LLM-Based Porting of Optimized C++ to CUDA Through Deoptimization and Reoptimization

Daichi Mukunoki[1,*], Ryo Mikasa[1], Shun-ichiro Hayashi[1], Tetsuya Hoshino[1] and Takahiro Katagiri[1]

[1]Nagoya University, Furo-cho, Chikusa-ku, Nagoya 464-8601, Japan

Abstract

When porting high-performance computing (HPC) code from CPU to GPU, CPU-oriented optimizations may obstruct LLM-based CUDA translation. We design and evaluate a Deopt-Reopt workflow that first simplifies the input C++ code and then retranslates and reoptimizes it for CUDA, comparing it against direct translation (Direct) on twelve HPC kernels with two LLMs (`gpt-oss-120b` (O120) and `qwen-3-235b-a22b-instruct-2507` (Q235)) in Single-shot (one pass) and Iterative (repeated refinement) settings. In Single-shot, among 18 testable cases Deopt-Reopt was significantly faster among successful trials (after BH-FDR correction) in five — most clearly for `conv2d`, where CPU- and GPU-oriented designs diverge — but Direct was faster in three, so removing CPU-specific optimizations is not universally beneficial. An exploratory Direct-3 control that equalizes the LLM-call count left Deopt-Reopt ahead in only four of nineteen testable cases, with Direct-3 ahead in four others. In Iterative, repeated generation and repair narrow the mode gap — markedly so for O120 — while Q235 retains large Deopt-Reopt advantages on `conv2d`, `ddgemm`, and `bgemm`. Deopt-Reopt's effect on feasibility is also mixed — sharply higher for some kernels Direct rarely compiles, lower for others — and because performance is conditioned on successful trials, the benefit is conditional, not a guaranteed end-to-end gain. Overall, Deopt-Reopt is effective but non-universal, with gains depending on the kernel, model, search budget, and success rate.

Keywords

Large Language Models, Code Translation, CUDA, HPC, Deoptimization

1. Introduction

Scientific computing code is often developed from a simple implementation and then optimized for a specific architecture, but the simple version may be lost or never maintained. When such CPU-oriented code is ported to a GPU, handwritten optimizations such as SIMD expansion, blocking, and data-layout transformations may obscure the algorithmic structure needed for a different parallelization strategy. In such cases, deoptimization (removing architecture-specific optimizations before translation) may improve portability, although it may also discard structure that direct translation could have preserved.

In recent years, the code generation capability of large language models (LLMs) has improved rapidly, and their application to automatic translation and optimization of HPC code has been actively studied [1, 2, 3]. Most existing work, however, focuses on direct translation from CPU code to GPU code or on one-way optimization. The effectiveness of removing architecture-specific optimizations before LLM-based GPU porting has not been sufficiently investigated.

This paper studies the effectiveness of deoptimization in LLM-based GPU porting of CPU code from C++ to CUDA. We compare two workflows: Direct, which translates the CPU code straight to CUDA, and Deopt-Reopt, which first deoptimizes and then translates and reoptimizes. We evaluate both in a Single-shot (one pass) and an Iterative (repeated repair and selection) setting, using two LLMs, OpenAI's `gpt-oss-120b` (O120) and Alibaba Cloud's `qwen-3-235b-a22b-instruct-2507` (Q235), on twelve HPC kernels. The analysis focuses on when deoptimization helps, when reoptimization recovers performance, and when Iterative refinement narrows the gap between the workflows. We use feasibility to mean that translated code compiles, runs, and passes validation (functional success, which can

GeCoIn 2026: Generative Code Intelligence Workshop, co-located with the 35th International Joint Conference on Artificial Intelligence (IJCAI-ECAI 2026), August 15–17, 2026 — Bremen, Germany

*Corresponding author.

mukunoki.daichi.p2@f.mail.nagoya-u.ac.jp (D. Mukunoki)

include correct-but-slow outputs; we do not impose a practical-performance threshold), and report performance only over successful trials (successful-trial performance), separately from feasibility.

The contributions of this paper are as follows: (1) a Deopt-Reopt workflow that simplifies optimized C++ before CUDA translation and reoptimization; (2) a comparison with Direct and a call-count-matched Direct-3 control; (3) an evaluation on twelve HPC kernels with two LLMs in Single-shot and Iterative settings; and (4) a characterization of when deoptimization helps or Iterative search narrows the gap.

2. Related Work

CPU-to-CUDA porting has long been studied through compiler- and directive-based approaches such as OpenMP-to-GPGPU [4] and PPCG [5], which rely on analyzable dependences or annotations rather than LLM-based translation of simplified CPU code. LLM-based GPU code generation is also active: omniCUDA [6] targets OpenMP-to-CUDA, KernelBench [7] evaluates kernel generation, and CUDA-LLM [8] and CudaForge [9] use iterative generation and hardware feedback for CUDA optimization.

LLMs have also been studied for HPC code translation, including Fortran→C++ translation data and multi-turn translation [3, 10], cross-language and C-to-CUDA translation corpora [2, 11], and Fortran→C++/C++→CUDA dialogue-based data generation with compilation, execution, and unit-test feedback [12]. These demonstrate LLM-based porting but do not evaluate removing CPU-specific optimizations before retranslation and reoptimization. Work on LLM-based refactoring, deobfuscation, and input-token reduction [13, 14, 15] motivates simplification; we evaluate LLM-based deoptimization specifically for C++→CUDA porting.

3. Common Evaluation Setting

The task is LLM-based porting of heavily CPU-optimized C++ HPC kernels to CUDA: each input is a hand-optimized C++ implementation for aarch64/Neoverse-V2 using NEON intrinsics or other CPU-oriented optimizations, which we translate to CUDA and evaluate for feasibility and performance.

Experiments were performed on a GH200 system with a 72-core Arm Neoverse-V2 CPU and an NVIDIA H100 GPU. The software stack is NVHPC 25.9, CUDA 12.6, cuDNN 9.5.1, NVPL 25.9, and ArmPL 25.07.1. CPU C++ is compiled with `nvc++` (`-O3 -fopenmp -tp=neoverse-v2 -std=c++17`) and CUDA with `nvcc` (`-O3 -arch=sm_90 -std=c++17`). Each run uses all CPU cores with threads pinned and their memory bound to the local NUMA node. Each candidate has one warm-up and five timed runs (minimum reported); GPU timings use CUDA events after data placement and exclude host-device transfers and validation.

We evaluate two open-weight LLMs independently in all workflows: `gpt-oss-120b` (O120) [16], a 117B-parameter MoE (~5.1B active) with controllable reasoning, and `qwen-3-235b-a22b-instruct-2507` (Q235) [17], a ~235B-parameter MoE (~22B active) general-purpose non-thinking instruct model. For inference, both models are served through the Cerebras inference API[1] with temperature 0.5, `max_completion_tokens=16384`, and no fixed random seed (O120 uses `reasoning_effort=medium`, the gpt-oss default); serving-side numerical precision is provider-controlled and was not recorded.

Input and generated code are normalized by removing C/C++ comments and blank lines while preserving string literals; all reported LOC values use this normalized code.

Performance differences between Direct and Deopt-Reopt are tested by a two-sided Mann-Whitney U test (α=0.05) over successful trials, separately from feasibility; cases with fewer than three successes are not tested. Unadjusted p-value marks are exploratory, while primary significance uses the Benjamini–Hochberg false discovery rate (BH-FDR) correction [18] (α=0.05) separately for Single-shot and Iterative. The Direct-3 control is auxiliary, so D+R vs Direct-3 uses unadjusted p-values only. Feasibility is reported separately, as per-mode success rates in Table 2.

We use the twelve HPC kernels in Table 1. B denotes batch size, M, N, K principal dimensions, C input channels, H, W spatial sizes, BS block size, and T iterations. Before observing results, we group kernels

[1] https://www.cerebras.ai/inference

Table 1
Benchmark kernels and evaluation settings. Groups are divergent-style (Div.), dependency/convention-dominated (Dom.), and shared-style (Shr.); '*' denotes hand-coded or different-precision reference performance. Tol. is the per-element relative-error tolerance; Conv2D also accepts a 10^{-4} absolute error. GF/s denotes GFlops/s.

Group	Kernel	Problem	CPU-input structure and GPU-side interpretation	LOC	Unit	Input perf	Ref. perf CPU	Ref. perf GPU	Tol.
Div.	Conv2D	2D convolution (FP32), N=16, C=32, K=64, H=W=96	Filter-transpose cache, blocking, NEON SIMD; GPU tiles with shared memory.	142	GF/s	2328	474.9	24647	10^{-3}
Div.	BFFT	Batched complex FP64 FFT, B=8192, N=1024	Radix-4 butterflies, bit reversal, twiddles; small FFTs remap differently on GPU.	123	GF/s	598.9	596.4	4907	10^{-8}
Div.	Softmax	Row-wise FP64 softmax, M=1024, N=32768	Handwritten NEON exp, SIMD, buffering; maps to warp/block reductions.	67	GB/s	185.6	210.3*	1235	10^{-8}
Div.	BGEMM	Batched FP64 GEMM, B=1024, M=N=K=128	NEON FP64 microkernel, blocking, prepacking; GPU uses batch/tile parallelism.	210	GF/s	1935	159.4	32848	10^{-10}
Dom.	DFSpMM	Sparse–dense, double-float (DF) arithmetic (48-bit significand), CSR, 32 nnz/row, M=K=24576, N=128	Sparse access and DF arithmetic dominate; structure constrains the GPU mapping.	148	GF/s	17.3	735.3*	2181*	10^{-7}
Dom.	FFT	Single large complex FP64 FFT, N=4194304	Radix-4 butterflies, bit reversal, twiddles, NEON wrapper; stage dependences dominate.	519	GF/s	209.8	170.0	3154	10^{-8}
Dom.	BTDMA	Batched FP64 tridiagonal solve, B=32768, N=256	Parallel Thomas, two-element elimination; forward/backward dependences dominate.	92	GB/s	295.7	159.4	141.1	10^{-8}
Dom.	DDGEMM	Dense GEMM, double-double (DD) arithmetic (106-bit significand), M=N=K=256	DD arithmetic dominates runtime over parallelization.	86	GF/s	44.1	4.3*	569.9*	10^{-22}
Shr.	Stencil	5-point FP64 Jacobi stencil, N=3072, T=100	NEON SIMD, buffering, locality optimization; grid-update strategy reusable on GPUs.	116	GB/s	1404	970.8*	2789*	10^{-8}
Shr.	SpMM	Sparse–dense (FP64), CSR, 32 nnz/row, M=98304, K=524288, N=128	p-loop unrolling, output tiling; CSR row-wise processing is a useful GPU hint.	98	GB/s	414.6	203.9*	752.7	10^{-10}
Shr.	GEMM	Dense FP64 GEMM, M=N=K=1024	NEON FP64 microkernel, blocking, prepacking; data-movement intent stays useful.	259	GF/s	305.5	2653	40895	10^{-8}
Shr.	SpMV	FP64 BSR banded sparse matrix–vector, M=K=524288, BS=4	Block-sparse format, expansion, vectorization are useful GPU hints.	63	GB/s	1508	685.6	2527	10^{-10}

by how CPU-oriented structure relates to the natural GPU strategy: divergent-style (CPU optimizations obscure a different GPU mapping), dependency/convention-dominated (constrained by algorithmic dependences or numerical conventions), and shared-style (CPU-side structural hints still help on the GPU). This is an author-defined interpretive framework, not a deterministic predictor.

Table 1 reports LOC, input/reference performance, units, and the per-kernel validation tolerance. Input perf is the measured performance of the hand-optimized CPU C++ input (the porting source); Ref. perf is the vendor-library or hand-coded reference on the CPU and the GPU. Outputs are validated against vendor libraries where available (NVPL/ArmPL on the CPU; cuDNN/cuFFT/cuBLAS/cuSPARSE on the GPU) and otherwise against high-precision or hand-coded references (e.g., binary128 for DDGEMM and an FP64 reference for DFSpMM). Correctness is checked by benchmark-specific validation drivers

at the benchmarking problem sizes, using per-element relative error $|x - x_{\text{ref}}| / \max(|x_{\text{ref}}|, 10^{-12})$ against the listed tolerance; Conv2D additionally accepts a maximum absolute error of 10^{-4}. DFSpMM and DDGEMM use double-word arithmetic [19], representing a value as an unevaluated sum of two native words: double-double (DD) uses two FP64 words (a ~106-bit significand) and double-float (DF) is the FP32-word counterpart. DDGEMM and DFSpMM use the operation-count definitions of their GEMM and SpMM counterparts. Except for FP32 Conv2D, interfaces are FP64.

4. Single-shot Workflow

4.1. Method

We first use a one-pass Single-shot workflow to observe the basic effect of LLM-based deoptimization under three modes.

- Deopt-Reopt (D+R): input code → deoptimization → CUDA translation → reoptimization, using three LLM calls.
- Direct (D): input code → CUDA translation → reoptimization (no deoptimization), using two LLM calls.
- Direct-3 (D3, control): Direct translation followed by reoptimization v1 and reoptimization v2, matching the three LLM calls of D+R; D3 trials are newly sampled and are not a subset of Direct trials. If translation does not produce passing code (PASS), the trial stops; if v1 fails, v2 is skipped, and if v2 fails, v1 is adopted.

Direct-3 tests whether a D+R successful-trial performance advantage can be explained only by the larger number of LLM calls; it gives Direct an extra reoptimization pass and serves as an auxiliary call-count control for successful-trial performance, not a success-rate control. Because Direct-3 spends its extra call on reoptimization rather than retrying translation, it does not test whether the same budget used for additional translation or repair attempts would close Direct's feasibility gap. The deoptimization phase removes architecture-specific optimizations while preserving the function signature, problem sizes, and numerical results within tolerance; only code passing the same validation driver and tolerance (Table 1) as the input C++ is accepted. The translation phase converts the (possibly deoptimized) C++ code to a complete CUDA implementation preserving the entry-point contract: the generated function takes device pointers and implements only device-side computation, while host allocation and host-device transfers are handled by the benchmark harness. Reoptimization then improves the CUDA code for the target GPU without prescribing specific techniques. If a required stage fails to produce passing code, the trial is counted as a failure; when a later reoptimization stage fails, the last passing CUDA code is retained.

Prompts specify the phase objective, input code, source/target languages, contract and numerical constraints, and previous-stage code or evaluation results; target hardware information is given to the translation and reoptimization prompts but not the deoptimization prompt (a hardware-agnostic step). We run 50 independent trials per mode and model (150 per kernel-model pair, 3600 in total) and evaluate success rate and the distribution/median of final performance over successful trials; following the common evaluation setting of Section 3, the three modes share the LLM, temperature, prompts, problem sizes, validation, and hardware.

4.2. Results

Figure 1 shows the Single-shot results on a 3×4 kernel grid; each panel places the two models side by side, with boxplots for Direct, Direct-3, and Deopt-Reopt.

Success rates vary widely by model and kernel. O120 obtained successful trials for all twelve kernels, whereas Q235 had 0/50 successes in both Direct and Deopt-Reopt for `spmm` and `gemm`. At the same time, Q235 achieved near-complete success on `spmv` under Direct, with Deopt-Reopt at a high but lower rate. Feasibility thus depends more on model–kernel compatibility than on model strength alone.

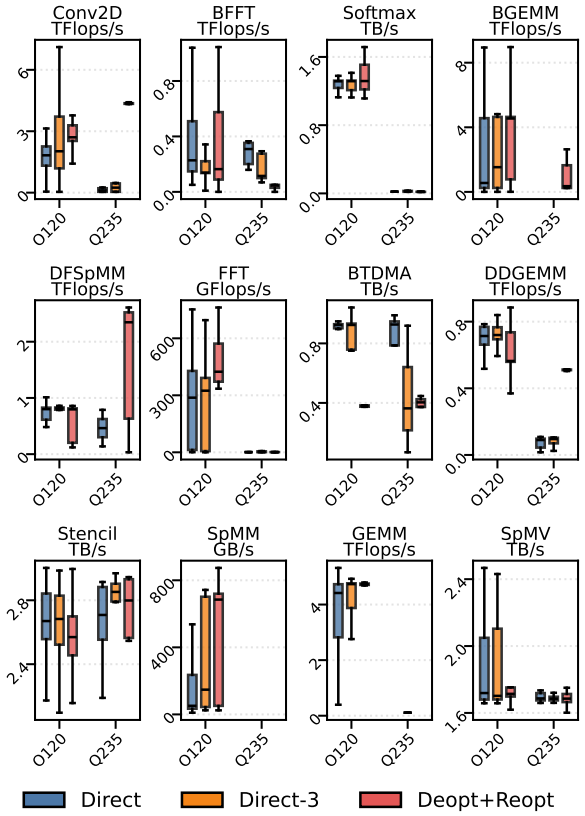

Figure 1: Single-shot final performance (50 trials). Boxplots use successful trials only.

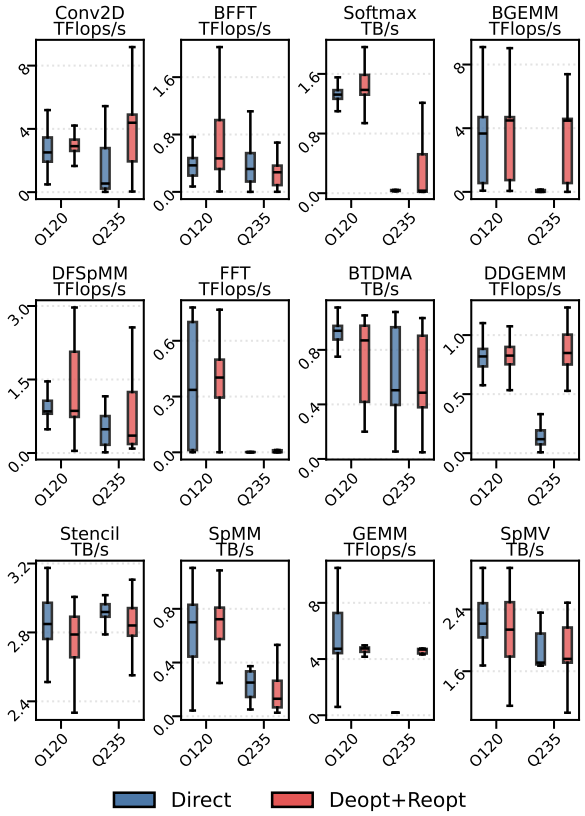

Figure 2: Iterative final performance (50 trials). Boxplots use successful trials only.

Accordingly, Table 2 should be read with success rate and successful-trial performance together, not median performance alone. Feasibility shifts are strongly model-dependent: Deopt-Reopt sharply raises Q235 success on most kernels, consistent with Q235 being anchored to the input CPU implementation, whereas for O120 the effect is mixed and can even lower feasibility.

Across the two models, among 24 cases the Mann-Whitney U test was applied to 18 testable cases. After BH-FDR correction within the Single-shot main series, eight cases were significant: five favored Deopt-Reopt (Q235 `conv2d` and `ddgemm`, O120 `conv2d`, `spmm`, and `gemm`) and three favored Direct (O120 `ddgemm` and `btdma`, Q235 `bfft`). Magnitudes vary widely — DR/D ratios from over 40× down to ≈0.15× (Direct faster) — so the performance effect is genuinely mixed rather than uniformly favoring Deopt-Reopt. These cases span all three benchmark groups rather than concentrating in one: only `conv2d` favors Deopt-Reopt on both models, while `ddgemm` reverses between models (Deopt-Reopt for Q235, Direct for O120). The benefit is thus kernel- and model-specific rather than a property of a single kernel class, and on some dependency-dominated kernels (O120 `ddgemm` and `btdma`) discarding the CPU structure instead hurts.

To check whether the D+R successful-trial performance advantage merely reflects its extra LLM call, we ran the auxiliary Direct-3 control (three LLM calls, no deoptimization; a performance, not success-rate, control) for all 24 cases with 50 trials each, of which 19 were testable. Deopt-Reopt was significantly faster than Direct-3 (unadjusted) in four cases, with median DR/D3 ratios 1.3/1.3/18/5.4 for O120 `conv2d`, O120 `fft`, Q235 `conv2d`, and Q235 `ddgemm`; conversely Direct-3 was faster in four others, with D3/DR ratios 1.3/1.0/2.4/2.5 for O120 `ddgemm`, `dfspmm`, `btdma`, and Q235 `bfft`. Of the five Single-shot main-series Deopt-Reopt advantages, three (`conv2d` on both models and Q235 `ddgemm`) remained significant against Direct-3, so part of the advantage reflects the deoptimization phase itself rather than call count alone; on the other kernels the call-count-matched control no longer favors Deopt-Reopt. Full Direct-3 statistics are in the released artifact.

Overall, Single-shot Deopt-Reopt most clearly helps `conv2d` (both models) and sharply raises Q235 feasibility, but the successful-trial performance comparison is mixed — Direct is significantly faster on several kernels (O120 `ddgemm` and `btdma`, Q235 `bfft`) — and Deopt-Reopt costs about 50% more LLM calls than Direct.

5. Iterative Workflow

5.1. Method

Single-shot is sensitive to one LLM output, so we also use an Iterative workflow that repeatedly plans, generates, evaluates, repairs, and selects candidates. This compares the two modes under a search process closer to practical HPC porting, where both receive the same repair and iteration opportunities.

Each phase (deoptimization, translation, reoptimization, and direct translation) runs for up to three generations. In each generation, the LLM first acts as a project manager (PM) proposing a strategy from the current code and results; three programmer (PG) calls then generate complete implementations; generated code is compiled, executed, and validated; each PG may repair its code once on error; and the best correct candidate seeds the next generation or the final result. Direct omits deoptimization and passes the input code directly to translation. PM, PG, and repair calls use the same LLM and parameters, with role differences expressed only in the prompt. Because the three PGs share one PM strategy, tests use the final trial-level performance value rather than treating PGs as independent samples.

For deoptimization selection, we use LOC as a simple and interpretable proxy for simplification: it reflects removal of SIMD expansion and loop unrolling, but not portability as a whole. Among candidates that pass validation, deoptimization adopts the smallest-LOC code subject to being smaller than the input CPU LOC, whereas reoptimization adopts the smallest-runtime code. When no valid candidate is smaller than the input, the input is carried forward unchanged; this occurred in only 1.0% (11/1110) of successful Iterative Deopt-Reopt trials, so Deopt-Reopt essentially performs genuine deoptimization. If any phase fails to obtain a valid candidate, the trial is counted as a translation failure. Each Iterative trial is more expensive than Single-shot; we run 50 trials per kernel-mode for both models, evaluating translation success rate, execution performance, and LOC. The twelve kernels therefore produce 2400 Iterative trials, with Deopt-Reopt's extra-phase cost included; success-rate comparisons are within-model.

5.2. Results

Results are shown in Figure 2, with performance and LOC progression in Figure 3.

In Iterative, repeated generation, repair, and selection partially mitigated the Single-shot feasibility problems, which were concentrated in Q235: Deopt-Reopt recovered several Q235 kernels with few or no Single-shot successes, while O120 retained successful trials for all twelve kernels. Of the 24 cases, 23 were testable; after BH-FDR correction within Iterative seven were significant — five favored Deopt-Reopt (Q235 `conv2d`, `ddgemm`, and `bgemm`; O120 `bfft` and `softmax`) and two favored Direct (O120 and Q235 `stencil`). Only the three Q235 cases carry a large effect: `conv2d` and `ddgemm` reach DR/D median ratios of 8× and 7×, and for `bgemm` the gap is feasibility-driven (Direct succeeds in only 7/50 trials, mostly near-failures). The remaining four significant cases are far smaller — O120 `bfft` at 1.3×, and O120 `softmax` and the two `stencil` cases within a few percent. For O120, then, iteration largely closes the Single-shot gap by letting Direct also adapt toward the GPU. For Q235 `fft`, both workflows have median performance near 1 GFlops/s; the outputs pass validation but stay far below the GPU reference, i.e., they do not reach high performance. A large median gap need not be significant: several sizeable median differences in Table 2 are not statistically significant, so medians must be read together with the test.

Zero-success Single-shot recovery is most visible for Q235, where PM-driven strategy updates and one allowed repair provide a recovery path. Deopt-Reopt markedly raises Q235 Iterative feasibility, recovering several kernels (`bgemm`, `spmm`, `gemm`, `dfspmm`) that Direct rarely compiles.

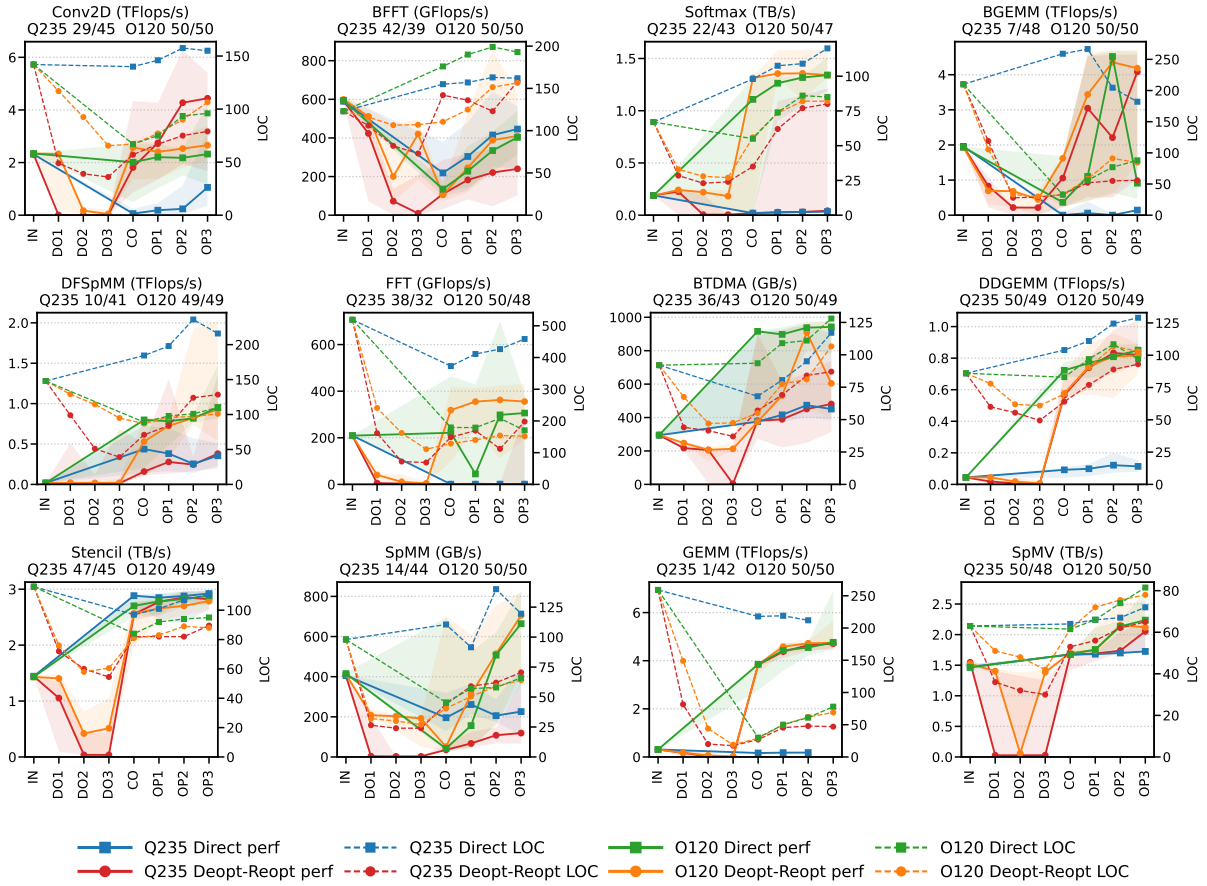

Figure 3: Performance and LOC progression in Iterative (50 trials) over three generations. Solid line with band: median and Q1–Q3 (left axis); dashed line: LOC (right axis). Marker shape encodes mode (square: Direct, circle: Deopt-Reopt) and color encodes model. Panel titles: per-model success counts.

Overall, Iterative narrows the mode difference markedly for O120, whereas Q235 retains substantial Deopt-Reopt advantages on `conv2d`, `ddgemm`, and `bgemm`.

Table 2 summarizes the win/loss pattern. Because these medians are conditional on success, end-to-end performance also depends on the success rate: weighting a conditional median by success rate can reverse the favored mode, so the medians should be read together with the success columns. The three-group classification has clear exceptions — the shared-style kernel O120 `spmm` shows a large Single-shot Deopt-Reopt advantage, the divergent-style kernel Q235 `bfft` favors Direct, and on the divergent-style O120 `conv2d` Direct catches up under Iterative — so it is only an interpretive framework.

6. Conclusion

We evaluated Deopt-Reopt for LLM-based C++-to-CUDA porting on twelve HPC kernels. In Single-shot, Deopt-Reopt was significantly faster among successful trials in five testable cases — most clearly for `conv2d`, where the CPU optimization structure diverges from the natural GPU strategy — but Direct was faster in three (O120 `btdma` and `ddgemm`, Q235 `bfft`), so removing CPU-specific optimizations is not universally beneficial; an exploratory Direct-3 control left the call-count-matched comparison roughly balanced. In Iterative, repeated generation and repair narrowed the gap, markedly for O120; Q235, however, retained large Deopt-Reopt advantages on `conv2d`, `ddgemm`, and `bgemm` (after BH-FDR correction, seven of 23 testable cases stayed significant, but only these three carry a large effect). Because performance is measured over successful trials, a lower success rate can offset a conditional gain, so the evidence supports a conditional-performance benefit, not a guaranteed end-to-end one. In sum,

Table 2
Comparison of Direct (D) and Deopt-Reopt (DR). Groups (Div./Dom./Shr.) as in Table 1. Success is over all trials; performance is the median over successful trials. A marker on a median indicates that the corresponding mode is significantly faster among successful trials: * at unadjusted p<0.05, and † also after BH-FDR correction (within the Single-shot main series, or within Iterative). § marks a median computed over fewer than three successful trials (not statistically tested). Performance units follow the Unit column in Table 1.

Group	Bench	O120								Q235							
		Single-shot (50 trials)				Iterative (50 trials)				Single-shot (50 trials)				Iterative (50 trials)			
		Succ. (%)		Med perf		Succ. (%)		Med perf		Succ. (%)		Med perf		Succ. (%)		Med perf	
		D	DR	D	DR	D	DR	D	DR	D	DR	D	DR	D	DR	D	DR
Div.	Conv2D	64	70	1824	2704*†	100	100	2512	2908*	12	96	106	4366*†	58	90	545	4389*†
	BFFT	48	52	228	164	100	100	370	467*†	12	66	309*†	46	84	78	319	273
	Softmax	94	90	1315	1317	100	94	1322	1385*†	2	52	20§	18	44	86	35	34
	BGEMM	88	90	552	4546*	100	100	3666	4485	0	94	—	354	14	96	1.79	4474*†
Dom.	DFSpMM	86	82	801	797	98	98	850	860	4	88	463§	2346	20	82	485	355
	FFT	42	56	288	424	100	96	336	402	2	20	0.909§	0.958	76	64	1.08	0.979
	BTDMA	92	44	923*†	380	100	98	940*	869	8	10	927	404	72	86	504	486
	DDGEMM	66	82	714*†	564	100	98	819	826	42	80	89	510*†	100	98	119	848*†
Shr.	Stencil	82	58	2670*	2570	98	98	2849*†	2788	36	62	2708	2799	94	90	2919*†	2840
	SpMM	66	54	51	685*†	100	100	700	723	0	0	—	—	28	88	251	130
	GEMM	84	66	4416	4731*†	100	100	4732	4748	0	0	—	—	2	84	181§	4676
	SpMV	68	60	1719	1714	100	100	2216	2139	96	72	1687	1686	100	96	1711	1760

Deopt-Reopt is an effective but non-universal technique for LLM-based GPU porting, with gains that depend on the kernel, the model, the search budget, and the success rate.

The study is limited to two LLMs, one GPU environment, twelve kernels, a single prompt-template family, and LOC as a proxy for simplification. Correctness was validated at the benchmark problem sizes; generalization to unseen sizes, layouts, sparsity patterns, and boundary cases remains untested. A fully failure-inclusive analysis of the conditional comparison (best-of-N under a fixed call budget, bootstrap intervals) is left to future work. The small per-case samples also yield low statistical power, leaving some large median gaps non-significant. Future work includes larger codebases, more model families, automatic control of deoptimization degree, simplification metrics beyond LOC, a prompt-only deoptimization baseline that omits the explicit intermediate C++, and an ablation isolating the deoptimization stage while holding downstream LLM calls fixed; we report median ratios only as a rough magnitude indicator. The experimental code and data are available in a public repository[2].

Acknowledgments

This work was supported by the "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN)" in Japan (Project ID: jh260065), JSPS KAKENHI JP25K24387, and the JST Next-Generation Edge AI Semiconductor Research and Development Project JPM-JES2511.

Declaration on Generative AI

During the preparation of this work, the authors used Claude Opus 4.8 (Anthropic) in order to: Drafting content, Text Translation, Paraphrase and reword, Improve writing style, Abstract drafting, Grammar and spelling check, Citation management, Formatting assistance, Peer review simulation, and Content enhancement, and to assist with writing and debugging the experimental and analysis code. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

[2] https://github.com/mukunoki/deopt_reopt

References

[1] A. Dhruv, A. Dubey, Leveraging large language models for code translation and software development in scientific computing, in: Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '25), 2025, pp. 1–9. doi:10.1145/3732775.3733572.

[2] A. TehraniJamsaz, A. Bhattacharjee, L. Chen, N. K. Ahmed, A. Yazdanbakhsh, A. Jannesari, Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming, in: Advances in Neural Information Processing Systems, volume 37, 2024, pp. 100965–100999.

[3] L. Chen, B. Lei, D. Zhou, P.-H. Lin, C. Liao, C. Ding, A. Jannesari, Fortran2cpp: Automating fortran-to-c++ translation using llms via multi-turn dialogue and dual-agent integration, arXiv preprint arXiv:2412.19770 (2024).

[4] S. Lee, S.-J. Min, R. Eigenmann, Openmp to gpgpu: A compiler framework for automatic translation and optimization, in: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009), 2009, pp. 101–110.

[5] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, F. Catthoor, Polyhedral parallel code generation for cuda, ACM Transactions on Architecture and Code Optimization 9 (2013) 1–23. Art. 54.

[6] M. Gruzewski, Automated transformation of openmp to cuda kernels using ai models, Procedia Computer Science 270 (2025) 3352–3361.

[7] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, A. Mirhoseini, Kernelbench: Can llms write efficient gpu kernels?, in: Proceedings of the 42nd International Conference on Machine Learning (ICML 2025), volume 267 of *Proc. Mach. Learn. Res.*, 2025, pp. 47356–47415.

[8] W. Chen, J. Zhu, Q. Fan, Y. Ma, A. Zou, Cuda-llm: Llms can write efficient cuda kernels, arXiv preprint arXiv:2506.09092 (2025).

[9] Z. Zhang, R. Wang, S. Li, Y. Luo, M. Hong, C. Ding, Cudaforge: An agent framework with hardware feedback for cuda kernel optimization, arXiv preprint arXiv:2511.01884 (2025).

[10] N. R. Ranasinghe, S. M. Jones, M. Kucer, A. Biswas, D. O'Malley, A. Most, S. L. Wanna, A. Sreekumar, Llm-assisted translation of legacy fortran codes to c++: A cross-platform study, in: Proceedings of the 1st Workshop on AI and Scientific Discovery: Directions and Opportunities, Assoc. Comput. Linguistics, 2025, pp. 58–69. doi:10.18653/v1/2025.aisd-main.6.

[11] Y. Wen, Q. Guo, Q. Fu, X. Li, J. Xu, Y. Tang, Y. Zhao, X. Hu, Z. Du, L. Li, C. Wang, X. Zhou, Y. Chen, Babeltower: Learning to auto-parallelized program translation, in: Proceedings of the 39th International Conference on Machine Learning (ICML 2022), volume 162 of *Proc. Mach. Learn. Res.*, 2022, pp. 23685–23700.

[12] L. Chen, N. Xu, W. Chen, B. Lei, P.-H. Lin, D. Zhou, R. Thakur, C. Ding, A. Jannesari, C. Liao, Beyond code pairs: Dialogue-based data generation for llm code translation, arXiv preprint arXiv:2512.03086 (2025).

[13] J. Cordeiro, S. Noei, Y. Zou, Llm-driven code refactoring: Opportunities and limitations, in: Proceedings of the 2025 IEEE/ACM 2nd International Workshop on Integrated Development Environments (IDE), 2025, pp. 32–36.

[14] D. Beste, G. Menguy, H. Hajipour, M. Fritz, A. E. Cinà, S. Bardin, T. Holz, T. Eisenhofer, L. Schönherr, Exploring the potential of llms for code deobfuscation, in: Detection of Intrusions and Malware, and Vulnerability Assessment – DIMVA 2025, Proceedings, Part I, volume 15747 of *LNCS*, Springer, 2025, pp. 267–286.

[15] Y. Wang, X. Li, T. N. Nguyen, S. Wang, C. Ni, L. Ding, Natural is the best: Model-agnostic code simplification for pre-trained large language models, Proceedings of the ACM on Software Engineering 1 (2024) 586–608.

[16] OpenAI, gpt-oss-120b & gpt-oss-20b model card, https://cdn.openai.com/pdf/419b6906-9da6-406c-a19d-1bb078ac7637/oai_gpt-oss_model_card.pdf, 2025.

[17] Qwen Team, Qwen3-235b-a22b-instruct-2507 model card, https://huggingface.co/Qwen/Qwen3-235B-A22B-Instruct-2507, 2025.

[18] Y. Benjamini, Y. Hochberg, Controlling the false discovery rate: a practical and powerful approach

to multiple testing, Journal of the Royal Statistical Society: Series B (Methodological) 57 (1995) 289–300.

[19] T. J. Dekker, A floating-point technique for extending the available precision, Numerische Mathematik 18 (1971) 224–242.