VERILOCC: End-to-End Cross-Architecture Register Allocation via LLM

Anonymous ACL submission

Abstract

Modern GPUs evolve rapidly, yet production compilers still rely on hand-crafted register allocation heuristics that require substantial retuning for each hardware generation. We introduce VERILOCC, a framework that combines large language models (LLMs) with formal compiler techniques to enable generalizable and verifiable register allocation across GPU architectures. VERILOCC fine-tunes an LLM to translate intermediate representations (MIRs) into target-specific register assignments, aided by static analysis for cross-architecture normalization and generalization and a verifier-guided regeneration loop to ensure correctness. Evaluated on matrix multiplication (GEMM) and multi-head attention (MHA), VERILOCC achieves 85–99% single-shot accuracy and near-100% pass@100. Case study shows that VERILOCC discovers more performant assignments than expert-tuned libraries, outperforming rocBLAS by over 10% in runtime.

1 Introduction

Modern GPUs have dramatically reshaped deep learning by offering massive parallel computational powers (Krizhevsky et al., 2012). Unleashing this performance requires not only hardware advances (Jouppi et al., 2017; Wang et al., 2021; Zhao et al., 2025), but also increasingly sophisticated software stacks (Chen et al., 2018; LI et al., 2023; Ma et al., 2020; NVIDIA Corporation, 2024; Wu et al., 2025; Zheng et al., 2020). At the core are optimizing compilers, which translate high-level GPU kernels into efficient, hardware-specific binaries.

Many of these optimizations are NP-complete; a central example is *register allocation* (Alfred et al., 2007), as illustrated in Figure 1. The task involves assigning virtual registers in the compiler's intermediate representation (MIR) to physical registers in the instruction set architecture (ISA). A *correct* allocation must (1) consistently map each virtual register to the same physical register, and (2) assign virtual registers with overlapping lifetimes to disjoint physical registers.

Achieving *performant* register allocation is even harder, as it requires modeling architectural details such as register bank conflicts, pipeline stalls, and memory spill costs. Modern compilers rely on hand-crafted or learned heuristics (Lozano et al., 2019; Quintão Pereira and Palsberg, 2008), while performance-critical libraries such as BLAS (AMD Inc., 2024a) often resort to handwritten assembly optimized for specific GPUs. Both approaches demand substantial engineering effort and are difficult to retarget across hardware generations.

Recent work has explored learning-based compiler optimizations, but many rely on handcrafted features (Chen et al., 2021) or lack correctness guarantees from static analysis or formal verification (Liu et al., 2024). General-purpose models like ChatGPT also struggle with register allocation, often producing incorrect sequential assignments that ignore liveness constraints (see Appendix B). This motivates our approach: combining the expressiveness of large language models (LLMs) (OpenAI, 2023; Qwen et al., 2025) with formal compiler techniques to achieve correctness and cross-architecture generalization.

This paper introduces VERILOCC, a learning-based register allocator that formulates register assignment as an end-to-end sequence-to-sequence (seq2seq) translation task (Sutskever et al., 2014) task for LLM. Our key insight is that while GPU architectures differ in low-level details, they share core traits, such as in-order pipelines and sharded register banks, which persist across vendors and generations. VERILOCC fine-tunes an LLM to translate MIR into target-specific register assignments, treating different MIRs as dialects of a shared computational language. Inspired by neural machine translation (Sutskever et al., 2014; Bahdanau et al., 2015; Vaswani et al., 2017), VER-

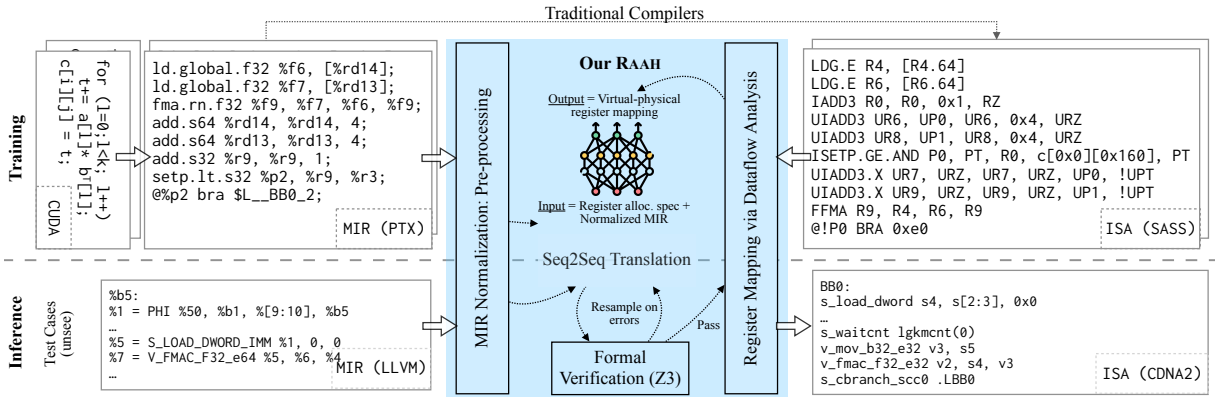

Figure 1: Overall training and inference workflow of VERILOCC. Our seq2seq formulation learns to translate normalized MIR into virtual-to-physical register mappings, derived from MIR and ISA through dataflow analysis that tracks value propagation. For clarity, we show only the MIR (e.g., PTX) and ISA (e.g., SASS) fragments that load matrix A from memory and perform dot-product accumulation.

ILOCC learns to map virtual to physical registers while adapting to the syntactic and semantic variations of diverse GPU toolchains.

To be practical, VERILOCC must overcome three key challenges. First, compiler transformations such as inlining and loop unrolling (Alfred et al., 2007) can inflate MIR size beyond the LLM's effective context window (Hsieh et al.); for instance, a single multi-head attention (MHA) kernel can exceed 50,000 tokens. Second, the model must generalize across architectures while respecting hardware-specific constraints like register file sizes and reserved registers. Third, correctness is critical – an invalid allocation may silently corrupt a program or render it unexecutable.

VERILOCC addresses these challenges via a combination of normalization and verification, as illustrated in Figure 1. We use static analysis to normalize both MIR and ISA representations while preserving program semantics (Lattner et al., 2007; Xie and Aiken, 2007). As shown in Figure 2, MIRs from different compilers are transformed into a unified format, and register mappings are extracted via dataflow analysis as JSON-style dictionaries. This reduces token length by 80–90% in our experiments (see Table 1), improving LLM reasoning efficiency and allowing the model to focus on allocation logic. Normalization also enables the creation of a heterogeneous training dataset that combines compiler-generated outputs from multiple toolchains with expert-optimized libraries—teaching the model both general strategies and target-specific heuristics. To ensure correctness, we incorporate a verifier that checks candidate allocations and re-samples until all hardware constraints are satisfied.

We evaluate VERILOCC on two critical GPU kernels: general matrix multiplication (GEMM) and multi-head attention (MHA), which together account for over 90% of inference time in modern LLMs. A fine-tuned 7B LLM achieves 85–99% single-shot correctness and near-100% pass@100 with verification. A case study on the GEMM kernel running on AMD's MI250x GPU shows that VERILOCC discovers register assignments that exploit architectural features overlooked by existing compilers, outperforming rocBLAS (AMD Inc., 2024a), a production-grade, hand-optimized library, by 11.6% in runtime. These results demonstrate that VERILOCC combines the flexibility of data-driven learning with the reliability required for deployment in production compiler toolchains.

This paper makes the following contributions.

- We propose VERILOCC, a learning-based register allocator that combines LLMs, static analysis, and verifier-guided regeneration to achieve correctness and cross-architecture generalization.
- We evaluate VERILOCC on GEMM and MHA kernels, where a fine-tuned 7B LLM achieves 85–99% single-shot accuracy and near-100% pass@100, validating the feasibility of LLM-based register allocation.
- We show that VERILOCC can discover novel register assignments that outperform expert-tuned libraries, achieving 11.6% runtime improvement over rocBLAS on AMD MI250x.

Reproducibility. We will release our implementation and datasets upon acceptance.

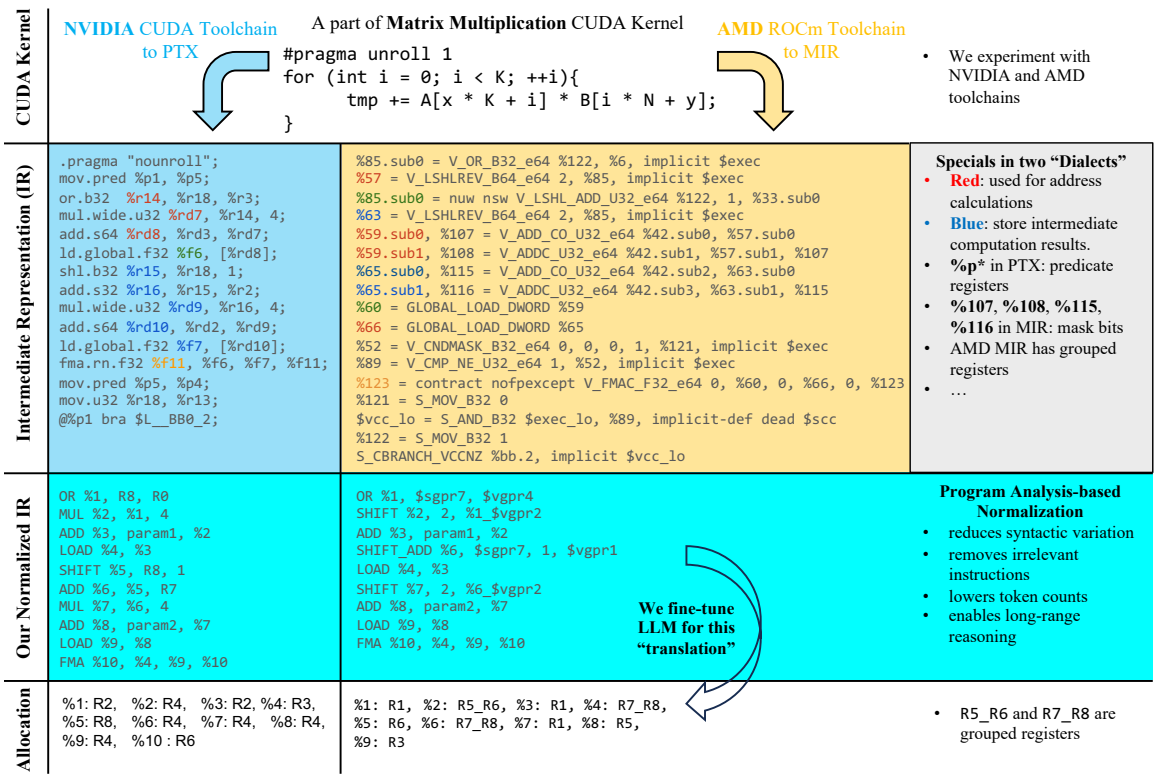

Figure 2: Illustrative example of MIR and ISA fragments across architectures. Register allocation occurs during the translation from MIR to ISA. For cross-architecture learning, we introduce a normalization process: the input to the LLM is a normalized MIR, and the output is a structured mapping between virtual and physical registers.

2 Preliminaries: Register Allocations in GPU Compilers

IR and MIR. Modern compilers use Intermediate Representations (IRs) extensively to perform optimizations. The compilers take the application written in high level languages (e.g., CUDA) as inputs and then transform it to multiple levels of IRs. The compilers tackle different optimization goals at different levels of IRs. For example, LLVM (Lattner and Adve, 2004) performs target independent optimizations such as constant propagations and dead code elimination at the level of LLVM IR, and performs target-specific optimizations such as coalescing memory accesses at the lower level machine IR (MIR). While different compiler toolchains have different namings (e.g., CUDA PTX vs SASS in the NVIDIA toolchains, and LLVM IR and LLVM MIR in the ROCm toolchains (AMD Inc., 2024b)), they share similar design principles. In the rest of the paper we use IR and MIR to refer to the target-independent and target-specific IRs.

Register Allocation. Programs in MIR are not directly executable. MIRs use *virtual registers*, or ϕ-nodes in SSA forms (Bilardi and Pingali, 2003) to represent values in the programs. To realize the MIR to executable ISAs, the register allocators assign these virtual values to *physical registers*. It must consistently assign the same value to the same register, and ensure that values with overlapping life cycles are assigned to disjoint registers. Additionally, the assignment must satisfy the hardware constraints (e.g., a 64-bit value must be assigned to two consecutive 32-bit registers). Due to the limited number of physical registers and hardware constraints, the register allocator might copy the values across registers, or temporarily spill the values to main memory.

Finding performant results of register allocations is particularly important for GPU programs since it directly affects the available parallelism and instruction latency. A performant allocation would minimize the copies and spills, and consider low-level hardware features to minimize pipeline stalls. For example, NVIDIA GPUs organize the registers into 4 banks. Reading values in the same banks requires stalling the pipeline for another cycle, thus in Figure 1 the compiler avoid the stalls by choosing R4, R6, and R9 in the FFMA instruction which are in separate register banks. Note that finding the optimal register allocation is NP-complete (Chaitin et al., 1981), thus compilers often rely on either

manual or learning-based heuristics when optimizing register allocations.

Complexity and Challenges. Low-level code generation introduces substantial complexity that poses unique challenges for our sequence-to-sequence formulation. When compiled for GPUs, kernels undergo aggressive transformations such as inlining, loop unrolling, and memory coalescing. These optimizations significantly inflate the size of both the MIR and the final assembly. As shown in Table 1, each multi-head attention (MHA) kernel averages 419 lines in PTX format, compared to 1,860 lines in LLVM IR. After tokenization, this corresponds to 9,294 tokens for PTX and 51,037 tokens for LLVM, far exceeding the context limits of current coding LLMs (Hsieh et al.). As a result, naively fine-tuning LLMs struggles to capture long-range dependencies among register uses and assignments.

Architectural variability presents a second major challenge. Different GPU vendors employ distinct IRs and toolchains: NVIDIA uses PTX, while AMD uses a custom MIR format. As shown in Figure 2, these representations differ in syntax, instruction structure, and register classes, making it difficult for cross-architecture modeling without careful normalization.

Finally, correctness is non-negotiable in compilers. A single invalid register assignment can lead to runtime crashes or silent corruption, rendering the compiled program unusable. Any learning-based allocator must therefore be paired with a mechanism that ensures compiler-level soundness.

These observations lead to three key challenges:

1. **Long sequences.** Optimized kernels produce IRs and ISAs that exceed the effective context window of typical LLMs, making long-range reasoning difficult.
2. **Cross-architecture generalization.** The model must reconcile shared allocation strategies with ISA-specific syntax and constraints.
3. **Compiler-level correctness.** The system must ensure sound register assignments to produce functional executables.

3 The VERILOCC Framework

Figure 1 illustrates the overall workflow of VERILOCC, which formulates register allocation as a sequence-to-sequence (seq2seq) transformation. Given tokenized MIR as input, the model generates structured register assignments in JSON format. However, raw MIR and ISA representations are often long, sparse, and inconsistent across architectures, making it difficult for LLMs to infer control flow and data dependencies implicitly. To address this, VERILOCC applies static analysis to normalize inputs and explicitly expose key semantic information at inference time. Finally, a verifier-guided regeneration loop ensures that the generated allocations satisfy semantic correctness.

3.1 Seq2Seq Formulation

Formally, given an input sequence $\mathcal{X}$ representing the tokenized MIR (potentially augmented with auxiliary information), the model generates an output sequence $\mathcal{Y}$ corresponding to register assignments in structured JSON format:

$$P(\mathcal{Y}|\mathcal{X}) = \prod_{i=1}^{m} P(y_i|\mathcal{X}, y_1, \ldots, y_{i-1}) \quad (1)$$

We train this mapping with a standard cross-entropy loss:

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m} \log P(y_i|\mathcal{X}, y_1, \ldots, y_{i-1}) \quad (2)$$

VERILOCC normalizes the MIR before using it as input. The auxiliary information consists of the control and data dependency, the constraints of the task (e.g., the number of available registers), as well as hardware constraints on reserved registers and values. VERILOCC follows the control flow graph to allocate registers per basic block. Figure 2 presents two concrete examples of normalized MIRs along with their corresponding register mappings.

3.2 Static Analysis-based Normalization

VERILOCC performs static analysis for two tasks: (1) reconstructing the results of register allocations from the training data, as well as making control and data dependency explicit during inferences, and (2) normalizing the MIR / ISA to reduce the number of tokens and to improve generalizations.

3.2.1 Register Assignment Reconstruction

The NVIDIA toolchain does not make the results of register allocations readily available. To recover the mappings for training, VERILOCC transforms both the MIR and ISA to the Static Single Assignment (SSA) forms, and performs standard context-insensitive, path-insensitive, flow-sensitive global

analysis (Xie and Aiken, 2007) to reconstruct the mappings. Particularly, VERILOCC follows the control flow graphs of both MIR and ISA, reconstructs the mappings by comparing the corresponding basic blocks, and finally consolidates the mappings for the full function. Using SSA forms allows VERILOCC easily to deal with the reordered instructions. VERILOCC uses heuristics to deal with cases where the toolchain chooses different instructions between MIRs and ISAs (e.g., lowering the `mul.wide.u32` instruction to a bit shift in Figure 2).

VERILOCC uses the same techniques above to analyze the control flows and data dependency of the inputs. The information is later injected as auxiliary information into the input sequences.

3.2.2 Normalizing MIRs

VERILOCC normalizes the MIRs for the training data and the inference inputs to reduce the number of tokens and to enhance generaliability across mutliple architectures and workloads. First, it strips out irrelevant metadata (e.g., comments and debug symbols), and replaces the ISA-specific prologues (e.g., the pointer to the function arguments) as symbolic values since they are irrelevant to the task of register allocations. Second, it normalizes the instructions and register classes of different architectures to common representations. For example, VERILOCC normalize `fma.rn.f32` and `V_FMAC_F32_e64` to `FMA` in the normalized MIR. VERILOCC also classifies a register to either a scalar or vector register, and it stores an either integer, float, or boolean value. Additionally, VERILOCC renumbers the registers for each basic block to reduce the ranges of the tokens which improves the reasoning performance of the models.

3.3 LLM Fine-tuning Pipeline

We implement VERILOCC using a 7B decoder-only language model fine-tuned on the normalized input-output pairs described above. As the backbone model, we use `Qwen2.5-Coder-7B-Instruct` for its strong coding performance and efficiency (Qwen et al., 2025). During training, the model is exposed to register allocation examples from NVIDIA and/or AMD toolchains. We will present more details in the experiment section.

3.4 Verifier-guided Inference

At inference time, the input is a normalized, tokenized MIR sequence along with auxiliary annotations, and the model autoregressively generates a register assignment dictionary. To eliminate hallucinations or incorrect results from the LLM, VERILOCC consists of a verifier to validate the results of register allocations before returning them to the compiler toolchain.

Table 1: Statistics of the data sets. The numbers of lines and tokens are averaged across different configurations.

Kernel	**# Configs**	**GPU**	**# Lines**	**# Tokens** (raw / normalized)	
GEMM	3,375	NVIDIA	79	1,233 / 113	90.8% ↓
		AMD	71	2,289 / 274	88.0% ↓
MHA	1,512	NVIDIA	419	9,294 / 1,944	79.1% ↓
		AMD	1,860	51,037 / 8,955	82.5% ↓

VERILOCC constructs a Satisfiability Modulo Theories (SMT) problem based on the input MIR to validate the followings:

- *Consistency.* It consistently assigns a virtual register to the same physical register.
- *Safety.* Virtual registers with overlapping life cycles are assigned to disjoint physical registers.
- *Realizability.* The assignment satisfies the hardware constraints.

Modeling the verification as SMT problems enables VERILOCC to reason about the solutions across various control flow paths, which is crucial for well-optimized performance-critical workloads like GEMM. Specifically, we use Z3 4.14.0 for validating the results of register allocations. It continuously re-samples the model until a valid allocation is found or a maximum number of attempts are exhausted.

4 Experiments

The evaluation aims to answer the following questions both qualitatively and quantitatively:

- How effective can VERILOCC generate correct register allocations?
- How effective can VERILOCC generalize over new programs and architectures?
- How effective can normalizations improve model performance?
- Is VERILOCC sufficiently fast to be used real-world settings?

4.1 Datasets

To understand the practical requirements of register allocation on modern GPUs, we curate data from two of the most computationally intensive kernels in large-scale deep learning: general matrix multiplication (**GEMM**) and multi-head attention (**MHA**) (Vaswani et al., 2017). These kernels dominate the inference workload of LLMs, often ac-

Table 2: Effectiveness of VERILOCC in three different settings. "w/o norm." is an ablation study of VERILOCC by disabling the MIR normalization. It reports the numbers for both the GEMM and MHA test cases.

		GEMM (768 test cases)					**MHA** (1865 test cases)				
		Greedy Decoding		**Sampling Decoding**			**Greedy Decoding**		**Sampling Decoding**		
Setup	**Model**	**Pass ↑ / Fail ↓**	**Pass Rate ↑**	**Pass@100 ↑**	**AvgTry ↓**	**MaxTry ↓**	**Pass ↑ / Fail ↓**	**Pass Rate ↑**	**Pass@100 ↑**	**AvgTry ↓**	**MaxTry ↓**
Same-NV	VERILOCC	764 / 4	**99.48%**	**99.47%**	**1.15**	**122**	1796 / 69	**96.30%**	**99.74%**	**1.44**	**149**
	w/o norm.	756 / 12	98.44%	99.08%	1.51	158	1779 / 86	95.39%	99.74%	1.52	181
Same-AMD	VERILOCC	764 / 4	**99.48%**	**99.86%**	**1.10**	**183**	1794 / 71	**96.19%**	**96.84%**	**2.61**	**105**
	w/o norm.	762 / 6	99.22%	99.35%	1.15	201	1775 / 90	95.17%	95.87%	3.09	176
Mixed	VERILOCC	753 / 15	98.05%	**98.56%**	1.14	**143**	1601 / 264	**85.84%**	**89.76%**	**6.37**	**227**
	w/o norm.	749 / 13	98.29%	98.05%	1.13	170	1491 / 374	79.95%	84.24%	9.11	241

counting for more than 90% of runtime (Dao et al., 2022).

To comprehensively evaluate register allocation strategies, we collect MIRs and ISAs of GEMM and MHA of various configurations. Each configuration has its own shapes of memory tiles (Kjolstad et al., 2017), different levels of unrolling, and different lengths of software pipelines. The dataset closely resembles real-world libraries including BLAS and FlashAttention (Dao et al., 2022), which compute the results for different dimensions of inputs with GPU-specific configurations for best runtime performances. The dataset consists of MIRs and ISAs for both NVIDIA RTX 4090 and AMD MI210 GPUs. Table 1 describes the statistics of the data set. The MHA kernels exhibit higher computational complexity with significantly more basic blocks per instance than GEMM, resulting in a total dataset of 9,325 instances. Please refer to Appendix A for more details.

4.2 Evaluation setups

We evaluate VERILOCC on two servers: one with equipped with 500GB SSD, 10GbE ethernet, and a RTX 4090 GPU. The other server with equipped with 17TB SSD, 10GbE ethernet, and a MI210 GPU. Both servers run Ubuntu 22.04, CUDA 12.4 and ROCm 6.3.1. We serve the models at the server with NVIDIA GPU and report the average of 100 runs of the runtime performances.

We randomly partition the dataset and use 80% / 20% of the data for training and testing, respectively. We consider two experiment settings: (1) **Same-NV** and **Same-AMD**: training and inferences are done with data on the same NVIDIA or AMD hardware architecture; and (2) **Mixed**, where training and inferences are done with data from both architectures.

Our primary metric is the **Pass Rate**, which measures the percentage of single-shot greedy decoding generations that pass the verification, i.e., produce a correct register allocation. We also report **Pass@100**, the proportion of test cases where at least one valid allocation is generated within 100 attempts. To assess runtime overhead if one integrates our LLM-based register allocator into the compiler toolchains, we additionally report the average attempts (**AvgTry**) and the maximum attempts (**MaxTry**) required to generate the first correct allocation.

4.3 Main Results

Table 2 reports the number of passing and failing instances, along with the percentages of valid generations under one-shot (Pass@1) and up to 100 attempts (Pass@100). It also includes the average and maximum number of attempts required to produce the first valid allocation.

VERILOCC achieves strong single-shot performance, with Pass@1 rates ranging from 85% to 99% across GEMM and MHA test cases. MHA kernels pose greater difficulty due to longer sequences and more complex dependencies, with a Pass Rate of 85.84% in the mixed training setting. Nevertheless, a simple resampling strategy proves highly effective, Pass@100 is nearly 100%, with the average number of attempts under 10, and maximum attempts typically falling within 100 to 200.

Overall, VERILOCC works very well in the two Same settings. While the Mixed setting is more challenging, VERILOCC still generalizes well. This indicates that LLMs can learn transferable patterns in register allocation across different ISAs.

4.4 Ablation on MIR Normalization

The MIR normalization is effective at reducing the input token numbers by about 80% to 90% as shown in Table 1. We further compare the end-to-end effect of disabling the normalization in Table 2. The normalized version consistently perform better in terms of almost all evaluation metrics. This

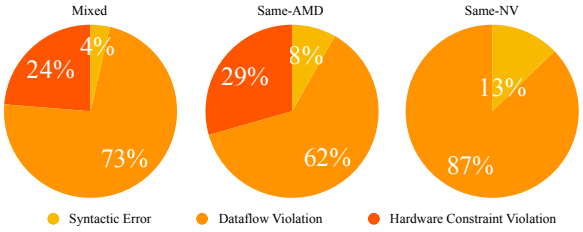

Figure 3: Error Distributions of VERILOCC on MHA under Different Settings. The total numbers of errors on GEMM are too limited, so we skip them.

confirms the importance of our proposed MIR normalization.

4.5 Error Analysis of Greedy Decoding

We analyze all error cases from VERILOCC's greedy decoding on MHA to better understand its limitations. GEMM is excluded because the total number of errors on GEMM is too small. Figure 3 shows the distribution of error types across different settings. Errors fall into three main categories

- **Syntactic Errors**: These account for 4–8% of failures and involve malformed JSON output, e.g., missing braces, broken key-value pairs, or extraneous text. Such errors are typically easy to fix via post-processing.
- **Dataflow Violations**: The most severe, comprising 62–73% of errors, where VERILOCC overwrites values that are still live later in the program. These semantic violations directly compromise program correctness.
- **Hardware Constraint Violations**: Making up 24–29% of errors, these occur when the model assigns values to register configurations that violate hardware rules, for example, using non-consecutive registers (`vgpr0_vgpr2`) for 64-bit operands, where contiguous pairs (e.g., `vgpr0_vgpr1`) are expected.

Interestingly, the proportion of dataflow violations is lower when the model is trained on a single architecture, suggesting that cross-architecture generalization remains a key challenge despite our normalization efforts. As a future direction, one could incorporate constrained decoding that explicitly masks out currently occupied registers during generation, helping to reduce both dataflow violations and hardware constraint errors.

4.6 Efficiency in End-to-End Compilation

The NVIDIA and AMD toolchains compile the test cases in 1107 ms on average. VERILOCC spends most of the time in two components: model servings and verification. Our evaluation server serves the Qwen2.5-Coder-7B-Instruct model at the speed of 8260 input tokens per second and 131 output tokens per second with no concurrent requests. Our end-to-end measurements closely match the numbers: the average time of serving a single inference in VERILOCC is 945 ms. VERILOCC can adopt techniques like continuous batchings (Kwon et al., 2023) and radix attentions (Zheng et al., 2024) to significantly speed up the inferences. This is left to future work.

The verifications take significant more time. On average it takes 31.77 ms to verify the correctness of the assignments with Z3. The longest verification takes 7.7 seconds. While validating worst case assignments does require the full capabilities of Z3, it is possible to accelerate the common cases where the verification can be done via following the control and data dependency in polynomial time. We leave this optimization to future work.

4.7 Case Study: Beyond Existing Compilers

One thing worth noting is that VERILOCC is able to find a more performant solution for GEMM compared to rocBLAS, the state-of-the-art GEMM library offered by AMD. This refined version of GEMM achieves 111.44 TFLOPS when multiplying 128×4096 and 4096×4096 `fp16` matrices with a strided batch of 3, which is 11.63% faster than rocBLAS. A detailed analysis shows that VERILOCC discovers a more performant assignment than the one used in the expert-optimized rocBLAS. Particularly, the MI250x GPU is based on the CDNA2 architecture, which introduces dedicated matrix core units to accelerate matrix multiplications. It offers two types of vector registers: standard architectural vector registers (VGPRs) or accumulation VGPRs (AccVGPRs), where the VGPRs can be used by all compute components but the AccVGPRs are exclusive to the matrix core uints. VERILOCC discovers an assignment that stores the values of the matrices in the AccVGPRs aggressively while rocBLAS stores them in the VGPRs. The publicly available ISA documentation (AMD Inc., 2022) indicates there should be no performance differences. We suspect that the matrix core units have quickly access to the AccVGPRs at the micro architecture level, which has a lower latency thus improves the performance. Though anecdote, it demonstrates the practical advantage and potential of VERILOCC over expert-optimized libraries.

Table 3: Effectiveness of VERILOCC while using 3B model on GEMM. MHA is too hard for 3B model.

		Greedy Decoding	Sampling Decoding		
Setup	**Model**	**Pass Rate ↑**	**Pass@100 ↑**	**AvgTry ↓**	**MaxTry ↓**
Same-NV	VERILOCC	**89.06%**	**98.44%**	**6.84**	**782**
	w/o norm.	88.02%	97.14%	7.03	977
Same-AMD	VERILOCC	**87.24%**	**98.67%**	**7.29**	**793**
	w/o norm.	86.46%	98.67%	7.85	993

4.8 Performance of 3B Model

We evaluate `Qwen2.5-Coder-3B-Instruct` to assess the feasibility of using smaller models for register allocation. While the 3B model achieves reasonable performance on GEMM (up to 98% pass@100 with sampling), it consistently fails on MHA tasks. This highlights a clear capability gap: smaller models struggle with the complex dependencies and register pressure present in real-world kernels like MHA. Notably, our normalization method continues to provide benefits at this scale, improving pass rates and reducing decoding attempts. These results suggest that more sophisticated compiler tasks may require larger models to maintain effectiveness.

5 Related Work

Learning-based Compiler Optimization. Recent advances in machine learning have opened new directions for compiler optimization by replacing handcrafted heuristics with learned models. MLGO (Chen et al., 2021) applies reinforcement learning in LLVM for tasks like inlining and register allocation, but depends on manual features and tight coupling with compiler internals. Meta's LLM Compiler (Liu et al., 2024) trains large language models on IR and assembly for end-to-end translation, yet lacks static analysis or verification, limiting correctness guarantees. General-purpose models like ChatGPT also struggle with register allocation, producing sequential, invalid assignments without liveness reasoning (Appendix B). In contrast, our work combines the expressiveness of LLMs with formal compiler techniques, using normalization and verifier-guided decoding to ensure correctness and cross-architecture generalization.

Optimizing tensor compilers. Modern AI models are essentially tensor programs. Optimizing tensor compilers (Chen et al., 2018; LI et al., 2023; Ma et al., 2020; Ragan-Kelley et al., 2013; Wu et al., 2025; Zheng et al., 2020) realize tensor programs to perfomant implementations on GPU or ASICs (Jouppi et al., 2017). They automatically apply techniques such as memory tiling (Ragan-Kelley et al., 2013), operator fusions (Chen et al., 2018), software pipelining (Ma et al., 2020) to search efficient schedules to utilize the hardware. They mostly operate at the tensor level on intermediate representations such as MLIR (Lattner et al., 2021), and delegates the effort of generating low-level ISAs to the GPU / ASIC toolchains. VERILOCC operates at a lower level of the software stack. It focuses on improving the register allocations inside the toolchains. They can be combined together to further improve performances.

Register allocation. Register allocation is one of the integral components of the compiler backends. Optimal register allocation is NP-Complete, therefore compilers uses manual heuristics, constraint solvings (Quintão Pereira and Palsberg, 2008), combinatorial optimizations (Lozano et al., 2019) and reinforcement learnings (VenkataKeerthy et al., 2023) to find satisfactory solutions. Compilers also opt for linear register allocations (Poletto and Sarkar, 1999) in just-in-time compilations. VERILOCC leverages LLMs to *learn* effective strategies of register allocations directly from the MIRs and ISAs, opening a new path path toward generalizable, semantics-aware compiler optimization.

Static analysis and validations. Static analysis (Lattner et al., 2007; Xie and Aiken, 2007) is effective to detect errors and vulnerabilities in the programs. VERILOCC uses sound static analysis (Mai et al., 2023) (i.e., no false negative) to effectively validate the results of register allocations are correct.

6 Conclusions and Future Work

We demonstrate that large language models (LLMs) can effectively learn register allocation as a sequence-to-sequence task, achieving high correctness rates and competitive runtime performance. VERILOCC generalizes across architectures, benefits from normalization, and even surpasses expert-optimized libraries in some scenarios, highlighting the potential of learning-based approaches in compiler backends.

A key direction for future work is integrating the verifier into training. One possibility is to provide verifier feedback as additional input to the model during fine-tuning. Another avenue is to explore reinforcement learning, where the verifier acts as a reward signal to directly guide the model toward sound and performant allocations.

Limitations

While VERILOCC shows strong results on register allocation for key GPU kernels, it has several limitations. First, our evaluation focuses on structured compute kernels (GEMM and MHA) that dominate LLM inference workloads; generalizing to less regular or control-heavy programs remains future work. Second, although we demonstrate cross-architecture generalization between NVIDIA and AMD backends, adaptation to entirely different compilation pipelines (e.g., CPU, tensor compilers) would require additional normalization and fine-tuning. Third, the model's performance degrades for highly complex kernels when using smaller architectures (e.g., 3B), suggesting that larger models may still be necessary for difficult compiler tasks. Finally, while our verifier ensures correctness, the re-sampling loop can add latency in low-confidence cases.

We believe these are meaningful directions for future work, including scaling to more optimization targets, improving efficiency, and expanding to broader compiler infrastructures.

References

V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools*. pearson Education.

AMD Inc. 2022. Amd instinct mi200 instruction set architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/instinct-mi200-cdna2-instruction-set-architecture.pdf.

AMD Inc. 2024a. rocBLAS: Rocm basic linear algebra subprograms (blas) library. https://github.com/ROCm/rocBLAS. Accessed: 2024-05-01.

AMD Inc. 2024b. Rocm compiler and programming model documentation. https://rocm.docs.amd.com.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*.

Gianfranco Bilardi and Keshav Pingali. 2003. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425.

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, and 1 others. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594.

Yundi Chen, Hanrui Wei, William Yu, Haonan Zhao, Yao Zhao, and Hanjun Yang. 2021. Mlgo: a machine learning guided compiler optimization framework. In *NeurIPS*.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: fast and memory-efficient exact attention with io-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? In *First Conference on Language Modeling*.

Norman P. Jouppi, Cliff Young, Nishant Patil, David 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, and 57 others. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA. Association for Computing Machinery.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA).

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 611–626, New York, NY, USA. Association for Computing Machinery.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA. IEEE Computer Society.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14.

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA. Association for Computing Machinery.

YIJIN LI, Jiacheng Zhao, Sun Qianqi, Haohui Mai, Lei Chen, Wanlu Cao, Yanfan Chen, Li zhicheng, YING LIU, Xinyuan Zhang, Xiyu Shi, Jie Zhao, Jingling Xue, HUIMIN CUI, and XiaoBing Feng. 2023. Sirius: Harvesting whole-program optimization opportunities for dnns. In *Proceedings of Machine Learning and Systems*, volume 5, pages 186–202. Curan.

Liang Liu, Marc Brockschmidt, Vivek Murali, and et al. 2024. The meta llm compiler: A suite of open-source models for code optimization. *arXiv preprint arXiv:2407.02524*.

Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial register allocation and instruction scheduling. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(3):1–53.

Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association.

HaoHui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. 2023. Honeycomb: Secure and efficient GPU executions via static validation. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 155–172, Boston, MA. USENIX Association.

NVIDIA Corporation. 2024. *CUDA Toolkit Documentation: Compiler*.

OpenAI. 2023. Gpt-4 technical report. `https://cdn.openai.com/papers/gpt-4.pdf`.

Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913.

Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register allocation by puzzle solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 216–226, New York, NY, USA. Association for Computing Machinery.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA. Association for Computing Machinery.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

S VenkataKeerthy, Siddharth Jain, Anilava Kundu, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. 2023. Rl4real: Reinforcement learning for register allocation. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 133–144.

Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, pages 1083–1095. IEEE Press.

Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: A multi-level superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, Boston, MA. USENIX Association.

Yichen Xie and Alex Aiken. 2007. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16–es.

Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Huazuo Gao, Jiashi Li, Liyue Zhang, Panpan Huang, Shangyan Zhou, Shirong Ma, Wenfeng Liang, Ying He, Yuqing Wang, Yuxuan Liu, and Y. X. Wei. 2025. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures. *Preprint*, arXiv:2505.09343.

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)*, pages 863–879. USENIX Association.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph Gonzalez, Clark Barrett, and Ying Sheng. 2024. Sglang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, volume 37. Curran Associates, Inc. Vancouver, Canada.

A Kernel Configurations

A **GEMM** kernel calculates $C = \alpha \cdot A \times B + \beta \cdot C$ with matrices $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$, and $C \in \mathbb{R}^{M \times N}$. To comprehensively evaluate register allocation strategies, we vary the matrix dimensions M, N, and K across a range of values from 2 to 16, resulting in $15 \times 15 \times 15 = 3,375$ distinct problem configurations. This breadth of configurations allows us to observe how register allocation patterns adapt to different memory access patterns and computational requirements.

MHA is the defining component of Transformer architectures, consisted of several computational phases: (1) projection of queries, keys, and values through linear transformations, (2) computation of attention scores through scaled dot-product attention, and (3) application of attention weights to values, followed by a final linear projection. This complex sequence of operations introduces intricate dependencies and memory access patterns, making optimal register allocation particularly challenging and impactful for performance.

We sample a large configuration space inspired by real LLM inference workloads:

- **Heads**: [16, 24, 32, 40, 64, 128]
- **Batch Size**: [1, 32, 64, 128, 256, 512, 1024]
- **Head Dim**: [32, 40, 59, 64, 80, 96, 111, 128, 160, 192, 224, 256]
- **Sequence Length**: [2048]
- **Attention Group**: [1, 4, 8]

This parameter space yields $6 \times 7 \times 12 \times 1 \times 3 = 1,512$ distinct configurations, representing a comprehensive sampling of the operational demands placed on modern LLM inference systems.

B ChatGPT Results

We analyze ChatGPT's behavior[1] on the register allocation task and observe substantial limitations in its ability to handle this core compiler optimization. As shown in Figure 4, ChatGPT outputs a naive sequential assignment strategy that fails to employ register reuse. This is reflected in the monotonically increasing register identifiers (`vgpr1` through `vgpr15`), with no attempt to recycle registers whose associated variables have expired. The result suggests a lack of liveness-aware reasoning and highlights the need for more specialized modeling or structural supervision to support compiler-level correctness.

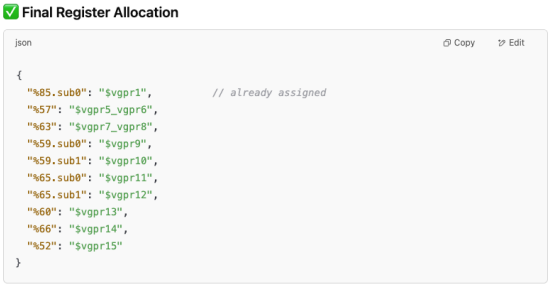

Figure 4: Allocation Result from ChatGPT

[1] https://chatgpt.com/share/682a07f9-bf50-8006-a7ac-f07b439439e4