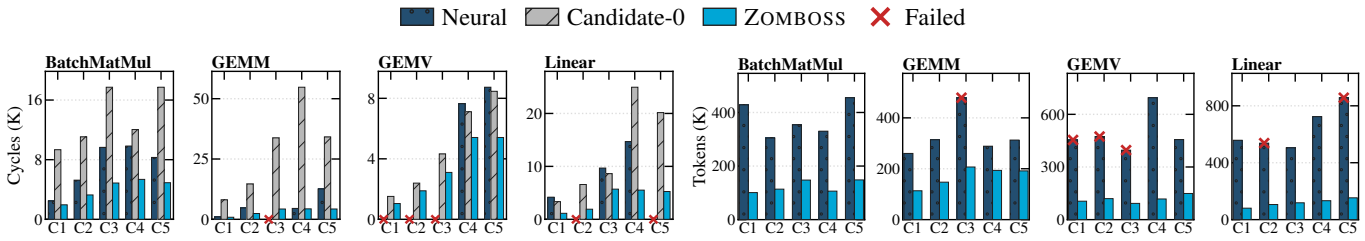
PLENA evidence and objective. A PLENA resolved plan is lowered to target code and executed by PLENA's event-driven transaction-level emulator [25]. The emulator models compute execution, instruction scheduling, and memory transactions at cycle granularity, including detailed HBM timing, and reports a cycle-approximate execution latency. We minimize this latency, which reflects the dependency stalls and overlap of the realized multi-engine schedule.

Trusted selection. CANDIDATE-0 and searched candidates follow the same target-specific path. After each evaluation, the controller replaces the incumbent only when the new candidate is verified and strictly improves the primary metric. Otherwise, the previous incumbent remains. Gemmini compares board-measured cycles, while PLENA compares transaction-level latency. The agent can propose mappings and interpret feedback, but it cannot change the correctness policy, performance metric, or final selection rule. Thus, neural reasoning influences which mappings are explored without controlling how their native code, correctness, or performance is judged.

IV. EVALUATION

A. Setup

Accelerator platforms. We evaluate ZOMBOSS on Gemmini and PLENA. For Gemmini, we use the `GemminiRocketConfig` provided by Chipyard [2], [7], 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



(a) Kernel cycles.

(b) Inference tokens.

Fig. 5. Gemmini results across 20 workload instances. Lower is better. A red cross marks a Neural run that produced no correct kernel. ZOMBOSS achieves the lowest cycle count on every instance and uses fewer tokens than Neural. CANDIDATE-0 requires no model inference.

Gemmini performance measurements from this physical implementation.

For PLENA, we use the authors' official RTL release and the transaction-level mode of their official simulator [1], [25]. The simulator executes compiled assembly against an explicit memory layout and produces both numerical outputs and simulated latency. Following KernelCraft's PLENA evaluation methodology [19], we use the simulator-reported cycle count as the performance metric.

Model and workloads. All agentic experiments use DeepSeek-V4-Flash [5]. Across both accelerators, we adopt the applicable benchmark definitions and input shapes published by KernelCraft [19]. We refer to each workload–input-shape combination as a *workload instance*. For each workload instance, Neural and ZOMBOSS receive the same maximum number of agent iterations, giving both methods an equal search budget.

B. Gemmini Results

Comparison. We compare three methods across 20 instances from BatchMatMul, GEMM, GEMV, and Linear. *Neural* follows KernelCraft's direct agentic workflow and generates a native kernel for each instance. CANDIDATE-0 is the compiler's default legal mapping and requires no model inference. ZOMBOSS reports the best verified mapping found by its feedback-directed search. Every method uses the same workload inputs, correctness oracle, and execution boundary. For a successful Neural run, we report the correct kernel with the lowest cycle count. We report the token usage of the complete inference trajectory for both successful and failed runs.

Mapping search improves performance beyond the verified default. CANDIDATE-0 produces a correct executable mapping for every instance, so its gap to ZOMBOSS isolates the optimization headroom available after legality and code generation have been handled. Figure 5a shows that this headroom depends strongly on the workload family. ZOMBOSS's geometric-mean speedup over CANDIDATE-0 is $8.54\times$ for GEMM, $3.44\times$ for BatchMatMul, $3.06\times$ for Linear, and $1.40\times$ for GEMV. The matrix–matrix workloads expose the largest benefit from workload-specific tiling and reuse decisions. Across the complete suite, ZOMBOSS improves every instance and achieves a $3.34\times$ geometric-mean speedup over CANDIDATE-0.

The same figure reveals a second effect: direct native-kernel generation has uneven reliability across workload families. Neural succeeds on all five BatchMatMul instances, four of five GEMMs, two of five GEMVs, and three of five Linear instances. ZOMBOSS retains a correct verified incumbent on all 20 instances. It also achieves a lower cycle count than Neural's best correct kernel in each of Neural's 14 successful runs, with a $1.79\times$ geometric-mean speedup. The compiler boundary therefore supplies complete workload coverage while preserving enough mapping freedom to outperform both the default mapping and direct agentic generation.

Focused mapping search reduces inference cost. Figure 5b shows that ZOMBOSS consumes 132K tokens per instance on average, compared with 459K for Neural, a 71.2% reduction. Neural's six failed trajectories average 533K tokens, 24.6% more than its successful trajectories at 428K. These failures are expensive searches: the direct workflow continues spending inference on a native realization and finishes without a usable kernel.

The largest savings occur on Linear (81.4%) and GEMV (76.4%), the two lowest-success families for Neural. The pattern links reliability and cost: direct generation spends its largest budgets in the families it handles least reliably. The shorter ZOMBOSS trajectories also produce the fastest kernel on every instance. Reusing the compiler's machine realization focuses inference and performance feedback on mapping choices, giving each trajectory a smaller search responsibility and a verified result.

C. PLENA Results

Comparison. We compare the same three methods across 36 instances from 12 workload families spanning matrix, normalization, activation, and attention computations. Neural again generates a native kernel for each instance, whereas CANDIDATE-0 and ZOMBOSS use the reusable PLENA backend. For Neural, we report the correct kernel with the lowest cycle count and the token usage of the complete trajectory whether or not it succeeds. For ZOMBOSS, we report the lowest-cycle verified result among three search runs and the token usage of the trajectory that produced that result.

Symbolic coverage exposes optimization headroom. Figure 6(a) shows that ZOMBOSS improves CANDIDATE-0 on 34 of 36 instances and matches it on the remaining two, achieving

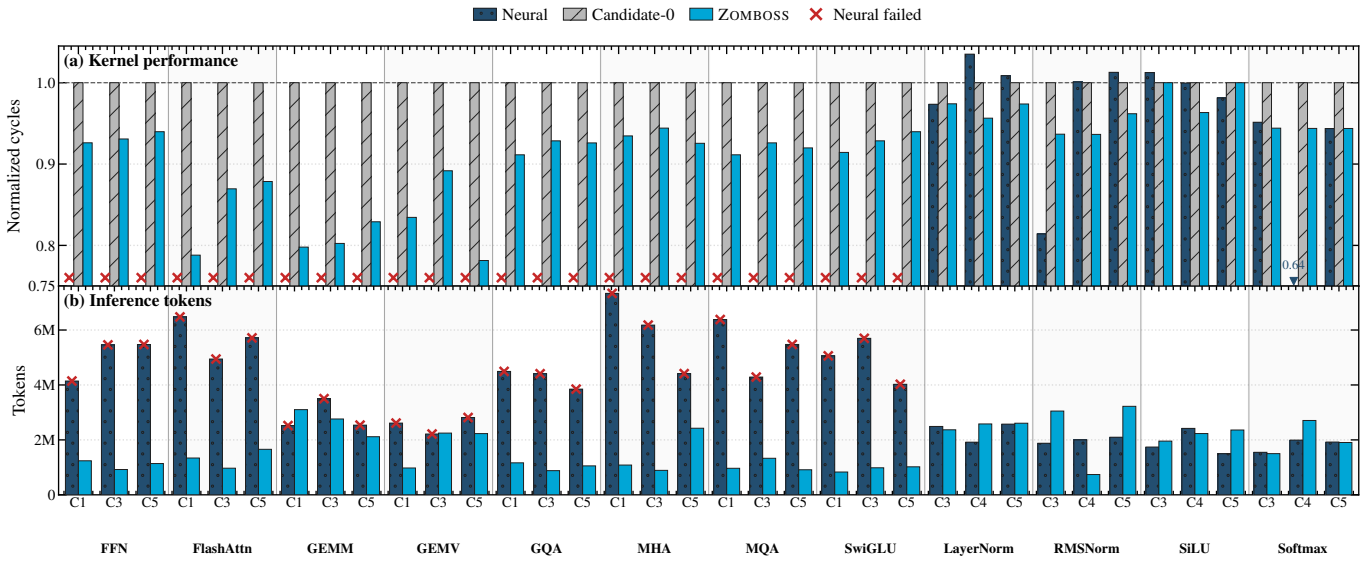


Fig. 6. PLENA results across 36 workload instances. Panel (a) reports kernel cycles normalized to CANDIDATE-0; lower is better. Panel (b) reports complete-trajectory inference tokens. A red cross marks a Neural run that produced no correct kernel. For ZOMBOSS, both panels report the lowest-cycle result among three search runs, with tokens taken from the same run. CANDIDATE-0 requires no model inference.

a $1.10\times$ geometric-mean speedup across the suite. The largest family-level gains occur on GEMM ($1.24\times$), GEMV ($1.20\times$), and FlashAttention ($1.18\times$), where tiling, data placement, and reuse create substantial mapping headroom beyond the conservative default.

The same figure reveals a second effect: direct native-kernel generation has much lower coverage. Neural produces a correct kernel on all three LayerNorm, RMSNorm, SiLU, and Softmax instances, but on none of the 24 instances from the other eight families. This coverage gap coincides with mapping-surface complexity. All 12 Neural successes occur on instances exposing at most ten free mapping knobs, whereas Neural fails on all 21 instances exposing more than ten. ZOMBOSS improves CANDIDATE-0 on every one of these 21 higher-dimensional instances, achieving a $1.11\times$ geometric-mean speedup, while retaining a correct verified incumbent on all 36 instances. On the 12 instances where Neural succeeds, the two methods are performance-competitive: ZOMBOSS is faster on seven and Neural on five, while Neural is $1.02\times$ faster in geometric mean. This comparison is necessarily conditioned on Neural success. On the other 24 instances, ZOMBOSS still achieves a $1.12\times$ geometric-mean speedup over CANDIDATE-0, converting the portion of the suite with no Neural result into both complete coverage and measurable optimization.

Focused search avoids costly failed trajectories. Figure 6(b) shows that ZOMBOSS consumes 1.71M tokens per instance on average, compared with 3.72M for Neural, a 54.2% reduction. Its average inference cost is likewise lower, at \$0.070 per instance versus \$0.201 for Neural, a 65.1% reduction.

The savings are concentrated in the instances that direct generation cannot complete. Across Neural's 24 failed trajectories, ZOMBOSS uses 1.43M tokens per instance versus 4.58M for Neural, a 68.9% reduction, while still returning a verified kernel. On the 12 successful Neural instances, Neural instead uses fewer tokens, 2.01M versus 2.27M. Thus, PLENA's suite-level savings do not come from shortening every search. They come from reusing the backend's machine realization on the instances where a direct agent otherwise spends a long trajectory without producing a usable result.

D. Cross-Platform Summary

Across both targets, ZOMBOSS's benefits are consistent along three dimensions:

- **Performance over the symbolic baseline.** ZOMBOSS achieves a $3.34\times$ geometric-mean improvement over CANDIDATE-0 on Gemmini and a $1.10\times$ improvement on PLENA. It improves 54 of the 56 instances and matches the baseline on the remaining two.
- **Token efficiency.** Relative to Neural, ZOMBOSS reduces full-suite token usage by 71.2% on Gemmini and 54.2% on PLENA.
- **Complete correctness coverage.** ZOMBOSS returns a correct verified kernel on 20/20 Gemmini and 36/36 PLENA instances, achieving 100% coverage on both platforms. Neural succeeds on only 14/20 (70.0%) and 12/36 (33.3%), respectively—26/56 (46.4%) instances overall.

Together, these results show that placing neural search inside a verified compiler boundary consistently improves the symbolic starting point while reducing search cost and preserving complete workload coverage.

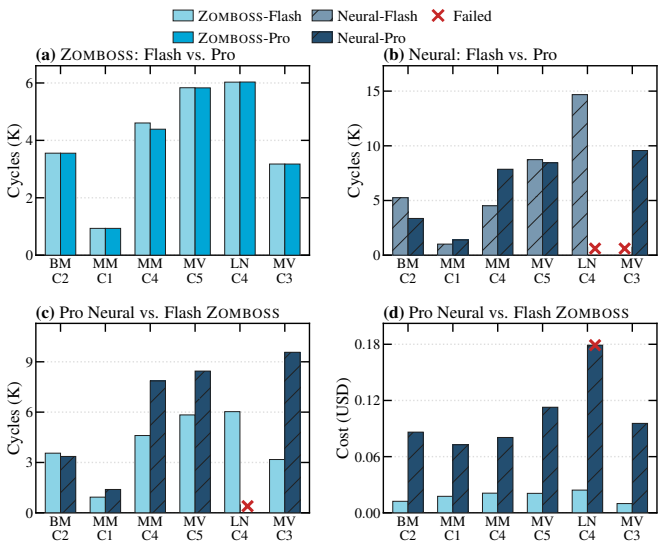


Fig. 7. Gemmini model-capability ablation. (a) ZOMBOSS: Flash vs. Pro. (b) Neural: Flash vs. Pro. (c, d) Pro Neural vs. ZOMBOSS-Flash in cycles and API cost, respectively. Crosses indicate failed runs. BM, MM, MV, and LN abbreviate BatchMatMul, GEMM, GEMV, and Linear.

V. DISCUSSION

A. Sensitivity to Model Capability

An alternative explanation for our results is that Neural is disadvantaged by an insufficiently capable model. Existing accelerator-kernel benchmarks do not establish a monotonic capability ordering, as task completion and optimization quality are not perfectly correlated [19]. We therefore perform a controlled within-family ablation, replacing Flash with Pro, which DeepSeek-V4 positions as the stronger configuration for challenging tasks [5].

We construct a capability-stratified set of six Gemmini instances from the Flash results. The set includes an easy control (BatchMatMul C2), two intermediate instances (GEMM C1 and C4), a difficult but solvable boundary case (GEMV C5), an instance near Flash's success boundary (Linear C4), and a ceiling probe on which Flash fails (GEMV C3). For a uniform one-result-per-instance comparison, we report the first run (seed 1) for each model–method pair. Flash and Pro candidates are replayed using the same board timing harness. Figure 7 summarizes the results; each bar reports that run, and crosses directly denote failed Neural trajectories.

The compiler-mediated result is insensitive to the model upgrade. For ZOMBOSS, five of the six instances differ by at most 0.1% between Flash and Pro. Pro improves GEMM C4 by 4.7%, but its geometric-mean cycle count across all six instances is only 0.8% lower, and both models retain a verified result throughout. In contrast, substituting Pro raises the geometric-mean API cost by 17.0×. Thus, on Gemmini's enumerable mapping spaces, Flash already crosses the capability threshold needed to guide the search. Beyond this point, the symbolic mapping engine, compiler-owned realization, and verified incumbent selection determine the returned kernel more strongly than additional model capacity.

Additional capability affects direct generation, but not monotonically. Pro recovers a correct direct kernel for GEMV C3, where Flash fails, showing that additional capability can move the success boundary. It does not uniformly improve the task, however: Pro fails on Linear C4, which Flash solves. Among the four instances that both Neural variants solve, Pro is faster on BatchMatMul C2 and GEMV C5 but slower on both GEMM instances; its geometric mean cycle count is 10.2% higher than Flash's. A stronger general model can therefore solve a previously failed case without inducing a monotonic change in kernel correctness or quality.

A stronger direct Neural does not close the gap. The Pro Neural succeeds on five of six instances, whereas ZOMBOSS-Flash returns a verified kernel for all six. Among the five successful Neural runs, ZOMBOSS-Flash is faster on four and achieves a 1.60× geometric-mean speedup. Pro Neural's only win is 5.5%, while ZOMBOSS's largest advantage is 3.01×. Moreover, Pro Neural costs 5.92× more than ZOMBOSS-Flash in geometric mean across all six trajectories, including the money spent on the failed Linear C4 trajectory.

These results separate the value of neural guidance from the value of a larger model. Neural guidance remains useful for proposing performance-oriented mapping directions, as shown by ZOMBOSS's improvement over CANDIDATE-0, but once the model is sufficiently capable, upgrading it produces no measurable kernel-quality benefit inside the symbolic boundary. We therefore use Flash for the main evaluation: it preserves ZOMBOSS's correctness and performance at substantially lower inference cost. More broadly, these results suggest that a symbolic boundary can make kernel quality less sensitive to model capability once the model can navigate the exposed mapping space. This conclusion is specific to the finite Gemmini mapping surfaces evaluated here.

B. Neural Selection versus Random Sampling

Gemmini exposes a finite legal mapping catalog, a natural alternative is to sample mappings uniformly and retain the fastest measured candidate. We ask whether Neural's adaptive selection improves over this baseline under the same evaluation budget.

We evaluate both policies on all 20 Gemmini instances. Random samples legal candidates uniformly without replacement. Neural uses the measured cycles and structural observations from previous trials to choose each subsequent candidate. Both policies use the same exhaustively enumerated and verified catalog, share the same compiler-default baseline, and receive at most 12 unique candidate evaluations. Values and ranks of untried candidates remain hidden, and the catalog is independently permuted on every Neural turn. The harness returns the fastest candidate that each policy actually evaluates. The sole experimental variable is therefore whether previous measurements direct the remaining evaluation budget.

Neural selection reduces tail regret. With measured board cycles as feedback, Neural reaches the known exhaustive optimum on all 20 instances, compared with 14 for Random. Under the 0.25% measurement dead band, Neural wins six

head-to-head comparisons and ties the remaining 14; Random's worst-case regret is 4.89%. All six Neural wins occur in catalogs containing 31–55 candidates, where the 12-evaluation budget covers only 21.8–38.7% of the legal surface. These results show that adaptive selection converts the same compile-and-measure budget into more effective coverage and removes the observed tail of missed mappings. We therefore use Neural to direct mapping selection. These results are scoped to the small enumerable catalogs evaluated here; larger search spaces require separate evaluation.

C. Implications for Agentic Design-Space Exploration

The broader implication of ZOMBOSS is a new programming interface for hardware-aware kernel programming and optimization. Direct agentic generation asks a policy to produce a complete native program, coupling workload mapping with instruction selection, memory management, synchronization, and other fixed machine rules. In ZOMBOSS, the mapping surface, symbolic completion, and verified evaluator instead expose a typed interface from workload-level decisions to measured executable candidates. This interface is not intrinsically neural: an LLM agent, a heuristic, or a conventional search algorithm can all propose mapping intents while using the same compiler-owned realization and trusted selection path. Accelerator kernel generation can therefore be reformulated from end-to-end synthesis of a native program into design-space exploration bounded by a compiler-defined symbolic interface.

This reformulation separates the intelligence used to explore a space from the infrastructure that defines and validates it. A search policy need not learn how to encode a legal DMA sequence or maintain configuration state before it can test a tiling, placement, precision, or overlap hypothesis. It instead receives workload-conditioned decision domains, proposes a point or region, and obtains correctness, performance, and structural feedback for the legal plans completed by the compiler. Different proposal policies can consequently be compared or combined without changing the workload contract, native code generator, evaluator, or correctness boundary. The symbolic interface is thus both a correctness boundary and a reusable DSE substrate.

It also opens a direct path toward a stronger evolutionary harness for agentic DSE. AlphaEvolve demonstrates how an archive of evaluated programs, automated fitness, and LLM-generated variations can sustain long-horizon algorithmic search [20]; ShinkaEvolve further improves this pattern with exploration-aware parent selection, novelty filtering, adaptive model selection, and accumulated search knowledge [17]. A ZOMBOSS harness could apply these mechanisms at two levels: evolve verified mapping intents directly, or evolve heuristics that generate intents, rather than evolving unrestricted native programs. It could maintain an archive of evaluated mappings, select parents by performance and diversity, mutate or recombine typed decisions, and let symbolic completion legalize each offspring before execution. Such a design would preserve ZOMBOSS's compiler boundary while allowing evolutionary, heuristic, and neural operators to share experience and explore larger conditional mapping spaces more efficiently.

VI. RELATED WORK

Mapping-space search and autotuning. Tensor compilers such as TVM, Ansor, and MetaSchedule and accelerator mapping tools such as Timeloop and MAESTRO share the same basic abstraction: a manually designed scheduling or mapping space is explored by a search policy and scored by execution, an analytical model, or a learned surrogate [3], [16], [22], [24], [28]. ZOMBOSS retains this separation between space and policy, but derives the executable backend and its workload-conditioned legal surface from machine specifications, then verifies realized kernels on the target path.

Agentic accelerator kernel generation. A separate line of work asks LLM agents to write and optimize accelerator kernels themselves. KernelCraft generates close-to-metal programs for emerging accelerators, and AutoComp uses execution feedback to optimize code for tensor accelerators [8], [19]. AccelOpt directly rewrites Trainium NKI kernels using measured profiles, beam search, and an optimization memory distilled from slow–fast kernel pairs [26]. Together, these systems show that agents can produce correct, high-performance kernels for specialized accelerators, not only GPUs. They keep kernel construction and machine realization inside the agentic loop. ZOMBOSS instead asks the agent for uses a generated backend to complete legal choices and emit native code, making the machine realization reusable across workloads.

VII. CONCLUSION

We present ZOMBOSS, a compiler-mediated framework that separates reusable machine realization from per-workload mapping search for emerging accelerators. A generated backend exposes a workload-conditioned mapping surface, completes and legalizes proposed mappings, and lowers them to native code. A neural agent uses verified correctness and performance feedback to direct exploration. Across 20 Gemmini and 36 PLENA workload instances, ZOMBOSS returns a correct verified kernel for all 56 instances; direct Neural succeeds on 26. ZOMBOSS improves CANDIDATE-0 on 54 instances and matches it on two, with geometric-mean speedups of $3.34\times$ on Gemmini and $1.10\times$ on PLENA. It also reduces inference tokens relative to Neural by 71.2% and 54.2%, respectively. These results establish a compiler-defined symbolic interface as an effective boundary for agentic kernel generation. The boundary turns native kernel synthesis into verified design-space exploration and exposes the same target realization to neural, heuristic, evolutionary, and conventional search policies.

REFERENCES

[1] AICrossSim, "PLENA simulation system," GitHub repository, 2026, accessed: 2026-07-31. [Online]. Available: https://github.com/AICrossSim/PLENA_Simulator

[2] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[3] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 578–594.

[4] Y. Chen, H. Lan, Z. Du, S. Liu, J. Tao, D. Han, T. Luo, Q. Guo, L. Li, Y. Xie, and T. Chen, "An instruction set architecture for machine learning," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3331469

[5] DeepSeek-AI, "Deepseek-v4: Towards highly efficient million-token context intelligence," 2026. [Online]. Available: https://arxiv.org/abs/2606.19348

[6] R. Gao, H. Jin, J. Yang, and N. Bleier, "Tensorlift: Automatic extraction of tensor-level isa semantics from accelerator rtl via mlir semantic lifting," in *IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '26. New York, NY, USA: Association for Computing Machinery, 2026. [Online]. Available: https://doi.org/10.1145/3831252.3833946

[7] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual ACM/IEEE Design Automation Conference*, ser. DAC '21. IEEE Press, 2022, p. 769–774. [Online]. Available: https://doi.org/10.1109/DAC18074.2021.9586216

[8] C. Hong, S. Bhatia, A. Cheung, and Y. S. Shao, "Autocomp: A powerful and portable code optimizer for tensor accelerators," *arXiv preprint arXiv:2505.18574*, 2025.

[9] C. Hong, S. Bhatia, A. Haan, S. K. Dong, D. Nikiforov, A. Cheung, and Y. S. Shao, "Llm-aided compilation for tensor accelerators," in *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–14.

[10] B.-Y. Huang, S. Lyubomirsky, Y. Li, M. He, G. H. Smith, T. Tambe, A. Gaonkar, V. Canumalla, A. Cheung, G.-Y. Wei, A. Gupta, Z. Tatlock, and S. Malik, "Application-level validation of accelerator designs using a formal software/hardware interface," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 2, Feb. 2024. [Online]. Available: https://doi.org/10.1145/3639051

[11] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 703–718. [Online]. Available: https://doi.org/10.1145/3519939.3523446

[12] D. Jain, M. Frigo, J. Arora, A. Pardeshi, Z. Wang, K. Patel, and C. Mendis, "Artifact of taidl: Tensor accelerator isa definition language with auto-generation of scalable test oracles," Aug. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16934755

[13] D. Jain, M. Frigo, J. Arora, A. Pardeshi, Z. Wang, K. Patel, and C. Mendis, "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*, ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, Oct. 2025, p. 1316–1333. [Online]. Available: https://doi.org/10.1145/3725843.3756075

[14] D. Jain, A. Pardeshi, M. Frigo, K. Khulbe, K. Patel, S. Lochan, J. Arora, and C. Mendis, "Automatically generating ml compiler backends from tensor accelerator isa descriptions," *Proc. ACM Program. Lang.*, no. OOPSLA2, Oct. 2026. [Online]. Available: https://doi.org/10.1145/3839457

[15] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: http://arxiv.org/abs/1704.04760

[16] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 754–768.

[17] R. T. Lange, Y. Imajuku, and E. Cetin, "Shinkaevolve: Towards open-ended and sample-efficient program evolution," *arXiv preprint arXiv:2509.19349*, 2025.

[18] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin *et al.*, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.

[19] J. Nie, H. Wu, Y. Lai, Z. Cao, C. Zhang, B. Lou, E. Wang, J. Cheng, T. M. Jones, R. D. Mullins, R. Antonova, and Y. Zhao, "Kernelcraft: Benchmarking for agentic close-to-metal kernel generation on emerging hardware," in *Forty-third International Conference on Machine Learning*, 2026. [Online]. Available: https://openreview.net/forum?id=CG3TwM48M8

[20] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog, "Alphaevolve: A coding agent for scientific and algorithmic discovery," 2025. [Online]. Available: https://arxiv.org/abs/2506.13131

[21] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini, "Kernelbench: can llms write efficient gpu kernels?" in *Proceedings of the 42nd International Conference on Machine Learning*, ser. ICML'25. JMLR.org, 2025.

[22] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[23] T. Sereda, T. S. John, B. Bartan, N. Serrino, S. Katti, and Z. Asgar, "Kforge: Program synthesis for diverse ai hardware accelerators," *arXiv preprint arXiv:2511.13274*, 2025.

[24] J. Shao, X. Zhou, S. Feng, B. Hou, R. Lai, H. Jin, W. Lin, M. Masuda, C. H. Yu, and T. Chen, "Tensor program optimization with probabilistic programs," *Advances in Neural Information Processing Systems*, vol. 35, pp. 35 783–35 796, 2022.

[25] H. Wu, C. Xiao, J. Nie, X. Guo, B. Lou, J. T. H. Wong, Z. Mo, C. Zhang, P. Forys, C. Ai, T. Adeniran, W. Luk, H. Fan, J. Cheng, T. M. Jones, R. Antonova, R. Mullins, and A. Zhao, "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, 2026, pp. 100–115. [Online]. Available: https://doi.org/10.1109/ISCA66397.2026.00023

[26] G. Zhang, S. Zhu, A. Wei, Z. Song, A. Nie, Z. Jia, N. Vijaykumar, Y. Wang, and K. Olukotun, "Accelopt: A self-improving llm agentic system for ai accelerator kernel optimization," 2025. [Online]. Available: https://arxiv.org/abs/2511.15915

[27] Z. Zhang, R. Wang, S. Li, Y. Luo, M. Hong, and C. Ding, "Cudaforge: An agent framework with hardware feedback for cuda kernel optimization," 2025. [Online]. Available: https://arxiv.org/abs/2511.01884

[28] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating {High-Performance} tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.