

Latent Reasoning Guidance for Parallel Code Translation

Tomer Bitan¹ Erel Kaplan¹ Roei Bar-Yadin¹ Lian Ghrayeb¹
Le Chen² Samyak Jhaveri³ Niranjana Hasabnis⁴ Gal Oren^{1,5}

¹Technion – Israel Institute of Technology, Haifa, Israel

²Argonne National Laboratory, Lemont, USA ³University of California, Irvine, USA

⁴Code Metal, USA ⁵Stanford University, Stanford, USA

{tomerbital, erel.kaplan, roee.bar, lian.ghrayeb}@campus.technion.ac.il
lechen@anl.gov, samyaknj@uci.edu, niranjan@codemetal.ai
galoren@stanford.edu

Abstract

Tackling complex coding tasks often requires autonomous agents and iterative repair pipelines. These increasingly rely on large amounts of test-time computation, often spending many decoding and repair steps before discovering whether a program compiles, runs, or validates. Executable parallel-code translation is an effective setting for earlier guidance because success is behavioral rather than textual. However, most guidance methods act only after complete programs or textual traces are decoded. This motivates the question: can latent reasoning provide an earlier intervention point, before the model commits to code?

We study a test-time latent guidance method for this setting that trains a smaller Process Reward Model (PRM) over continuous latent prefixes and uses it to select among alternate hidden-state trajectories before final code decoding, separately from but compatible with post-decoding optimization. On a 76-task ParaTrans benchmark evaluation, latent PRM guidance improves mean validation rate from 32.89% with unguided latent reasoning to 42.1%, outperforming fine-tuned and vanilla baselines in the same setting. These gains persist under the same three-attempt repair loop. These results provide bounded evidence that useful alternative latent continuations exist and that PRM-scored latent branch selection can improve executable outcomes in this setting without retraining the main generative model.

1 Introduction

Scientific and high-performance applications often need to move between programming models such as CUDA, OpenMP, OpenCL, and serial CPU implementations as hardware, compiler support, and deployment constraints change (Kadosh et al.,

2023a). Parallel API translation is therefore an important but difficult code-generation problem because translations must preserve the source computation while satisfying target-specific requirements for parallel structure, memory movement, synchronization, and API usage (Jin and Vetter, 2023; Bitan et al., 2025).

Large language models and autonomous code agents can propose flexible transformations beyond rigid source-to-source rules (Kadosh et al., 2023b; Chen et al., 2024a; Kaplan et al., 2026), but in executable parallel-code translation, even plausible looking outputs may fail to compile, crash, or violate reference behavior, pushing many systems toward generate→test→repair loops.

Small differences in intermediate reasoning can lead to different compile→run→validate outcomes, and poor high-level parallelization choices can be hard to repair after decoding. This motivates an earlier intervention point: can nearby reasoning trajectories lead to better executable outcomes before any token-level repair is attempted?

Existing guidance methods typically score decoded programs, textual reasoning traces, or completed candidates (Cobbe et al., 2021; Lightman et al., 2024; Li et al., 2025; Dai et al., 2024).

Latent reasoning offers an earlier intervention point by shifting some test-time computation into continuous hidden-state trajectories before standard decoding resumes (Hao et al., 2025; Xu et al., 2025; Yue et al., 2025; Zou et al., 2026). An intervention during the continuous phase can have a large effect redirecting subsequent latent steps and decoded tokens before the model commits to an implementation strategy. This is especially relevant to parallel-code translation, where early choices about work decomposition, memory movement, and synchronization constrain much of the final program. However, recent work has begun to investigate this intervention point through latent-state optimization (You et al., 2026; Li et al., 2026)

¹GitHub:<https://github.com/Scientific-Computing-Lab/Parallax.git>

We examine this question in executable parallel-code translation, a domain that requires specialized parallel-programming knowledge and in which pre-decoding guidance must remain compatible with post-decoding repair.

We study a test-time latent guidance method for parallel API translation that keeps the primary model frozen, samples perturbed hidden-state branches during latent reasoning, scores partial latent prefixes with a smaller PRM, and resumes generation from the highest-scoring branch, placing task-specific adaptation in the compact guidance model rather than retraining the generator. Related work and positioning are discussed in Appendix A.

Empirically, latent PRM guidance improves ParaTrans validation by 9.21 percentage points over unguided latent reasoning, and the gain persists when inserted into the same three-attempt repair loop. A held-out branch-selection test further suggests that the PRM can learn a weak but useful local preference signal to avoid undesirable latent continuations.

We make three contributions. First, we study pre-decoding latent branch selection for executable parallel-code translation and test its compatibility with post-decoding repair. Second, we instantiate this intervention with a smaller PRM and evaluate it on ParaTrans, where correctness requires compilation, execution, and validation rather than textual similarity. Third, we provide bounded mechanism-level evidence from a held-out branch-selection test that a smaller PRM can provide a conservative local preference signal over continuous latent branches before final code decoding.

2 Method

We instantiate a test-time optimization method for guiding latent reasoning without fine-tuning the main generator in parallel code translation. Specifically, we keep a large latent-reasoning primary model frozen and train a smaller PRM to score partial latent trajectories before text decoding, as shown in Figure 1.

2.1 Latent-reasoning Primary Model

Let M be a frozen autoregressive model equipped with continuous latent reasoning. In our main experiments, M is LLaMA-3.3-70B. Following the COCONUT-style latent-reasoning paradigm (Hao et al., 2025), the model performs a fixed number of latent steps before returning to standard token de-

coding. Let $h_t \in \mathbb{R}^{d_M}$ denote the primary model’s hidden state at latent step t , before the language-modeling head. During the latent phase, h_t is not decoded into a discrete token; instead, it is mapped back into the M input space and used as the next continuous input.

Instead of training the main model in the manner proposed in Hao et al. (2025), we use the training-free alignment transform of Zou et al. (2026). Given the input embedding matrix W_{in} and output unembedding matrix W_{out} , we compute

$$W_a = (W_{\text{out}}^\top W_{\text{out}})^{-1} W_{\text{out}}^\top W_{\text{in}},$$

the closed-form solution to

$$\min_{W_a} \|W_{\text{out}} W_a - W_{\text{in}}\|_F^2.$$

At latent step t , the aligned continuous input for the next step is $z_{t+1} = h_t W_a$. We use $K_{\text{train}} = 6$ latent steps when collecting PRM training trajectories and $K_{\text{test}} = 12$ latent steps during inference.

2.2 Latent process reward model

We train a smaller reward model to estimate the potential of a partial latent trajectory of the reasoning process via the mean terminal quality score. The PRM uses a Qwen-Coder-7B backbone with a scalar value head. Because the PRM receives hidden states from the primary LLaMA-3.3-70B model rather than its own token embeddings, we prepend a learned adapter that maps this hidden dimension to the PRM embedding dimension. In our implementation, this maps 8192-dimensional generator states to the 3584-dimensional Qwen-Coder input space using a linear layer, GELU activation, and layer normalization.

For every latent reasoning step t , the PRM V_θ receives the prefix trajectory

$$\tau_{\leq t} = (h_1, \dots, h_t)$$

and predicts a scalar value

$$V_\theta(\tau_{\leq t}).$$

The target is the empirical terminal reward obtained from rollout sampling.

For rollout supervision at each latent step t , we sample $B_{\text{train}} = 3$ candidate branch states: two perturbed resamples and the unperturbed hidden state.

$$\tilde{h}_t^{(b)} = h_t + \epsilon^{(b)}, \quad \epsilon^{(b)} \sim \mathcal{N}(0, 1.4^2 I). \quad (1)$$

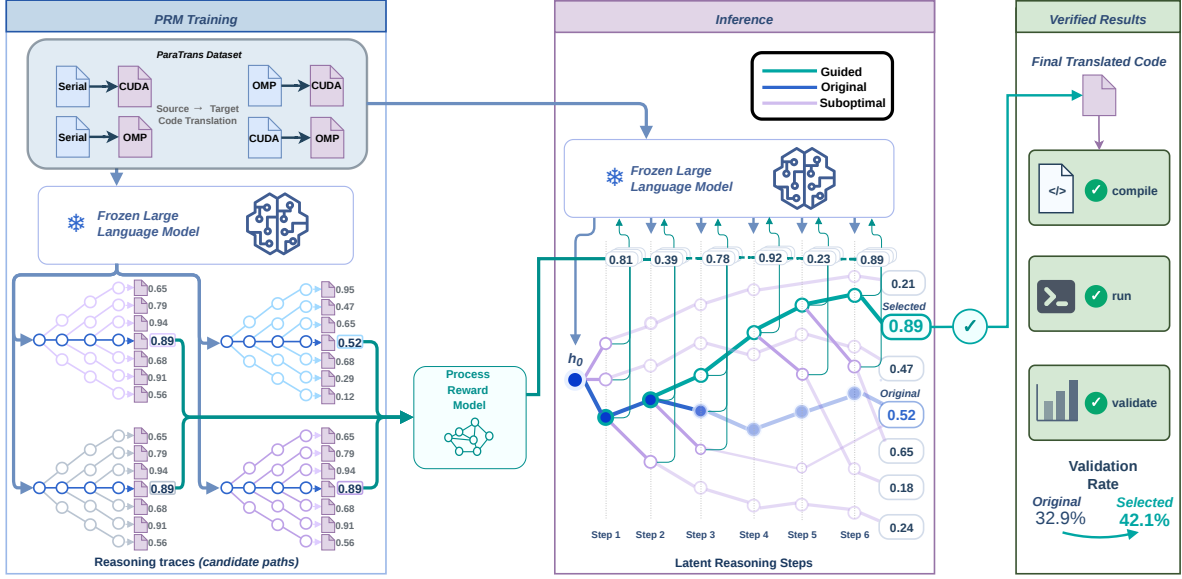


Figure 1: **Latent PRM-guided translation pipeline.** Executable feedback is converted into a pre-decoding guidance signal for parallel-code translation. (1) PRM training uses ParaTrans paired tasks across Serial, CUDA, and OpenMP: a frozen latent-reasoning generator samples candidate hidden-state paths, decodes them, and assigns supervision from executable outcomes. (2) The process reward model learns to score partial latent branches before code is emitted. (3) During inference, a new source task is processed by the frozen generator; candidate latent branches are proposed at each step, scored by the PRM, and reduced to the highest-scoring trajectory before final decoding. (4) Verified results are obtained by compiling, running, and validating the translated code against the reference behavior. This design tests whether latent trajectories can be steered before decoding while leaving the main generator frozen.

For each branch b , we roll out to terminal code $y_t^{(b)}$ and score the completed program with an automatic reward function $S(\cdot)$.

The prefix target is the average terminal score of its continuations:

$$R_t = \frac{1}{B_{\text{train}}} \sum_{b=1}^{B_{\text{train}}} S(y_t^{(b)}),$$

The PRM is trained with mean squared error:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N \left(V_{\theta}(\tau_{\leq t}^{(i)}) - R_t^{(i)} \right)^2.$$

2.3 Training data and reward construction

We construct the PRM training set from primary-model reasoning trajectories on ParaTrans samples used by Bitan et al. (2025) and supplement with additional samples created with the same methodology, balanced across CUDA→OpenMP, OpenMP→CUDA, Serial→OpenMP, and Serial→CUDA. We use 60 development samples for hyperparameter selection and train the final PRM on the combined training and development

set before evaluating on the held-out test tasks, as shown on the left side of Figure 1.

The terminal reward $S(y)$ combines executable and semantic signals, providing denser supervision than executable feedback alone. It is distinct from the reported benchmark metric. Compilation and execution receive fixed weights of 0.30 and 0.25, respectively. The remaining 0.45 weight is distributed across the validators available for the sample, proportional to their base weights. These validators include pass/fail validation, checksum or output matching when available, benchmark metrics, code-embedding similarity (Günther et al., 2024), an LLM-as-a-judge score, and a variable-comparison heuristic inspired by Kaplan et al. (2026). In contrast, final benchmark results use executable validation only: a translation must compile, run, and pass integrated validation against the reference behavior. Appendix C gives the exact reward construction and validator details, and Section 4.4 reports a grouped reward ablation.

2.4 PRM-guided latent inference

At inference time, we perform greedy branch selection during the latent reasoning phase, as shown

in the inference panel of Figure 1. For every latent step t , we score $B_{\text{test}} = 8$ candidate branch states: the unperturbed state h_t and seven perturbed variants from Eq. 1.

Each candidate forms a possible branch

$$\tau_{\leq t}^{(b)} = (h_1, \dots, h_{t-1}, \tilde{h}_t^{(b)}).$$

The PRM scores each candidate branch and selects

$$b_t^* = \arg \max_b V_\theta(\tau_{\leq t}^{(b)}).$$

Only the selected branch is retained. The selected hidden state $\tilde{h}_t^{(b_t^*)}$ is aligned with W_a and fed back into the primary model for the next latent step. After $K = 12$ latent steps, the primary model returns to standard autoregressive decoding and produces the final target code. While initial training is done with $K = 6$ latent steps for efficiency, we find that using more latent steps at inference can further improve performance, as shown in Appendix C.

3 Experimental Setup

The main evaluation uses the 76-task ParaTrans test set. As a metric, we report validation rate, which is a three-run mean fraction of translations that compile, execute without timing out, and pass all task-specific behavioral checks against the benchmark reference implementation, including output, checksum, or benchmark-specific checks where applicable. This is stronger than compilation or textual similarity, although it remains benchmark-level behavioral testing rather than a formal proof of semantic equivalence. We compare vanilla LLaMA-3.3-70B, the LLaMA-3.3-70B fine-tuned on ParaTrans from Bitan et al. (2025), unguided latent reasoning with the same frozen base model, random latent branch selection with the same perturbation budget, and latent PRM guidance. We do not evaluate latent reasoning on the fine-tuned generator or a combined fine-tuned-generator-plus-latent-PRM setting. Appendix C gives the full evaluation protocol, including direction counts, overlap checks, the repair setting, and the held-out branch-selection benchmark; direction-wise results appear in Appendix B.

4 Results

4.1 Main executable translation results

Table 1 reports executable validation rates on ParaTrans. The key comparison is between unguided

Table 1: Three-run mean executable validation rate (%) on the ParaTrans test set, with and without the same three-attempt repair loop.

Method	No repair	+3-attempt repair
Vanilla	14.55±2.55	30.67±4.34
Fine-tuned	33.48±1.11	36.00±3.11
Latent reasoning	32.89±3.44	36.40±5.47
Random branch selection	27.28±6.34	—
Latent PRM guidance	42.10±2.28	45.18±3.31

latent reasoning and latent PRM guidance, since both use the same frozen base model and differ only in whether latent branches are scored and selected. Latent PRM guidance improves validation by 9.21 percentage points without repair (paired task-bootstrap 95% CI [+3.07, +15.79], two-sided paired sign-flip $p = 0.0081$) and by 8.78 percentage points under the same three-attempt repair loop (95% CI [+2.63, +15.35], $p = 0.0064$). Latent PRM guidance without repair surpasses both configurations of the fine-tuned baseline, suggesting this can be an effective method for domain specialization on this dataset. A random branch-selection baseline with the same perturbation budget reaches only 27.28% validation rate, suggesting that many latent perturbations are harmful rather than useful. Without repair, latent PRM guidance already exceeds all non-PRM baselines, including repaired latent reasoning. With repair, it improves further to 45.18%, reinforcing the notion that pre-decoding guidance remains useful alongside later token-level repair. Section 4.4 reports the reward and latent-search-budget ablations. Appendix B reports the test-time overhead comparison, task-level win/loss/tie counts, direction-wise results, and a post-decoding best-of-8 control.

4.2 Latent-trajectory branch analysis

We test whether perturbed latent continuations can improve on the unperturbed trajectory. On the 952-sample PRM training set, 95% of trajectory samples contain at least one candidate branch with higher terminal reward than the original. 59% of candidate continuations improve over their corresponding unperturbed branch. Among samples with at least one trajectory that passes both validation and output comparison, 25% succeed only through branching: the original trajectory fails, while at least one perturbed continuation succeeds. These results show that better latent continuations often exist locally, though distinguishing them from

Table 2: Held-out branch-selection results on 500 instances. “Best” is strict best-branch accuracy; $> \text{Orig.}$ is strict improvement over the original continuation, and $\geq \text{Orig.}$ is at least as good as the original.

Selector	Selection rate (%)		
	Best	$> \text{Orig.}$	$\geq \text{Orig.}$
Original	33.66	-	-
Random	32.14 ± 0.46	43.0 ± 0.4	66.9 ± 0.3
Latent PRM	36.25 ± 0.41	45.0 ± 0.2	79.2 ± 0.3

harmful ones remains difficult.

4.3 Branch-selection analysis

The branch-selection test isolates whether the PRM can choose a useful latent branch from a fixed local candidate set. To avoid leakage, this PRM excludes the 120 training trajectories used to construct the 500 branch-selection instances.

Table 2 provides bounded support for the mechanism behind latent guidance. Relative to random selection, the PRM more often selects a branch that improves over the original continuation or at least preserves it. While only modestly improving strict best-branch accuracy, this local advantage is applied repeatedly across consecutive latent steps and multiple candidate branches, where choices can compound before decoding begins. Together with the poor random-branch result in Table 1, this suggests that the PRM mainly helps by filtering harmful latent perturbations rather than by reliably finding the single best branch.

4.4 Ablations

Reward Construction. Table 3 compares the full mixed reward with grouped executable-only and auxiliary-only PRM supervision. All variants use the same latent-guidance inference configuration, $B_{\text{test}} = 8$ and $K_{\text{test}} = 12$, and each entry is the mean executable validation rate over three runs.

The executable-only target retains compilation, execution, and integrated pass/fail or output-based behavioral checks. The auxiliary-only target retains code-embedding similarity, the LLM judge, and the variable-comparison signal. For each restricted variant, the active weights are renormalized over the retained signals. The full mixed target performs best, reaching 42.10% validation compared with 26.75% for executable-only supervision and 29.38% for auxiliary-only supervision.

This grouped ablation does not isolate every validator and therefore does not establish that each

PRM supervision	Validation (%)
Full mixed reward	42.10
Executable signals only	26.75
Auxiliary semantic signals only	29.38

Table 3: Three-run mean executable validation rate for the grouped PRM-supervision ablation on the ParaTrans test set.

B_{test}	K	Validation (%)
1	6	26.32
3	6	28.94
1	12	32.89
3	12	35.52
8	12	42.10

Table 4: Three-run mean validation rate (%) for the test-time latent-search-budget ablation on the ParaTrans test set.

component is necessary. It directly supports only the broader choice of combining executable and auxiliary supervision. The result is consistent with binary executable feedback being too sparse by itself and semantic proxies being insufficient on their own for executable correctness. The reported task metric remains compilation, execution, and integrated validation against the reference behavior.

Latent Search Budget. Table 4 ablates the test-time latent search budget by varying the number of candidate branches per step, B_{test} , and the number of latent reasoning steps, K . The $B_{\text{test}} = 1$ rows are unguided latent-reasoning controls; rows with $B_{\text{test}} > 1$ use learned PRM selection. Each entry is the mean executable validation rate over three runs.

Due to time and compute constraints, we construct PRM training trajectories under the smaller latent-step budget, but the preliminary pattern suggests that the PRM benefits from a larger decision surface at inference. Although task-specific PRM trajectories end at $K_{\text{train}} = 6$, the adapter, Qwen backbone, and value head share parameters across positions, and the backbone was pretrained on substantially longer sequences. More reasoning steps and more candidate options give the scorer additional opportunities to avoid weak continuations and occasionally choose stronger latent branches before decoding. We treat $K_{\text{test}} = 12$ as an empirical inference-time budget extension rather than guaranteed length generalization.

Limitations

This study is limited to one base-model family, one PRM backbone, and one executable parallel-code translation benchmark. The ParaTrans test set contains 76 held-out tasks, so the results should be read as evidence for this setting rather than as a general claim about latent guidance across models or domains. We also do not evaluate transfer of the learned value model across base models. The adapter architecture and perturbation scale remain fixed, while the three-attempt repair budget is controlled but not ablated. We also do not compare greedy selection with beam-style latent search, which would alter the test-time compute budget and introduce additional search hyperparameters. We leave these broader evaluations to future work.

Latent PRM guidance assumes access to intermediate hidden states, which limits its application to self-hosted models or otherwise white-box models unless black-box APIs expose these states.

As with other domain-specific adaptation methods, the approach also requires task-specific supervision to train the guidance model. In our setting, this supervision is costly because parallel-code translations are long and each labeled example requires multiple rollouts followed by executable and semantic validation. This limits the scale of the present study and makes broader robustness and transfer evaluations more difficult.

Finally, although the branch-selection analysis is consistent with the filtering interpretation, it does not by itself explain all success and failure cases. Finer-grained analysis of these cases could inform more reliable latent intervention strategies beyond local branch filtering.

Risk Statement

We do not identify material new risks from this work. The study is limited to offline benchmark evaluation of parallel-code translation.

Acknowledgments

Computational support was provided by Code Metal.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. [Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages](#). In *Proceedings of the 17th*

Conference of the European Chapter of the Association for Computational Linguistics, pages 1528–1542.

Tomer Bitan, Tal Kadosh, Erel Kaplan, Shira Meiri, Le Chen, Peter Morales, Niranjan Hasabnis, and Gal Oren. 2025. [UniPar: A unified LLM-based framework for parallel and accelerated code translation in HPC](#). In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE.

Gregory Bolet, Giorgis Georgakoudis, Harshitha Menon, Konstantinos Parasyris, Niranjan Hasabnis, Hayden Estes, Kirk Cameron, and Gal Oren. 2025a. [Can large language models predict parallel code performance?](#) In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, pages 1–6.

Gregory Bolet, Giorgis Georgakoudis, Konstantinos Parasyris, Harshitha Menon, Niranjan Hasabnis, Kirk W Cameron, and Gal Oren. 2025b. [Counting Without Running: Evaluating LLMs’ Reasoning About Code Complexity](#). *arXiv preprint arXiv:2512.04355*.

Aman Chaturvedi, Daniel Nichols, Siddharth Singh, and Abhinav Bhatele. 2025. [Hpc-coder-v2: Studying code llms across low-resource parallel languages](#). In *ISC High Performance 2025 Research Paper Proceedings (40th International Conference)*, pages 1–14. Prometheus GmbH.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. [CodeT: Code Generation with Generated Tests](#). *arXiv preprint arXiv:2207.10397*.

Le Chen, Nesreen K. Ahmed, Akash Dutta, Arijit Bhattacharjee, Sixing Yu, Quazi Ishtiaque Mahmud, Waqwoya Abebe, Hung Phan, Aishwarya Sarkar, Brandon Butler, Niranjan Hasabnis, Gal Oren, Vy A. Vo, Juan Pablo Munoz, Theodore L. Willke, Tim Mattson, and Ali Jannesari. 2024a. [The Landscape and Challenges of HPC Research and LLMs](#). *Preprint, arXiv:2402.02018*.

Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. 2024b. [OMPGPT: A generative pre-trained transformer model for openmp](#). In *European Conference on Parallel Processing*, pages 121–134. Springer Nature Switzerland Cham.

Le Chen, Nuo Xu, Winson Chen, Bin Lei, Pei-Hung Lin, Dunzhi Zhou, Rajeev Thakur, Caiwen Ding, Ali Jannesari, and Chunhua Liao. 2025. [Beyond Code Pairs: Dialogue-Based Data Generation for LLM Code Translation](#). *arXiv preprint arXiv:2512.03086*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Mihir Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, and 37 others.

2021. [Evaluating Large Language Models Trained on Code](#). *arXiv preprint arXiv:2107.03374*.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. [Training Verifiers to Solve Math Word Problems](#). *Preprint*, arXiv:2110.14168.
- Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. 2024. [Process supervision-guided policy optimization for code generation](#). *arXiv preprint arXiv:2410.17621*.
- Matthew T Dearing, Yiheng Tao, Xingfu Wu, Zhiling Lan, and Valerie Taylor. 2024. [Lassi: An LLM-based automated self-correcting pipeline for translating parallel scientific codes](#). In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, pages 136–143. IEEE.
- Jingcheng Deng, Liang Pang, Zihao Wei, Shicheng Xu, Zenghao Duan, Kun Xu, Yang Song, Huawei Shen, and Xueqi Cheng. 2026. [LLM Latent Reasoning as Chain of Superposition](#). *Preprint*, arXiv:2510.15522.
- Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. 2024. [Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents](#). *Preprint*, arXiv:2310.19923.
- Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. 2025. [Training Large Language Models to Reason in a Continuous Latent Space](#). *Preprint*, arXiv:2412.06769.
- Re'em Harel, Yuval Pinter, and Gal Oren. 2023. [Learning to parallelize in a shared-memory environment with transformers](#). In *Proceedings of the 28th ACM SIGPLAN annual symposium on principles and practice of parallel programming*, pages 450–452.
- Re'em Harel, Tal Kadosh, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2025. [Pragformer: data-driven parallel source code classification with transformers](#). *International Journal of Parallel Programming*, 53(1):2.
- Re'em Harel, Idan Mosseri, Harel Levin, Lee-or Alon, Matan Rusanovsky, and Gal Oren. 2020. [Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential](#). *International Journal of Parallel Programming*, 48(1):1–31.
- Yufan Huang, Mengnan Qi, Yongqiang Yao, Maoquan Wang, Bin Gu, Colin Clement, and Neel Sundaresan. 2023. [Program Translation via Code Distillation](#). *arXiv preprint arXiv:2310.11476*.
- Zheming Jin and Jeffrey S. Vetter. 2023. [A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model](#). In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 325–327.
- Tal Kadosh, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023a. [Quantifying OpenMP: Statistical insights into usage and adoption](#). In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE.
- Tal Kadosh, Niranjana Hasabnis, Prema Soundararajan, Vy A Vo, Mihai Capota, Nesreen Ahmed, Yuval Pinter, and Gal Oren. 2024a. [OMPPar: Automatic Parallelization with AI-Driven Source-to-Source Compilation](#). In *Machine Learning for Systems Workshop, NeurIPS*.
- Tal Kadosh, Niranjana Hasabnis, Vy A. Vo, Nadav Schneider, Neva Krien, Mihai Capota, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, Yuval Pinter, Timothy Mattson, and Gal Oren. 2024b. [MonoCoder: Domain-Specific Code Language Model for HPC Codes and Tasks](#). *Preprint*, arXiv:2312.13322.
- Tal Kadosh, Niranjana Hasabnis, Vy A. Vo, Nadav Schneider, Neva Krien, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, Yuval Pinter, Timothy Mattson, and Gal Oren. 2023b. [Scope is all you need: Transforming LLMs for HPC Code](#). *Preprint*, arXiv:2308.09440.
- Tal Kadosh, Nadav Schneider, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023c. [Advising OpenMP parallelization via a graph-based approach with transformers](#). In *International Workshop on OpenMP*, pages 3–17. Springer Nature Switzerland Cham.
- Erel Kaplan, Tomer Bitan, Lian Ghayeb, Le Chen, Tom Yotam, Niranjana Hasabnis, and Gal Oren. 2026. [ParaCodex: A Profiling-Guided Autonomous Coding Agent for Reliable Parallel Code Generation and Translation](#). *Preprint*, arXiv:2601.04327.
- Jiakang Li, Guanyu Zhu, Can Jin, Chenxi Huang, Dexu Yu, Ronghao Chen, Yang Zhou, Hongwu Peng, Xuanqi Lan, Dimitris N. Metaxas, and Youhua Li. 2026. [Latent Reward Steering: An Adaptive Inference-Time Framework that Implicitly Promotes Cognitive Behaviors in Reasoning LLMs](#). *Preprint*, arXiv:2606.00726.
- Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025. [CodePRM: Execution feedback-enhanced process reward model for code generation](#). In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8169–8182.
- Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. [Let's verify step by step](#). In *International*

- Conference on Learning Representations*, volume 2024, pages 39578–39601.
- Ilya Loshchilov and Frank Hutter. 2019. [Decoupled Weight Decay Regularization](#). In *International Conference on Learning Representations*.
- Francesca Lucchetti and Arjun Guha. 2025. [Understanding How CodeLLMs \(Mis\) Predict Types with Activation Steering](#). In *Proceedings of the 8th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pages 358–397.
- Matthew Macfarlane and Clem Bonnet. 2026. [Searching latent program spaces](#). *Advances in Neural Information Processing Systems*, 38:103463–103520.
- Gabriel Martinez, Mark K. Gardner, and Wu chun Feng. 2011. [CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures](#). pages 300–307.
- Robert Mueller-Albrecht. 2024. [Syclomatic: SYCL adoption for everyone-moving from CUDA to SYCL gets progressively easier: Advanced migration considerations](#). In *Proceedings of the 12th International Workshop on OpenCL and SYCL*, pages 1–2.
- Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024. [Hpc-coder: Modeling parallel programs using large language models](#). In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–12. Prometheus GmbH.
- Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. [Show Your Work: Scratchpads for Intermediate Computation with Language Models](#). *Preprint*, arXiv:2112.00114.
- Md Mahbubur Rahman, Arjun Guha, and Harshitha Menon. 2026. [Steering Code LLMs with Activation Directions for Language and Library Control](#). *arXiv preprint arXiv:2603.23629*.
- Nina Rimskey, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Turner. 2024. [Steering llama 2 via contrastive activation addition](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15504–15522.
- ROCm Developers. 2023. [HIPIFY: CUDA to HIP source translation tool](#). Accessed: 2025-06-09.
- Nadav Schneider, Niranjan Hasabnis, Vy A. Vo, Tal Kadosh, Neva Krien, Mihai Capotă, Guy Tamir, Ted Willke, Nesreen Ahmed, Yuval Pinter, Timothy Mattson, and Gal Oren. 2024. [MPIrigen: MPI Code Generation through Domain-Specific Language Models](#). *Preprint*, arXiv:2402.09126.
- Nadav Schneider, Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. [Mpi-ricol: Data-driven mpi distributed parallelism assistance with transformers](#). In *Proceedings of the SC’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 2–10.
- Changqing Shi, Yufei Sun, Rui Chen, Jiahao Wang, Qiang Guo, Chunye Gong, Yicheng Sui, Yutong Jin, and Yuzhi Zhang. 2025. [TransCL: An automatic CUDA-to-OpenCL programs transformation framework](#). *ACM Transactions on Architecture and Code Optimization*, 22(2):1–24.
- Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J Vazquez, Ulisse Mini, and Monte MacDiarmid. 2023. [Steering language models with activation engineering](#). *arXiv preprint arXiv:2308.10248*.
- Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. [Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. [Chain-of-thought prompting elicits reasoning in large language models](#). *Advances in neural information processing systems*, 35:24824–24837.
- Yige Xu, Xu Guo, Zhiwei Zeng, and Chunyan Miao. 2025. [SoftCoT: Soft chain-of-thought for efficient reasoning with LLMs](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23336–23351.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. [Tree of thoughts: Deliberate problem solving with large language models](#). *Advances in neural information processing systems*, 36:11809–11822.
- Runyang You, Yongqi Li, Meng Liu, Wenjie Wang, Liqiang Nie, and Wenjie Li. 2026. [Parallel Test-Time Scaling for Latent Reasoning Models](#). In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 44703–44717, San Diego, California, United States. Association for Computational Linguistics.
- Zhenrui Yue, Bowen Jin, Huimin Zeng, Honglei Zhuang, Zhen Qin, Jinsung Yoon, Lanyu Shang, Jiawei Han, and Dong Wang. 2025. [Hybrid latent reasoning via reinforcement learning](#). *arXiv preprint arXiv:2505.18454*.
- Zhen Zhang, Xuehai He, Weixiang Yan, Ao Shen, Chenyang Zhao, Shuohang Wang, Yelong Shen, and Xin Eric Wang. 2025a. [Soft Thinking: Unlocking the](#)

Reasoning Potential of LLMs in Continuous Concept Space. *arXiv preprint arXiv:2505.15778*.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025b. *The lessons of developing process reward models in mathematical reasoning*. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10495–10516.

Jiaru Zou, Ruizhong Qiu, Gaotang Li, Xiyuan Yang, Katherine Tieu, Pan Lu, Ke Shen, Hanghang Tong, Yejin Choi, Jingrui He, James Zou, Mengdi Wang, and Ling Yang. 2026. *Latent Collaboration in Multi-Agent Systems*. In *Forty-third International Conference on Machine Learning*.

A Expanded Related Work and Positioning

Execution-based and compiler-guided code-generation work has established that behavioral tests are a stronger signal for program quality than string overlap (Chen et al., 2021, 2022). Program-translation work then extended that view to translation settings through supervised adaptation, back-translation, distillation, and dialogue-based data generation (Ahmad et al., 2023; Huang et al., 2023; Chen et al., 2025). These papers are relevant because they define the broader code-generation landscape in which our task sits. These compiler-feedback and executable-repair methods compile, execute, rank, or revise decoded programs. Our method instead uses those delayed outcomes to train a scorer that acts before final code decoding; the shared repair-loop comparison tests whether the pre- and post-decoding intervention points are complementary.

HPC and parallel language translation-specific work is closer to our evaluation domain. UniPar, LASSI, ParaCodex, HPC-Coder, and HPC-Coder-v2 study executable parallel-code generation, while adjacent work studies parallelization assistance, fine-tuning, repair, or agentic adaptation with LLMs (Nichols et al., 2024; Bitan et al., 2025; Dearing et al., 2024; Kaplan et al., 2026; Chaturvedi et al., 2025; Harel et al., 2023; Schneider et al., 2023; Kadosh et al., 2023c; Harel et al., 2025; Bolet et al., 2025a; Kadosh et al., 2024b; Schneider et al., 2024; Chen et al., 2024b; Bolet et al., 2025b). Source-to-source translation systems such as SYCLomatic, HIPIFY, cu2cl, and TransCL intervene through deterministic or compiler-style program transformation rather than latent guidance (Mueller-Albrecht, 2024; ROCm Developers,

2023; Martinez et al., 2011; Shi et al., 2025; Kadosh et al., 2024a; Harel et al., 2020). Our setting is complementary: we ask whether a frozen generator can be steered before code is decoded. Thus, it is independent of downstream repair and transformation methods and can be combined with downstream repair or optimization methods.

Most existing guidance methods intervene after the program text is produced. Verifier-guided reasoning, chain-of-thought prompting, search, and PRMs provide the conceptual template for allocating additional test-time computation with auxiliary signals (Wei et al., 2022; Nye et al., 2021; Cobbe et al., 2021; Lightman et al., 2024; Wang et al., 2024; Yao et al., 2023; Zhang et al., 2025b). For code, CodePRM and process-supervision-guided policy optimization show that execution-aware supervision can improve program generation (Li et al., 2025; Dai et al., 2024). The key difference is that these methods score textual reasoning steps, decoded programs, or execution feedback after the text has been produced, whereas our method intervenes earlier by scoring continuous latent prefixes before code is emitted.

Latent reasoning offers an opportunity for this different intervention point. Prior work motivates continuous hidden states as a useful reasoning substrate and shifts part of test-time computation from explicit token sequences into continuous hidden-state trajectories before returning to standard decoding (Hao et al., 2025; Xu et al., 2025; Yue et al., 2025; Zhang et al., 2025a; Deng et al., 2026; Zou et al., 2026). Macfarlane and Bonnet (2026) is especially relevant because it treats latent computation as a search problem. You et al. (2026) is closest methodologically, combining stochastic latent sampling with LatentRM-guided best-of- N and beam search. It learns from rollout correctness on mostly arithmetic tasks. Instead we train a separate PRM from executable and semantic feedback for non-unique code translation and guide a frozen generator before decoding in a domain that requires task-specific expertise. Concurrent work by Li et al. (2026) uses reward gradients to optimize fragile sparse-autoencoder states during token-level reasoning, while we instead select discrete branches in a dedicated latent phase before decoding. A more distant but relevant line of work is activation steering. These methods modify internal activations at inference time, often by adding a contrastively derived direction to control a specific behavior (Turner et al., 2023; Rinsky et al.,

Area	Representative work	Intervention point	Representation	Exec.?	Difference
Code generation and translation	Chen et al. (2021), Chen et al. (2022), Ahmad et al. (2023), Huang et al. (2023), Bitan et al. (2025)	Training data, decoded programs, or repair loops	Tokens or program text	Often	We intervene before final decoding by scoring latent prefixes while keeping the generator frozen.
Parallel and HPC translation	Nichols et al. (2024), Chaturvedi et al. (2025), Dearing et al. (2024), Kaplan et al. (2026), Mueller-Albrecht (2024), ROCm Developers (2023), Martinez et al. (2011), Shi et al. (2025)	Fine-tuning, source-to-source translation, or post hoc repair	Program text or symbolic code	Often	These methods operate on decoded code or deterministic migration rules rather than on continuous latent prefixes.
PRMs and verifiers	Cobbe et al. (2021), Lightman et al. (2024), Wang et al. (2024), Li et al. (2025), Dai et al. (2024)	Token steps or completed programs	Text	Often	We supervise hidden-state prefixes before code is emitted.
Latent reasoning and latent guidance	Hao et al. (2025), Xu et al. (2025), You et al. (2026), Yue et al. (2025), Zhang et al. (2025a), Deng et al. (2026), Macfarlane and Bonnet (2026), Zou et al. (2026)	Internal reasoning phase or latent search	Hidden states or latent programs	Usually no	We attach a smaller PRM to score candidate latent branches for executable parallel-code translation. We study execution-supervised prefixes for non-unique code translation, selecting discrete branches.

Table 5: Taxonomy of the closest neighboring literatures by intervention point.

2024). Code-focused variants use similar interventions for type prediction or language and library control (Lucchetti and Guha, 2025; Rahman et al., 2026) but not in parallel code. We draw only on the broader idea that hidden vectors can serve as intervention points. Unlike activation steering, however, we move the intervention from decoding-time control to the latent-reasoning phase, and we use a learned value model to compare task-conditioned continuations rather than applying a fixed steering direction.

B Supporting Results

Paired task win/loss/tie analysis. We report task-level paired win/loss/tie counts using ParaTrans tasks averaged over three runs, matching the unit used for the paired bootstrap analysis. For each task, we compute the guided-minus-unguided validation-rate difference. A task is a win if the difference is positive, a loss if it is negative, and a tie otherwise. Confidence intervals and p-values follow the task-level paired bootstrap and two-sided sign-flip tests described in Appendix C, using 100,000 resamples or Monte Carlo samples. Table 6 shows that most tasks are unchanged, but

Setting	Wins	Ties	Losses
No repair	20	50	6
+3-attempt repair	18	53	5

Table 6: Task-level paired win/loss/tie counts for PRM-guided latent reasoning against unguided latent reasoning on the 76-task ParaTrans test set.

among tasks whose outcome changes, PRM guidance more often improves than degrades validation in both the no-repair and repair settings.

Direction Split. Table 7 breaks down compilation and validation rates by translation direction. PRM guidance improves over unguided latent reasoning in three of the four directions, with the largest gain on Serial→OpenMP, smaller gains on the two CUDA/OpenMP directions, and only compilation gain on Serial→CUDA. We view this split as additional descriptive evidence that the aggregate result is not driven by only one direction, rather than as proof of direction robustness. This pattern suggests that the aggregate improvement reflects both syntactic and semantic effects, since some directions improve mainly in compilation,

Method	C→O	O→C	S→O	S→C
LR	55.6/40.7	40.4/31.6	63.3/35.0	40.4/24.6
LR-PRM	53.7/46.3	47.4/36.8	65.0/60.0	42.1/24.6

Table 7: Direction-wise compilation/validation rates on ParaTrans. Abbreviations: LR = latent reasoning, C = CUDA, O = OpenMP, S = Serial.

Method	No repair	+3 repair
Vanilla	144.86	399.89
Fine-tuned	144.52	328.45
Latent reasoning	145.67	315.93
Random branch selection	145.81	–
Latent PRM guidance	147.33	327.40

Table 8: Mean model-inference time per sample (seconds) for the Table 1 configurations. Measurements use batch size 1 on two H200 GPUs. Compilation and validation, run separately on A40 GPUs, are excluded.

some in validation, and some in both.

Test-time compute cost. We estimate the incremental cost of PRM guidance relative to unguided latent reasoning under the same $K = 12$ latent-step setting with the 70B generator and one final decoding pass. PRM guidance adds $B_{\text{test}} = 8$ branch scores per step, including the unperturbed branch, or $8 \cdot 12 = 96$ Qwen-Coder-7B prefix-scoring calls per sample. By parameter count, this is roughly 9.6 LLaMA-3.3-70B-sized forward-step calls, while avoiding additional 70B rollouts or extra decoded programs. This is only an approximate call-level normalization and does not account for implementation details. The decoded non-code portion is unchanged in our logs, averaging approximately 160 tokens in both settings, with high variance, and the final code is decoded once in each condition.

This overhead is small compared with post-decoding repair. A repair attempt requires another full 70B decoding pass over the program, which in our logs generates approximately 1,853 tokens per attempt on average. Thus, the three-attempt repair setting adds approximately $3 \cdot 1,853$ additional 70B decoding steps, compared with a much smaller auxiliary PRM-scoring budget. Despite this much smaller extra budget, latent PRM guidance without repair reaches 42.1% validation, exceeding unguided latent reasoning with three repair attempts at 36.4%. This comparison suggests that the gains are not explained simply by adding test-time computation.

Table 8 reports mean model-inference time for every Table 1 configuration. We treat the small

differences in inference time as run-to-run variation rather than evidence that any method is faster. Times with repair additionally reflect the number and length of repair attempts triggered by failed translations and do not include the compilation and execution time for each kernel.

These measurements should not be read as a full efficiency claim. They show only that PRM scoring adds modest overhead in our implementation and remains compatible with token-level repair.

Post-decoding text-PRM reranking. We also evaluate a post-decoding best-of-8 control that applies the same reward supervision after complete programs have been generated. We sample eight complete outputs from unguided latent reasoning without latent perturbations, using temperature 0.8 during decoding, and train a text PRM over completed answers with the same reward function used for latent PRM training. Candidates selected by the text PRM achieve 25.00% validation, while an oracle@8 over the same candidate sets reaches 36.84%. The oracle result indicates that the sampled sets sometimes contain validating programs, while the gap between oracle@8 and the text PRM shows that this control does not reliably identify them. Because the latent and post-decoding candidate sets are generated differently, this is not a matched compute or candidate-set comparison. However, even this oracle remains below latent PRM guidance at 42.10%. This suggests that the main gain is not explained by PRM supervision alone. Taken together, these results support the value of applying the reward model to latent prefixes before decoding in this setting.

C Hyperparameters and Reproducibility

Evaluation protocol. Main evaluation uses the 76-task ParaTrans test set, split across CUDA→OpenMP (18), OpenMP→CUDA (19), Serial→OpenMP (20), and Serial→CUDA (19). We report three-run mean executable validation rates with and without a three-attempt self-repair loop. In the repair setting, failed translations are re-generated from the previous code and the execution-environment feedback. We also report a separate 500-instance held-out branch-selection benchmark, where each saved latent prefix is paired with the original branch and two perturbed alternatives, and instances without a unique best branch are excluded.

For pairwise comparisons, we use task-level

paired analysis. For each task, we average the binary validation outcomes over the three runs for each method, then compute the guided-minus-unguided task difference. We report the mean of these task-level differences. Confidence intervals are computed using a paired nonparametric bootstrap over tasks with $B = 100,000$ resamples. P-values are computed using a two-sided paired sign-flip randomization test over the same task-level differences with 100,000 Monte Carlo samples.

We used two NVIDIA H200 GPUs for primary-model inference when creating the dataset and running experiments. For PRM training and code evaluations, we used two NVIDIA A40 GPUs. For the three-attempt repair setting, we feed the execution-environment error and the previously generated code back to the model, then ask it to regenerate the full code. The exact prompt templates are shown below.

PRM optimization. We train the PRM with the AdamW optimizer (Loshchilov and Hutter, 2019) in two stages. In the first stage, we train only the adaptation layer, which maps the primary model hidden dimension of 8192 to the PRM backbone hidden dimension of 3584, together with the value head, for half an epoch. In the second stage, we unfreeze the remaining layers. We train with an effective batch size of 6 and a learning rate of $1 \cdot 10^{-6}$.

Model selection. We selected LLaMA-3.3-70B for comparability with prior ParaTrans and UniPar-style evaluations and for compatibility with our COCONUT-style latent-reasoning implementation. We selected Qwen-Coder-7B as a smaller code-specialized auxiliary scorer, preserving a clear separation between the frozen generator and the learned PRM. We did not ablate generator or PRM size, so these choices should not be interpreted as necessary, sufficient, or optimal.

Noise-scale calibration. We set the perturbation standard deviation to $\sigma = 1.4$ based on a preliminary two-point calibration on the 60-sample development split. With $B_{\text{train}} = 3$ candidates at each of $K_{\text{train}} = 6$ latent steps, the mean within-candidate reward range, $\max_b S(y_t^{(b)}) - \min_b S(y_t^{(b)})$, averaged over samples and latent steps, was 0.060 for $\sigma = 0.7$ and 0.086 for $\sigma = 1.4$. The modest increase suggested greater separation among candidate outcomes, so we fixed $\sigma = 1.4$ for subsequent experiments. This comparison is not a systematic

Split / artifact	Count	Test-kernel overlap
PRM dev	60	0
Branch-selection PRM train	832	0
Final PRM train	952	0
Branch-selection source set	120	0
ParaTrans test	76	–

Table 9: Leakage audit for PRM development/training artifacts and the held-out source trajectories used to construct the branch-selection benchmark. The branch-selection source set row counts trajectories, not the 500 derived branch-choice instances; overlap with the ParaTrans test set is computed using (kernel_name, from_api, to_api) fingerprints.

sweep and does not establish that $\sigma = 1.4$ is optimal.

Split and leakage audit. We define each translation task by the fingerprint (kernel_name, from_api, to_api). The PRM development split contains the first 15 kernels in each translation direction, for 60 examples total. It is used only for hyperparameter selection. After hyperparameters are fixed, the final PRM is trained on the union of the PRM training and development pools. For the branch-selection experiment, we train a separate PRM after removing the 120 source trajectories used to construct the branch-choice instances. The 76-task ParaTrans test set is held out from all PRM training, development, trajectory collection, reward construction, and hyperparameter selection. As a stricter check, we also compare kernel names while ignoring translation direction; this overlap is also zero.

Reward construction. The reward target mixes executable and proxy components. We keep that distinction explicit because the benchmark metric in the main paper is executable validation only. For a terminal program y , let $C(y) \in \{0, 1\}$ indicate successful compilation, $E(y) \in \{0, 1\}$ indicate successful execution without runtime failure, and $A_j(y) \in [0, 1]$ denote optional semantic or proxy validators available for that sample. We compute

$$S(y) = 0.30C(y) + 0.25E(y) + 0.45 \frac{\sum_{j \in \mathcal{A}(y)} w_j A_j(y)}{\sum_{j \in \mathcal{A}(y)} w_j}$$

where unavailable validators are omitted and the remaining validator weights are renormalized. The optional validators include pass/fail integrated validation, checksum or output matching when available, benchmark-specific metrics, Jina code-embedding cosine similarity to the reference target

code (Günther et al., 2024), a GPT-5-mini judge score, and the variable-comparison heuristic described below. In the common case where pass/-fail validation, Jina similarity, and the GPT-5-mini judge are available, their effective contributions to $S(y)$ are 0.2423, 0.1038, and 0.1038, respectively, in addition to the fixed 0.30 compilation and 0.25 execution weights.

GPT-5-mini judge. The judge is used only for PRM target construction and never for final benchmark scoring. For each candidate rollout, GPT-5-mini receives the source program, and the generated target-language program. It is asked to assess semantic equivalence, target-API correctness, and likely behavioral preservation, then return a bounded scalar score in $[0, 1]$ with a short justification. We use the scalar score as one optional validator in the renormalized reward above.

Judge Prompt Template

Role. You are an expert in HPC code translation evaluation. Compare the original and translated code section below. Assign a score between 0.0 and 1.0 based only on syntax correctness, faithfulness, and overall quality.

Scoring rubric.

- 0.00–0.20: incorrect or not on the right track
- 0.21–0.50: major issues
- 0.51–0.80: mostly correct with minor issues
- 0.81–1.00: strong and faithful

Rules.

- Focus on logic and correctness.
- Ignore style, but pay attention to pragma placement in OpenMP directives.
- If the code is incomplete because of cut off text or missing includes, assume the most reasonable completion for scoring.

Inputs.

- Kernel name: {kernel_name or 'UNKNOWN'}
- Original code: {original_code[:8000]}
- Translated code: {translated_code[:8000]}

Required output. Return strict JSON only:

```
{"score": <float 0.0-1.0>, "reason":
"<short reason>"}
```

Variable-comparison heuristic. The variable-comparison heuristic is intended to catch semantic shortcuts not exposed by unit tests alone. GPT-5-mini first identifies variables in the source and generated programs that represent the core computed quantity for the built-in test. We then instrument the reference and generated programs by adding print statements for the selected variables, execute both programs on the same test input, and compare the observed final values.

Variable Identification Prompt Template

Role. You are a precise static code analysis agent.

Hard rules.

- Static analysis only: do not suggest edits, do not modify code, and do not add instrumentation.
- Use only the code text provided in the request.
- Output must be valid JSON only, with no markdown or extra text.

Input fields.

- kernel_name: string
- original_code: string, possibly empty, containing concatenated files prefixed by // File: <filename>
- translated_code: string, possibly empty, using the same format

Goal. For each codebase, identify 1 to 3 variables that represent the most semantically central computed results of the program or kernel.

- Return multiple variables only when there are genuinely multiple important result variables.
- If there is one clear result variable, return only that one.
- Order variables by importance, with the most central first.

Critical distinction.

- Do not choose status flags, timing variables, loop counters, argument parsing variables, or configuration constants unless no computed result variable exists.
- Prefer numeric result variables and, failing that, the primary output buffer updated by the kernel or parallel region.

Priorities.

1. **Reported numeric result scalar.** Choose a non-boolean numeric scalar printed or logged as the main result, such as a checksum, error, norm, sum, energy, or output summary.
2. **Derived numeric result variable just before output.** If the printed value is an expression, choose the stored numeric variable closest to the output site.
3. **Primary output buffer, array, or pointer.** If there is no clear scalar result, choose the main computed output buffer.
4. **Last resort.** If nothing else fits, choose the single variable most central to the computed output, avoiding booleans and timing variables when possible.

Type, missing, ambiguity, and not-found rules.

- Return the declared type exactly as written in code; use UNKNOWN only when the type cannot be determined.
- If original_code is missing, set source status to MISSING with null fields.
- If translated_code is missing, set target status to MISSING with null fields.
- If multiple candidates satisfy the same highest applicable priority, still choose one top candidate, mark the status as AMBIGUOUS, and list alternatives in notes.
- Use NOT_FOUND only when code exists but no result variable can be identified under priorities 1–4.
- The variables array must contain 1 to 3 objects, ordered by importance.

Inference prompt. The following prompt is used for the initial (non-repair) translation pass and during dataset creation. The system message establishes the model’s role as an HPC translation ex-

pert, and the user template supplies the source API, target API, expected output file count and names, and the source code.

Regular Inference Prompt Template

System. You are an HPC expert specializing in translating between parallel programming APIs.

User template. For each kernel code provided, think about it step by step, and then translate it from {from_api} to {to_api}. The target {to_api} implementation is spread across {n} files: {names_str}. Provide the complete code in {to_api}. Do not truncate or use ellipses. Do not change the main function. Ensure correctness. All function names must match. The code to translate:

{source_code}

Three-Attempt repair prompt. The repair prompt is used in the three-attempt regeneration setting after a failed compilation or execution attempt. It provides the previous error trace together with the broken translation and asks the model to emit a complete corrected translation in a strictly parseable format.

Repair Prompt Template

Context. The previous translation attempt (iteration {iteration - 1}) failed with the following error:

```
- BEGIN ERROR -  
{error_text}  
- END ERROR -
```

Previous broken translation.

```
- BEGIN PREVIOUS ATTEMPT -  
{prev_code}  
- END PREVIOUS ATTEMPT -
```

Instruction. Please fix the error and produce a corrected translation.

Output format requirements.

- Provide the complete code in {to_api}.
- Do not truncate or use ellipses. Do not change the main function.
- Ensure correctness. All function names must match.
- Wrap the code in a single fenced block beginning with ```cpp and ending with ```. You may instead begin the block with ```c, ```cuda, ```omp, or ```cu if that better matches {to_api}.
- If the translation spans multiple files, place a // File: <filename> marker at the start of each file's content inside the same fenced block, using the exact target filenames.
- Do not include any prose after the closing fence.