KERNELBENCH: CAN LLMS WRITE EFFICIENT GPU KERNELS?

Anne Ouyang[1,*], Simon Guo[1,*], Simran Arora[1], Alex L. Zhang[2], William Hu[1], Christopher Ré[1], and Azalia Mirhoseini[1]

[1]Stanford University
[2]Princeton University

ABSTRACT

Efficient GPU kernels are crucial for building performant machine learning architectures, but writing them is a time-consuming challenge that requires significant expertise; therefore, we explore using language models (LMs) to automate kernel generation. We introduce **KernelBench**, an open-source framework for evaluating LMs' ability to write fast and correct kernels on a suite of 250 carefully selected PyTorch ML workloads. KernelBench represents a real-world engineering environment and making progress on the introduced benchmark directly translates to faster practical kernels. We introduce a new evaluation metric fast_p, which measures the percentage of generated kernels that are functionally correct and offer a speedup greater than an adjustable threshold p over baseline. Our experiments across various state-of-the-art models and test-time methods show that frontier reasoning models perform the best out of the box but still fall short overall, matching the PyTorch baseline in less than 20% of the cases. While we show that results can improve by leveraging execution and profiling feedback during iterative refinement, KernelBench remains a challenging benchmark, with its difficulty increasing as we raise speedup threshold p.

1 INTRODUCTION

AI relies on efficient GPU kernels to achieve high performance and cost and energy savings; however, developing kernels remains challenging. There has been a Cambrian explosion of ML architectures Tay et al. (2022); Peng et al. (2023); Dao & Gu (2024), but their available implementations routinely underperform their peak potential. We are seeing a proliferation of AI hardware NVIDIA (2017b; 2020; 2022); Jouppi et al. (2023); Groq; Cerebras; Graphcore, each with different specs and instruction sets, and porting algorithms across platforms is a pain point. A key example is the FlashAttention kernel Dao et al. (2022), which is crucial for running modern Transformer models — the initial kernel released in 2022, five years after the Transformer was proposed; it took two more years from the release of NVIDIA Hopper GPUs to transfer the algorithm to the new hardware platform. We explore the question: *Can language models help write correct and optimized kernels?*

AI engineers use a rich set of information when developing kernels and it is not clear whether language models (LMs) can mimic the workflow. They use compiler feedback, profiling metrics, hardware-specific specs and instruction sets, and knowledge of hardware-efficiency techniques (e.g., tiling, fusion, recompute). They can use programming tools ranging from assembly (e.g., PTX as in DeepSeek-AI (2025)) to higher-level libraries (ThunderKittens Spector et al. (2024), Triton Tillet et al. (2019)). Compared to existing LM code generation workloads Yang et al. (2024a), kernel writing requires a massive *amount* and *diversity* of information.

We first design an environment that reflects the typical AI engineer's workflow and supports providing LMs with this rich information. The environment should:

*indicates Equal Contribution. Correspondence Authors: aco@stanford.edu, simonguo@stanford.edu.

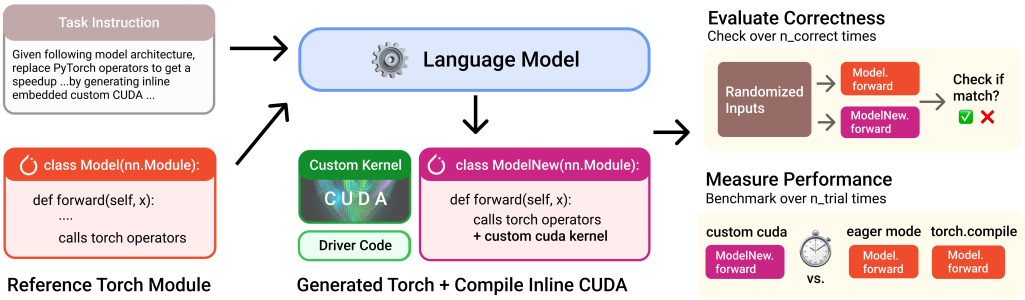


Figure 1: **KernelBench evaluates LMs' ability to generate performant GPU Kernels**. Overview of tasks in KernelBench: KernelBench tasks LMs with generating optimized CUDA kernels for a given target PyTorch model architecture and conducts automated evaluation

- **Automate** the AI engineer's workflow. The model should have full flexibility to decide which operators to optimize and how to optimize them.
- Support a **diverse** set of AI algorithms, programming languages, and hardware platforms.
- Make it **easy to evaluate** both performance and functional correctness of LM generations, ideally in a programmatic way. It should also capture profiling and execution information from generated kernels.

We introduce **KernelBench** to generate and evaluate kernels, which addresses the above considerations. KernelBench tests LM optimizations on three levels of AI workloads:

1. **Individual operations:** We include various AI operators, including matrix multiplies, convolutions, activations, norms, and losses. While PyTorch already uses expert-optimized closed-source kernels, making this a potentially challenging baseline, it is valuable if LMs can generate open-source kernels for the operations.
2. **Sequence of operations:** We provide problems that contain 3-6 individual operations together (e.g. a mainloop operator like matmul followed by pointwise operators like ReLU and Bias). This enables evaluating the models' ability to fuse multiple operators.
3. **End-to-end architectures:** We select architectures from popular AI repositories on Github including `pytorch`, `huggingface/transformers`, and `huggingface/pytorch-image-models`. These architectures contain many operations.

Mimicking an AI researcher's workflow, the LM takes PyTorch reference code as input and outputs an optimized version of the code. Similar to the human kernel development process, our environment enables the LM to iterate with compiler and profiler feedback to refine performance. The LM is free to use any programming language and decide both *which parts* of the PyTorch code to optimize, and *how* to optimize them. Our pipeline allows us to feed diverse information to the LMs, including hardware-specific information, example kernels, and compiler/profiler feedback.

We observe that frontier and open-source models perform poorly out-of-the-box on KernelBench, with OpenAI-o1 and DeepSeek-R1 matching the PyTorch Eager baseline on $< 20\%$ of the tasks. These model-generated kernels greatly suffer from execution errors, functional correctness issues, and are unable to perform platform-specific optimizations.

To identify areas for improvement, we conduct a series of experiments and analysis, and find that:

1. *Writing functionally **correct** kernels remains challenging for models:* while models are able to fix execution failures through either reasoning or multiple attempts, they struggle to produce functionally correct code. Furthermore, we observe a trade-off between LMs attempting more complex optimizations / niche hardware instructions (e.g., tensor core `wmma`) and producing error-free kernels. We hypothesize this is due to CUDA being a low-resource language in open-source training data, only 0.073% of popular code corpus The Stack v1.2 Li et al. (2023); Kocetkov et al. (2022).
2. *Models demonstrate potential to produce **performant** kernels via optimizations:* We observe a few instances where LMs make algorithmic improvements – e.g., exploiting sparsity, operator fusion,

and utilizing hardware features. We notice more of such instances when we explicitly condition the LM on hardware information (e.g., bandwidth and TFLOP specs) and demonstrations of hardware optimization techniques (e.g., tiling, fusion). While these capabilities remain nascent, LMs do **demonstrate potential** for generating performant kernels.

3. *Leveraging **feedback** is important for reducing execution errors and discovering faster solutions:* By providing execution results and profiler feedback to the LM in context, the kernel quality significantly improves after multiple refinements from 12%, 36%, and 12% in fast_1 to 43%, 72%, and 18% respectively.

Our findings highlight the technical challenges we need to solve in order to adopt LMs for kernel writing. These include but are not limited to: how to improve LM performance in a low-resource data regime, and how to select from the rich set of information we can provide to models. To address these challenges, we contribute (1) **an open-source framework** to study LM kernel generation with a comprehensive suite of evaluation problems and (2) **analysis of where current LMs stand** and how to realize a future of efficient kernels generated by models.

2 Related Works

Kernel libraries and compilers. We evaluate existing approaches for kernel programming along the dimensions of automation, breadth, and performance. Mainstream kernel programming libraries like cuDNN NVIDIA (2014), CUTLASS NVIDIA (2017a), and Apple MLX Apple (2020) are hardware-specific and demand substantial engineering effort from human experts. Other libraries, like ThunderKittens Spector et al. (2024) and Triton Tillet et al. (2019), successfully help AI researchers write a breadth of fast and correct kernels Arora et al. (2024); Yang & Zhang (2024), but still require human programming effort. Compiler-based tools, like torch.compile Paszke et al. (2019) and FlexAttention Team PyTorch et al. (2024), automatically provide a narrow slice of optimizations. In contrast to these efforts, we ask if LMs can automatically generate performant kernels for a breadth of AI workloads.

LLMs for performance-optimized code generation. In the past year, there have been several efforts to build LMs that can automate algorithmic coding Chen et al. (2021); Shi et al. (2024); Li et al. (2022), resolving GitHub issues Yang et al. (2024a;b), and domain-specific coding Yin et al. (2022); Lai et al. (2022). While these works focus on producing correct and functional code, subsequent works have explored LMs' ability to produce solutions with better *algorithmic and asymptotic efficiency* Nichols et al. (2024); Waghjale et al. (2024). KernelBench focuses on *wall-clock efficiency*. LMs generate high-performance computing (HPC) code, which requires an understanding of the underlying hardware features and device instruction set, and common performance characteristics of parallel processors.

Existing works in the space of HPC code generation have evaluated LM performance on translating arbitrary code samples from C++ to CUDA TehraniJamsaz et al. (2024); Wen et al. (2022) or generating well-known, low-level kernels such as GEMMs Valero-Lara et al. (2023); Wijk et al. (2024). KernelBench instead curates a set of 250 diverse kernels from real-world, modern deep learning workloads, many of which do not have existing human-written implementations — in other words, solving KernelBench tasks are immediately beneficial for real deep learning workloads.

3 KernelBench: A Framework for AI Kernel Generation

KernelBench is a new framework for evaluating the ability of language models to generate performant kernels for a breadth of AI workloads.

3.1 KernelBench Task Format

KernelBench contains 250 tasks representing a range of AI workloads, and is easily extensible to new workloads. The end-to-end specification for a task is illustrated in Figure 1 and described below.

Task input: Given an AI workload, the input to the task is a reference implementation written in PyTorch. Mimicking an AI researcher's workflow, the PyTorch code contains a class named `Model` derived from `torch.nn.Module()`, where the standard `__init__` and `forward()` functions (and any helper functions) are populated with the AI workload's PyTorch operations.

AI algorithms generally operate on large tensors of data. The optimal kernel for a workload depends on the size and data type (e.g., BF16, FP8) of the tensor. Therefore, each task additionally contains functions `get_inputs()` and `get_init_inputs()`, which specify the input tensors exactly.

Task output: Given the input, the LM needs to output a new class named `ModelNew` derived from `torch.nn.Module()`, which contains custom optimizations. For example, the LM can incorporate in-line kernel calls during the `forward()` function using CUDA-C extension in PyTorch.

In order to succeed, the LM needs to identify (1) which operations in the `Model` class would benefit most from optimizations, and (2) how to optimize those operations. The LM can use any hardware-efficiency techniques such as fusion and tiling or specialized instructions (e.g., tensor cores) and any programming library (e.g., PTX, CUDA, CUTLASS, Triton, ThunderKittens).

3.2 Task Selection

The 250 tasks in KernelBench are partitioned into three levels, based on the number of primitive operations, or PyTorch library functions, they contain:

- **Level 1 (100 tasks): Single primitive operation.** This level includes the foundational building blocks of AI (e.g. convolutions, matrix-vector and matrix-matrix multiplications, losses, activations, and layer normalizations). Since PyTorch makes calls to several well-optimized and often closed-source kernels under-the-hood, it can be challenging for LMs to outperform the baseline for these primitive operations. However, if an LM succeeds, the open-source kernels could be an impactful alternative to the closed-source (e.g., CuBLAS NVIDIA (2023)) kernels.
- **Level 2 (100 tasks): Operator sequences.** This level includes AI workloads containing multiple primitive operations, which can be fused into a single kernel for improved performance (e.g., a combination of a convolution, ReLU, and bias). Since compiler-based tools such as the PyTorch compiler are effective at fusion, it can be challenging for LMs to outperform them. However, LMs may propose more complex algorithms compared to compiler rules.
- **Level 3 (50 tasks): Full ML architectures.** This level includes architectures that power popular AI models, such as AlexNet and MiniGPT, collected from popular PyTorch repositories on GitHub. Given the scale of modern models, it is critical to use kernels when running training and inference. Unfortunately, it has been difficult for the AI community to generate performant kernels. For instance, it took 5 years from the release of the Transformer architecture Vaswani et al. (2017) to obtain performant kernels Dao et al. (2022), let alone today's many new architectures. Peak performance kernels for these architectures require algorithmic modifications that are often beyond the scope of a compiler.

3.3 Metric Design

Evaluation approach KernelBench is an evaluation-only benchmark. We do not provide ground truth kernels for the tasks since we imagine users benchmarking on a variety of hardware platforms (including new platforms), input types, and workloads. However, by design, KernelBench is *automatically verifiable*. Given a task, we randomly generate input tensors of the prescribed shape and precision and collect the PyTorch `Model` output. We can evaluate whether LM generations are correct and fast as follows:

1. **Correctness** We compare the `Model` output to the LM-generated `ModelNew` output. We evaluate on 5 random inputs per problem (detailed in Appendix C).
2. **Performance** We compare the wall-clock execution time of `Model` against `ModelNew` using repeated trials to account for timing variations.

Comparing LMs on KernelBench Some LMs may generate a small number of correct kernels that are very fast, while other LMs generate a large number of correct kernels that are quite slow. To capture both axes of correctness and performance, we introduce a new metric called fast_p, which is defined as the fraction of tasks that are both correct and have a speedup (computed as the ratio of PyTorch wall-clock time to generated kernel time) greater than threshold p. Formally:

$$\text{fast}_p = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(\text{correct}_i \wedge \{\text{speedup}_i > p\}),$$

where fast_0 is equivalent to the LM's correctness rate, as it measures the fraction of tasks for which the LM code is functionally correct regardless of its speed.

By adjusting the threshold parameter p, we enable evaluation of kernel performance at different speedup thresholds and capture the speedup distributions. For our evaluations, we focus on $p = 1$ as a starting point, with the possibility of increasing p as future methods for kernel generation improve. Additionally, using $p < 1$ for training is valuable, since PyTorch relies on complex optimized kernels, and matching even a fraction of their performance is still considered beneficial.

4 KERNELBENCH BASELINE EVALUATION

In this section, we investigate how a range of LMs perform when evaluated off-the-shelf on KernelBench and explore their capabilities and failure modes.

4.1 ONE-SHOT BASELINE

We evaluate LMs using a prompt that contains one example of a PyTorch `Model` input and `ModelNew` output, highlighting the task format. The example is simple, containing only an `add` operator (See Appendix D.1). Given this in-context example and the PyTorch task `Model` to optimize, the LM generates `ModelNew` via greedy decoding. We profile the generated code on an NVIDIA L40S GPU, and measure the fast_p metric across all problems. Table 1 shows that the LM-generated kernels achieve a speedup over PyTorch Eager in fewer than 20% of tasks on average.

fast$_1$ over:	PyTorch Eager			torch.compile		
KernelBench Level	1	2	3	1	2	3
GPT-4o	4%	5%	0%	18%	4%	**4%**
OpenAI o1	10%	24%	**12%**	28%	19%	**4%**
DeepSeek V3	6%	4%	8%	20%	2%	2%
DeepSeek R1	**12%**	**36%**	2%	**38%**	**37%**	2%
Claude 3.5 Sonnet	10%	7%	2%	29%	2%	2%
Llama 3.1-70B Inst.	3%	0%	0%	11%	0%	0%
Llama 3.1-405B Inst.	3%	0%	2%	16%	0%	0%

Table 1: **KernelBench is a challenging benchmark for current LMs**. Here we present fast_1, i.e. the percentage of problems where the model-generated kernel is faster than the PyTorch Eager and `torch.compile` baseline (default configuration) on NVIDIA L40S.

4.2 CORRECTNESS: ERROR ANALYSIS

In Figure 2, we analyze the failure modes of LMs across problems. It can be seen that a large proportion of model-generated kernels are incorrect. To better understand where model-generated kernels fail, we break down their correctness issues into execution failures (CUDA/`nvcc` / Python compile-time errors, CUDA memory violations, and runtime errors) and correctness errors (output tensor shape and value mismatches). We observe that the reasoning LMs (o1, R1) produce fewer incorrect solutions ($< 55\%$) than other models ($> 70\%$). However, we find this is mainly because they make fewer execution failures. All LMs struggle with functional correctness to a similar degree.

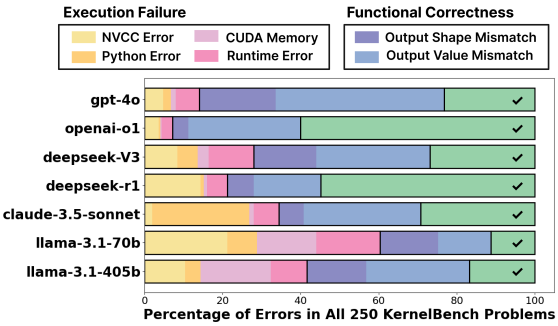


Figure 2: **We categorize failure modes of kernel code into execution failure and functional correctness**. For the one-shot baseline, reasoning models generate fewer kernels with execution failures, but all models struggle similarly with functional correctness.

4.3 PERFORMANCE: SPEEDUP DISTRIBUTION

A key point of interest is whether the functionally correct LM-generated kernels outperform the PyTorch baseline. Figure 3 shows the distribution of fast_p as p varies, indicating the percentage of

The `torch.compile` baseline runtime is sometimes slower than Torch Eager – this is due to reproducible runtime overhead (*not compile time*) that could be significant for small kernels in Level 1. We focus on PyTorch Eager for the rest of our analysis, but we elaborate on other baselines in Appendix C.

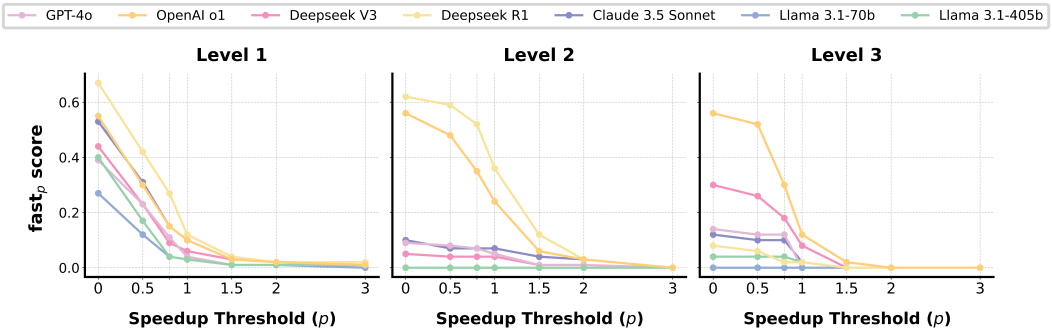


Figure 3: **Most LM-generated kernels are slow.** This figure shows the distribution of the fast$_p$ metric as the speedup threshold p (over PyTorch baseline) increases. fast$_0$ represents the number of *correct* kernels regardless of speed, and fast$_1$ represents the number of correct kernels achieving at least $> 1\times$ speedup over PyTorch. Increasing the threshold p increases the difficulty.

kernels that are p-times faster than the PyTorch Eager baseline (the top right of the plot is better). At $p = 1$, fewer than 15% of LM-generated kernels outperform PyTorch across all KernelBench levels. Reasoning-based LMs generally outperform the other LMs in providing speedups.

4.4 Performance Variations across Hardware

Our one-shot baseline makes no assumptions about the underlying hardware, so a natural question is how our analysis of the LM-generated kernels generalizes across various GPU types. Table 14 and Figure 7 show that kernels outperforming PyTorch Eager on NVIDIA L40S in Level 1 achieve similar speedups versus the baselines on other GPUs. However, on problems in Level 2, LMs exhibit larger variations in speedups across GPUs (Figure 8): DeepSeek R1-generated kernels achieve a fast$_1$ of 36% on NVIDIA L40S but 47% on NVIDIA A10G for Level 2. This suggests that one-shot LM-generated kernels may not generalize well across hardware. To generate target-specific kernels, we explore in Section 5.2 whether providing hardware-specific details in-context could help.

Our analysis reveals that the best models available today struggle to generate correct kernels that outperform the baseline PyTorch speeds. LM-generated kernels frequently fail due to simple compiler and run-time errors. Furthermore, it is difficult for LMs to write kernels that perform well across hardware platforms given simple instructions.

5 Analysis of Model Capabilities

In the last section, we found that KernelBench is a challenging benchmark for today's models. In this section, we conduct case studies to explore opportunities for improvement in future AI systems.

5.1 Case Study: Leveraging KernelBench Environment at Test-Time

As observed in Section 4.2, execution failures are the most frequent failure mode in LM-generated kernels. The environment provided by KernelBench allows us to collect rich signals, including compiler errors, correctness checks, and runtime profiling metrics, all of which can be fed back in to the LM to help it resolve kernel failures. To explore how well LMs can use this feedback, we evaluate and compare two baselines: (1) generating multiple parallel samples from the LM per KernelBench task and (2) sequentially generating kernels per KernelBench task by allowing the LM to iteratively refine using the execution feedback.

5.1.1 Repeated Sampling

The KernelBench environment enables programmatic verification of LM-generated kernels, allowing us to collect and evaluate multiple LM generations *per task* Brown et al. (2024); Li et al. (2022); Grubisic et al. (2024). We evaluate this *repeated sampling* approach using fast$_p$@k, which measures the percentage of tasks where the model generated **at least one functionally correct kernel that is p times faster than PyTorch Eager when drawing k samples**.

Repeated sampling helps LMs discover more fast and correct solutions. Figure 4 shows that repeated sampling with high temperature improves fast_1 as k increases across all three levels with both DeepSeek-V3 and Llama 3.1 70B. Notably, on Level 2, DeepSeek-V3 reaches a fast_1 of 37% with $k = 100$ samples, compared to just 4% in the one-shot baseline. Examining the samples, we find that high-temperature sampling helps explore the solution space, increasing the chances of generating error-free kernels with better optimizations. However, if a model has a very low inherent probability of solving a task, simply increasing the sampling budget has limited impact. For example, DeepSeek-V3 was never able to generate any correct solution for a group of 34 convolution variants in Level 1, even when attempting with 100 samples.

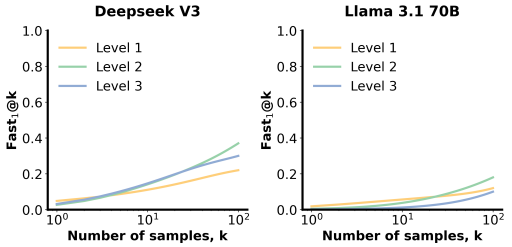
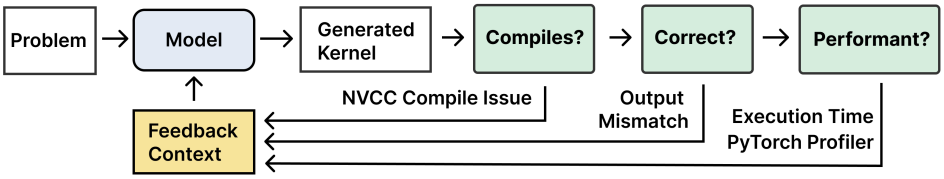


Figure 4: **Repeated sampling helps discover more correct and performant kernels**. As the number of parallel samples k increases (up to 100), fast_1@k improves for both DeepSeek-V3 and Llama 3.1-70B Instruct across all 3 KernelBench levels.

5.1.2 Iterative Refinement of Generations

The KernelBench environment is well-suited for collecting compiler feedback, execution errors, and timing analysis using tools like the PyTorch profiler as ground-truth signals. We investigate whether leveraging this feedback can help LMs to iteratively refine their generations.



We provide feedback to the model after each generation in a multi-turn process: after the initial generation, we provide the model with its previous generation G, as well as compiler/execution feedback E and/or profiler output P over its current generation. We define each generation and subsequent feedback as a *turn*, and run this **Iterative Refinement** process over N turns. For each turn, we measure $\text{fast}_p@N$, which is the percentage of tasks where the model generated **at least one** functionally correct kernel that is p times faster than PyTorch Eager by turn N.

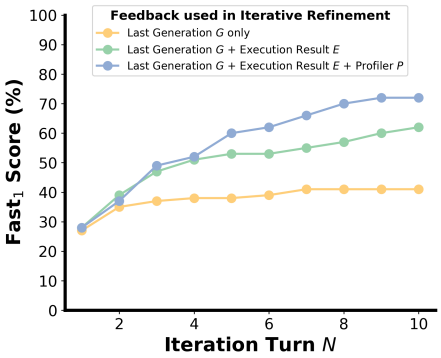


Figure 5: **Iterative refinement with execution feedback E and profiling information P enable models to improve kernel generations over turns**, as shown in the $\text{fast}_1@N$ trajectory of DeepSeek-R1 on Level 2. The percentage of problems where the best generated kernel up to turn N is correct and faster than PyTorch Eager consistently increases as we increase the number of turns.

Leveraging execution feedback helps reduce errors and improves overall speedups over time. We examine the fast_1 behavior at turn $N = 10$ in Table 2 and find that iterative refinement consistently improves performance across models and levels of KernelBench. DeepSeek-R1 on Level 2 results in the most notable improvement, where the combination of execution feedback E and profiler feedback P boosts fast_1 from 36% to 72% (shown in Figure 5).

Furthermore, by examining iterative refinement trajectories, we find that models self-correct more effectively with execution feedback E, fixing issues especially related to execution errors. DeepSeek-R1 on Level 1 and 2 can generate a functional kernel on >90% of the tasks within 10 turns of refinement (Table 9). However, the remaining incorrect kernels almost always fail due to functional

incorrectness, likely because correctness feedback is less granular than execution failure messages. We include successful and failed examples of iterative refinement trajectories in Appendix E.4.

5.1.3 COMPARING REPEATED SAMPLING AND ITERATIVE REFINEMENT

Method	**Level 1**			**Level 2**			**Level 3**		
	Llama-3.1 70B	DeepSeek V3	Deepseek R1	Llama-3.1 70B	Deepseek V3	Deepseek R1	Llama-3.1 70B	Deepseek V3	Deepseek R1
Single Attempt (Baseline)	3%	6%	12%	0%	4%	36%	0%	8%	2%
Repeated Sampling (@10)	5%	11%	N/A	3%	**14%**	N/A	1%	14%	N/A
Iterative Refinement w G	**9%**	9%	18%	0%	7%	44%	0%	14%	4%
Iterative Refinement w G+E	5%	13%	41%	**5%**	5%	62%	**8%**	**22%**	12%
Iterative Refinement w G+E+P	7%	**19%**	**43%**	4%	6%	**72%**	2%	14%	**18%**

Table 2: **Both repeated sampling and iterative improvement enable models to generate more correct and fast kernels compared to baseline:** Here we present the percentage of problems where the LM-generated kernel is correct and faster than baseline Torch Eager (Fast_1 in %) for the two test-time methods, both with the same sample budget of 10 calls. We further compare performance within iterative refinement achieved when leveraging previous Generation G, Execution Result E, and Timing Profiles P. Note we do not repeatedly sample DeepSeek R1, as its API endpoint does not provide a temperature parameter.

In Table 2, we compare repeated sampling and iterative refinement given a fixed budget of 10 inference calls. Both methods provide meaningful improvements over the one-shot baseline, with iterative refinement being more effective in 5 of the 6 cases. However, ultimately we find that the effectiveness of the test-time methods is inherently dependent on the quality of the base model. For instance, with repeated sampling, DeepSeek-V3 consistently outperforms Llama-3.1 70B across all three levels. Similarly, with iterative refinement, DeepSeek-R1 consistently improves using feedback E and P, while DeepSeek-V3 and Llama-3.1 70B does not always benefit from having such information.

5.2 CASE STUDY: GENERATING HARDWARE-EFFICIENT KERNELS VIA HARDWARE KNOWLEDGE

It is clear that LMs demonstrate limited success at generating hardware-efficient kernels. This is likely due to the scarcity of kernel code in the training data and the fact that the optimal kernel may need to change depending on the hardware platform-specific properties, as discussed in Section 4.4. In this case study, we explore providing 1) in-context examples of best-practices for kernel engineering and 2) in-context hardware specification details.

5.2.1 HARDWARE-AWARE IN-CONTEXT EXAMPLES

Well-written kernels often use techniques such as fusion, tiling, recompute, and asynchrony to maximize performance. We find that most of the one-shot generated kernels evaluated in Section 4 often do not use these techniques. Here, we explore whether providing explicit in-context examples that use these techniques can help the LMs improve their performance on KernelBench. Specifically, we include three in-context examples: GeLU Hendrycks & Gimpel (2023) using operator fusion, matrix multiplication using tiling Mills (2024), and a minimal Flash-Attention Dao et al. (2022); Kim (2024) kernel that demonstrates shared memory I/O management.

In-context examples degrade the LM's *overall* fast_1 score since LMs attempt more aggressive optimization strategies, but result in more execution failures. OpenAI o1's generations are 25% longer on average using the few-shot examples, compared to the generations produced by Section 4 baseline. However, among the correct solutions, the LMs apply interesting optimizations: we find that on 77% of GEMM variants in KernelBench Level 1, o1 applies tiling and improves speed over the one-shot baseline (although remains slower than PyTorch Eager due to the lack of tensor core utilization). On Level 2, o1 applies aggressive shared memory I/O management on 11 problems, and is able to outperform PyTorch Eager (See Appendix G).

5.2.2 SPECIFYING HARDWARE INFORMATION

As discussed in Section 4.4, kernel performance varies depending on the hardware platform. For instance, FlashAttention-2 Dao (2024) degrades 47% in hardware utilization going from the NVIDIA A100 to H100 GPU. FlashAttention-3 Shah et al. (2024), an entirely different algorithm, was written for the H100. In this study, we explore whether LMs can use (1) hardware specifications such as the GPU type (H100, A100, etc.), memory sizes, bandwidths, TFLOPS and (2) hardware knowledge (e.g. definitions of threads, warps, thread-blocks, streaming multiprocessors) in-context to generate improved kernels (See Appendix H for more detail on the context).

Models rarely generate kernels that are optimized for the underlying hardware, highlighting room for improvement for future models. Certain generations of GPUs (e.g. H100) feature a variety of new hardware units and instructions from their predecessors. Providing hardware information does not significantly impact the outputs of Llama 3.1 70B or DeepSeek-V3. Interestingly, we find that a subset of OpenAI o1 and DeepSeek-R1 generated kernels use hardware-specific instructions and optimizations. R1 attempts to generate warp matrix multiply-accumulate (`wmma`) instructions (Figure 9) for approximately 50% of the Level 1 matrix multiplication problems, although most fail to compile. Among the functionally correct generations, R1 and o1 produce 1-3 outliers per level that are $\geq 2\times$ faster than the Section 4 baselines. Overall, we find that LMs are better at adjusting their approaches when provided with few-shot examples in Section 5.2.1 than with hardware information.

6 DISCUSSION

6.1 DEEP DIVE INTO INTERESTING KERNELS

Here, we discuss a few surprising LM-generated kernels that demonstrate significant speedups over the PyTorch baseline. See detailed examples in Appendix E.

Operator fusion GPUs have small amounts of fast-access memory and large amounts of slow-access memory. Fusion can help reduce slow-access I/O costs by performing multiple operations on data that has been loaded into fast-access memory. We find that LMs optimize the GELU (2.9x) and Softsign (1.3x) operators by fusing their computations into a single kernel. LMs generated a kernel that fuses multiple foundational operators – matrix multiplication with division, summation, and scaling – giving a 2.6x speedup. Overall, LMs leave many fusion opportunities on the table.

Memory hierarchy Effective kernels explicitly manage utilization of the limited amounts of shared and register memory. In the generated kernels, we found kernels that uses GPU shared memory – cosine similarity (2.8x) and triplet margin loss (2.0x) – to achieve speedups. We did not find successful usages of tensor core instructions, which are crucial for AI performance.

Algorithmic optimizations Kernels can require algorithmic modifications to better utilize the hardware features. We found one interesting generation for the problem of performing a multiplication between a dense and diagonal matrix, where the kernel scales each row (or column), rather than loading the zero-entries of the diagonal matrix, yielding a 13x speedup over PyTorch Eager.

6.2 CONCLUSION

Our contributions are: (1) We present KernelBench, a framework that lays the groundwork for LM-driven kernel optimization, and (2) We evaluate a diverse set of models and approaches, analyzing their strengths and limitations, and providing insights into opportunities for improvement.

Overall, while most benchmarks eventually saturate, KernelBench is designed to dynamically evolve as new AI workloads arise. Our fast_p metric can be adapted over time to measure the speedup threshold (p) over increasingly advanced baselines (i.e., beyond the PyTorch baseline used in our work). Since PyTorch is cross-hardware platform compatible, the PyTorch-based tasks in KernelBench tasks can be evaluated on every *new hardware platform* release. Finally, unlike many benchmarks, success on KernelBench directly maps to production value and real-world impacts (lowering costs and reducing energy consumption at scale). These properties ensure that KernelBench will remain valuable in the ever-evolving AI landscape.

ETHICS STATEMENT

Optimized GPU kernels can lead to significant energy savings in large-scale machine learning workloads, reducing both computational costs and environmental impact. By providing a framework for AI-assisted performance tuning, KernelBench contributes to more energy-efficient AI systems, aligning with global efforts to reduce the carbon footprint of computing infrastructure.

KernelBench does not involve human studies or collect user data, eliminating privacy concerns. It also avoids proprietary or private code, relying solely on publicly available Github repositories.

ACKNOWLEDGEMENTS

We are grateful to Google DeepMind, Google, IBM, Stanford HAI, PyTorch, PrimeIntellect, and Modal for supporting this work. We thank Aaryan Singhal, AJ Root, Allen Nie, Anjiang Wei, Benjamin Spector, Bilal Khan, Bradley Brown, Dylan Patel, Genghan Zhang, Hieu Pham, Hugh Leather, John Yang, Jon Saad-Falcon, Jordan Juravsky, Marcel Rød, Mark Saroufim, Michael Zhang, Minkai Xu, Ryan Ehrlich, Sahan Paliskara, Sahil Jain, Shicheng (George) Liu, Suhas Kotha, Tatsunori Hashimoto, Vikram Sharma Mailthody, and Yangjun Ruan for insightful discussions and constructive feedback in shaping this work.

REFERENCES

Apple. Apple ml compute framework (mlx), 2020. URL https://developer.apple.com/metal/.

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher Ré. Simple linear attention language models balance the recall-throughput tradeoff. *International Conference on Machine Learning*, 2024.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Cerebras. Cerebras wafer-scale engine wse architecture. Online. https://cerebras.ai/product-chip/.

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, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. *International Conference on Learning Representations*, 2024.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *International Conference on Machine Learning (ICML)*, 2024.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

DeepSeek-AI. Deepseek-v3 technical report, 2025. URL https://github.com/deepseek-ai/DeepSeek-V3.

Graphcore. Graphcore IPU architecture. Online. https://www.graphcore.ai/products/ipu.

Groq. Groq architecture. Online. https://groq.com/.

Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. Priority sampling of large language models for compilers, 2024. URL https://arxiv.org/abs/2402.18734.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023. URL https://arxiv.org/abs/1606.08415.

Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings, 2023. URL https://arxiv.org/abs/2304.01433.

Peter Kim. Flashattention minimal. Online, 2024. https://github.com/tspeterkim/flash-attention-minimal.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code, 2022. URL https://arxiv.org/abs/2211.15533.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation, 2022. URL https://arxiv.org/abs/2211.11501.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023. URL https://arxiv.org/abs/2305.06161.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Christian J. Mills. Cuda mode notes - lecture 004. Online, 2024. https://christianjmills.com/posts/cuda-mode-notes/lecture-004/.

Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, Todd Gamblin, and Abhinav Bhatele. Performance-aligned llms for generating fast code, 2024. URL https://arxiv.org/abs/2404.18864.

NVIDIA. cudnn: Gpu-accelerated library for deep neural networks, 2014. URL https://developer.nvidia.com/cudnn.

NVIDIA. Cuda templates for linear algebra subroutines, 2017a. URL https://github.com/NVIDIA/cutlass.

NVIDIA. Nvidia Tesla V100 GPU architecture, 2017b.

NVIDIA. Nvidia A100 tensor core GPU architecture, 2020.

NVIDIA. Nvidia H100 tensor core GPU architecture, 2022.

NVIDIA. cuBLAS, 2023. URL https://docs.nvidia.com/cuda/cublas/.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, and Jiaming et al. Kong. Rwkv: Reinventing rnns for the transformer era. *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. Can language models solve olympiad programming?, 2024. URL https://arxiv.org/abs/2404.10952.

Benjamin Spector, Simran Arora, Aaryan Singhal, Daniel Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. 2024.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022.

Team PyTorch, Horace He, Driss Guessous, Yanbo Liang, and Joy Dong. FlexAttention: The flexibility of PyTorch with the performance of FlashAttention, 2024. URL https://pytorch.org/blog/flexattention/.

Ali TehraniJamsaz, Arijit Bhattacharjee, Le Chen, Nesreen K. Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=V6hrg4O9gg.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. URL http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf.

Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F. Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter. Comparing llama-2 and gpt-3 llms for hpc kernels generation, 2023. URL https://arxiv.org/abs/2309.07103.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *31st Conference on Neural Information Processing Systems (NIPS 2017)*, 2017.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15362–15376, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.859. URL https://aclanthology.org/2024.emnlp-main.859/.

Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. BabelTower: Learning to auto-parallelized program translation. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 23685–23700. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/wen22b.html.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Josh Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. Re-bench: Evaluating frontier ai rd capabilities of language model agents against human experts, 2024. URL https://arxiv.org/abs/2411.15114.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv:2405.15793*, 2024a.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024b. URL https://arxiv.org/abs/2410.03859.

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL https://github.com/sustcsonglin/flash-linear-attention.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks, 2022. URL https://arxiv.org/abs/2212.09248.

A KERNELBENCH TASK EXAMPLE

Here we provide an example task from KernelBench. Each task is wrapped in a class named `Model`. A task contains two key functions in the `Model` class, `__init__` and `forward`; helper functions are included if necessary. We fix the shape of inputs and vary the numerical values through randomly generated tensors. We provide two functions, `get_inputs` and `get_init_inputs`, for generating random parameters for initializing the model and running a forward pass, respectively.

PyTorch Reference Architecture:

```
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Simple model that performs a single matrix multiplication (C = A * B)
        ↪  with a large K dimension
    """
    def __init__(self):
        super(Model, self).__init__()

    def forward(self, A: torch.Tensor, B: torch.Tensor) -> torch.Tensor:
        """
        Performs matrix multiplication of A and B.

        Args:
            A: Input tensor of shape (M, K)
            B: Input tensor of shape (K, N)

        Returns:
            Output tensor of shape (M, N)
        """
        return torch.matmul(A, B)

M = 256
N = 256
K = 131072

def get_inputs():
    A = torch.randn(M, K)
    B = torch.randn(K, N)
    return [A, B]

def get_init_inputs():
    return []  # No special initialization inputs needed
```

We include an example output from a model below that attempts to optimize the task specification above. The model is not only responsible for generating the kernel code, but is also responsible for generating the peripheral code to integrate the kernel into the PyTorch framework. The evaluation framework will run the forward pass of the model as if it is a PyTorch operator, so a common approach is to inline the CUDA code.

```python
import torch
import torch.nn as nn
from torch.utils.cpp_extension import load_inline

# Define the custom CUDA kernel for matrix multiplication
matmul_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void matmul_kernel(const float* A, const float* B, float* C, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < K; ++k) {
            sum += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}

torch::Tensor matmul_cuda(torch::Tensor A, torch::Tensor B) {
    int M = A.size(0);
    int N = B.size(1);
    int K = A.size(1);

    auto C = torch::zeros({M, N}, A.options());

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (M + threadsPerBlock.y - 1) / threadsPerBlock.y);

    matmul_kernel<<<numBlocks, threadsPerBlock>>>(A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), M, N, K);

    return C;
}
"""

matmul_cpp_source = (
    "torch::Tensor matmul_cuda(torch::Tensor A, torch::Tensor B);"
)

# Compile the inline CUDA code for matrix multiplication
matmul = load_inline(
    name="matmul",
    cpp_sources=matmul_cpp_source,
    cuda_sources=matmul_source,
    functions=["matmul_cuda"],
    verbose=True,
    extra_cflags=[""],
    extra_ldflags=[""],
)


class ModelNew(nn.Module):
    def __init__(self):
        super(ModelNew, self).__init__()
        self.matmul = matmul

    def forward(self, A: torch.Tensor, B: torch.Tensor) -> torch.Tensor:
        return self.matmul.matmul_cuda(A, B)
```

B OPPORTUNITIES FOR FUTURE WORK

We show that there is significant room for improvement on KernelBench given the currently available models. First, future work can explore the development of advanced fine-tuning and reasoning techniques, including agentic workflows. Since CUDA is a low-resource language, it would be valuable for future work to open-source more high quality data. Second, LMs generate raw CUDA code in our experiments. However, future work can explore whether generating code using alternative programming abstractions (e.g., provided in ThunderKittens, CUTLASS, Triton, and others) can simplify the generation problem, for instance by making it easier for LMs to leverage tensor core instructions. Third, our evaluation has also been limited to GPUs so far and future work can expand to other hardware accelerators.

C EVALUATION METHODOLOGY AND BASELINES

All evaluations are conducted on a bare-metal NVIDIA L40S GPU with Ada Lovelace architecture unless otherwise stated (such as the device generalization experiments in Section 4.4 and the hardware case study in 5.2). The NVIDIA L40S has 48 GB of HBM memory and operates at 300W. Our environment uses Python `3.10`, PyTorch `2.5.0+cu124`, and CUDA `12.4`, which is also where our PyTorch Eager and `torch.compile` baselines are derived from.

C.1 KERNEL EVALUATION SETUP

Recall the KernelBench task entails a PyTorch reference module `Model` as baseline, and model-generated PyTorch architecture `ModelNew` with custom inline CUDA kernel.

For **correctness**, we set *num_correctness* to 5, where we check equivalence of output between reference architecture `Model` and generated architecture with custom kernel `ModelNew` with 5 randomized inputs. We elaborate on our choice in C.2.

For **performance**, we measure the wall-clock execution time of `nn.module.forward` for both `Model` and `ModelNew`. We ensure only one kernel is being evaluated (no other CUDA process) on current GPU. We warm up for 3 iterations and then set *num_profile* to 100 times which measures the elapsed execution time signaled between CUDA events `torch.cuda.Event`. We take the mean of the 100 trials, and also note its max, min, and standard deviation. While the wall clock time might vary for every trial, we note our coefficient of variation (CV): std/mean is consistently $< 3\%$, we use the mean of both measured wall clock time for comparisons.

To compute the speedup of generated architecture over baseline architecture for individual problems, we use the mean for both speedup $= T_{Model}/T_{ModelNew}$. For example, if $T_{Model} = 2$ ms and $T_{ModelNew} = 1$ ms, we have a 2x speedup with the newly generated kernel. We compare this speedup with our speedup threshold parameter p (as explained in section 3.3) to compute fast_p scores.

C.2 CORRECTNESS ANALYSIS VARYING NUMBER OF RANDOMLY GENERATED INPUTS

Checking equivalence of programs in a formal sense is undecidable. "The Halting Problem" (Turing, 1936) states that it is impossible to decide, in general, whether a given program will terminate for every possible input. This problem naturally extends to checking equivalence because in order to check whether two programs are equivalent, it is necessary to check their behavior for all inputs, including cases where one or both programs may not terminate. Since determining whether a program halts on a given input is undecidable (the Halting Problem), checking equivalence also becomes undecidable.

Approximate or heuristic methods are often used in practice for checking program equivalence. Random testing is the most common practical approach, where the program is run with sets of randomly chosen inputs, and their outputs are compared. Random testing is particularly effective for AI kernels, where control flow is simpler and the focus is primarily on numerical correctness. By using diverse inputs, it can uncover errors in computations or memory handling with high probability.

We use five sets of random inputs for correctness, which is a good tradeoff between the ability to catch errors and efficiency. In an experiment with 100 generated kernels, the results were as follows:

50 kernels were correct (all 5/5 and 100/100), 19 had output value mismatches (19 0/5 and 0/100), 4 had output shape mismatches, 10 encountered runtime errors, and 17 had compilation errors. Notably, the 0/5 and 0/100 failures indicate that no partial correctness was observed.

C.3 Distribution of Model Performance for One-Shot Baseline

Here we examine the quality of (functionally correct) kernel generations across a wide variety of models. Figure 6 shows the distribution of speedups for various kernels across different levels and models. The median speedup for both Level 1 and Level 3 are less than 1, and the median speedup for Level 2 is only slightly above one. Level 1 has the most significant outliers, in one case showing a speedup greater than 10. We explored some of these outlier cases in greater detail in Section 6.

Reasoning-optimized models (OpenAI-o1 and DeepSeek-R1) perform the best of out-of-the-box across all levels. These models demonstrate superior kernel generation capabilities, particularly excelling at Level 2 tasks (which mainly involve kernel fusion). In contrast, Llama 3.1 models (both 405B and 70B) perform poorly regardless of model size, suggesting that larger models do not necessarily guarantee better results for this task. DeepSeek-R1, while strong at Level 1 and 2, suffers significantly at Level 3, often generating incorrect kernels.

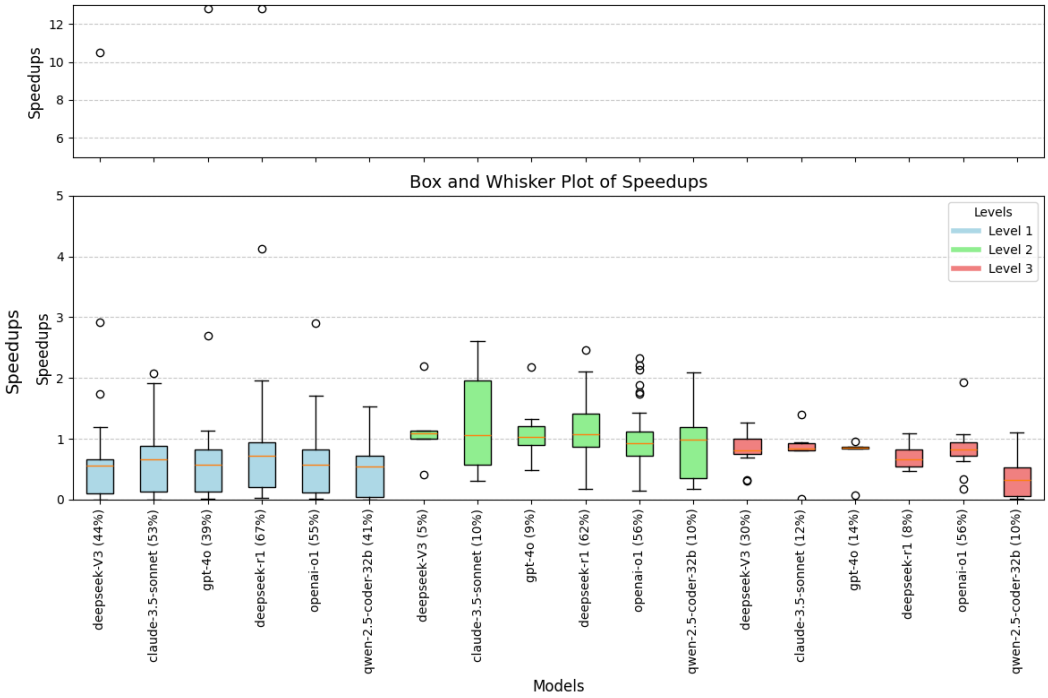


Figure 6: A box and whisker plot of the speedup relative to Torch Eager of (correct) kernels generated by various models in the one-shot baseline setting. We also write the percentage of correctly generated kernels next to the model name. We observe that among most models, the median speedup for correctly generated kernels is below 1.

C.4 PyTorch Baselines

PyTorch offers two common execution modes: Eager and `torch.compile`. Aside from the results shown in Table 1, all performance analysis is evaluated against PyTorch Eager.

PyTorch Eager is the default execution mode of PyTorch, which dynamically executes computation by invoking calls to highly optimized closed-source kernels.

PyTorch Compile or `torch.compile` uses rule-based heuristics over the underlying computation graph during an initial compilation phase and invokes various backends to perform optimizations like

kernel fusion and graph transformations. In Table 1, our performance baseline for `torch.compile` assumes the default configuration using PyTorch Inductor in default mode. Furthermore, we **exclude** the `torch.compile` compile time in our timing analysis, as we are only interested in the raw runtime behavior. `torch.compile` features multiple other backends and configurations, which we describe in Table 3.

We observe that the `torch.compile` baseline runtime is generally faster on Level 2 and 3 of KernelBench reference problems compared to PyTorch Eager, mostly due to the availability of graph-level optimizations like operator fusion. However, on Level 1 problems, `torch.compile` can exhibit higher runtimes than PyTorch Eager, which can be attribute to empirically-reproducible runtime overhead for `torch.compile` (*not compile time*) that is significant for small kernels.

Configuration	Backend	Mode	Description
PyTorch (Eager)	-	-	Standard PyTorch eager execution
Torch Compile	inductor	default	Default `torch.compile` behavior
Torch Compile	inductor	reduce-overhead	Optimized for reduced overhead
Torch Compile	inductor	max-autotune	Maximum autotuning enabled
Torch Compile	inductor	max-autotune-no-cudagraphs	Maximum autotuning without CUDA graphs
Torch Compile	cudagraphs	-	CUDA graphs with AOT Autograd

Table 3: Configurations and modes for PyTorch execution and optimization backends.

Other `torch.compile` backends. In Table 4, we show more one-shot baseline results for fast_1 against some of the other `torch.compile` baselines. We note on some other configurations fast_1 drops especially for Level 2, as the `torch.compile` backends apply more aggressive optimization (at the cost of extra compile-time overhead, which we do not measure). Due to the variability of `torch.compile` across configurations, we focus our analysis on PyTorch Eager.

fast_1 over:	torch.compile default			cudagraphs			max-autotune			max-autotune no-cudagraphs			reduce-overhead		
KernelBench Level	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Claude 3.5 Sonnet	29%	2%	2%	31%	7%	2%	31%	2%	0%	29%	2%	2%	31%	2%	0%
DeepSeek V3	20%	2%	2%	21%	4%	20%	21%	2%	2%	20%	2%	2%	21%	2%	0%
DeepSeek R1	**38%**	**37%**	2%	**42%**	**52%**	0%	**42%**	**29%**	0%	**38%**	**32%**	4%	**42%**	**28%**	0%
GPT-4o	18%	4%	**4%**	22%	6%	6%	21%	4%	2%	18%	3%	4%	21%	4%	0%
Llama 3.1-70B Inst.	11%	0%	0%	12%	0%	0%	12%	0%	0%	11%	0%	0%	12%	0%	0%
Llama 3.1-405B Inst.	16%	0%	0%	16%	0%	4%	16%	0%	0%	16%	0%	0%	16%	0%	0%
OpenAI O1	28%	19%	**4%**	33%	37%	**26%**	34%	8%	**4%**	30%	19%	**6%**	34%	8%	**2%**

Table 4: We compare KernelBench `torch.compile` baseline runtime across various configurations, all measured on NVIDIA L40S, in addition to what is showed in Table 1.

D EXPERIMENT PROMPTING DETAILS

We provide details for the prompting strategies and associated sampling strategies used in Section 4 and Section 5.

D.1 ONE-SHOT BASELINE PROMPT

For the one-shot baseline as shown in Section 4.1, we want to examine each model's out-of-the-box ability to generate kernels by providing the minimum set of information while ensuring the instructions and output format are clear. We query each model with the following prompt and a pair of in-context `add` examples (the PyTorch reference `add` and its CUDA kernel counterpart using inline compilation) to provide the output format. We sample the model with greedy decoding to ensure deterministic output, which is setting temperature $= 0$.

```
You write custom CUDA kernels to replace the pytorch operators in the given architecture
to get speedups.

You have complete freedom to choose the set of operators you want to replace. You may
make the decision to replace some operators with custom CUDA kernels and leave others
unchanged. You may replace multiple operators with custom implementations, consider
operator fusion opportunities (combining multiple operators into a single kernel, for
example, combining matmul+relu), or algorithmic changes (such as online softmax). You are
only limited by your imagination.

Here\'s an example to show you the syntax of inline embedding custom CUDA operators in
torch: The example given architecture is:
'''
import torch
import torch.nn as nn
import torch.nn.functional as F


class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()

    def forward(self, a, b):
        return a + b


def get_inputs():
    # randomly generate input tensors based on the model architecture
    a = torch.randn(1, 128).cuda()
    b = torch.randn(1, 128).cuda()
    return [a, b]


def get_init_inputs():
    # randomly generate tensors required for initialization based on the model architecture
    return []
'''

The example new arch with custom CUDA kernels looks like this:
'''
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.cpp_extension import load_inline

# Define the custom CUDA kernel for element-wise addition
elementwise_add_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void elementwise_add_kernel(const float* a, const float* b, float* out, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        out[idx] = a[idx] + b[idx];
    }
}
```

```

torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor b) {
    auto size = a.numel();
    auto out = torch::zeros_like(a);

    const int block_size = 256;
    const int num_blocks = (size + block_size - 1) / block_size;

    elementwise_add_kernel<<<num_blocks, block_size>>>(a.data_ptr<float>(), b.data_ptr<float>(), out.data_ptr<float>(), size);

    return out;
}
"""

elementwise_add_cpp_source = "torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor b);"

# Compile the inline CUDA code for element-wise addition
elementwise_add = load_inline(
    name='elementwise_add',
    cpp_sources=elementwise_add_cpp_source,
    cuda_sources=elementwise_add_source,
    functions=['elementwise_add_cuda'],
    verbose=True,
    extra_cflags=[''],
    extra_ldflags=['']
)

class ModelNew(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.elementwise_add = elementwise_add

    def forward(self, a, b):
        return self.elementwise_add.elementwise_add_cuda(a, b)
```

You are given the following architecture:

<PyTorch reference architecture for specific KernelBench Problem>

Optimize the architecture named Model with custom CUDA operators! Name your optimized output architecture ModelNew. Output the new code in codeblocks. Please generate real code, NOT pseudocode, make sure the code compiles and is fully functional. Just output the new model code, no other text, and NO testing code!

D.2 Repeated Sampling Prompts

For repeated sampling, we use the same prompt that we used for the one-shot baseline in Appendix D.1. We used the same sampling temperature described in Brown et al. (2024) as they allow sample diversity while ensuring quality. Specifically we use temperature $= 1.6$ for Deepseek-V3 and temperature $= 0.7$ for Llama 3.1-70B.

D.3 Iterative Refinement Prompts

For iterative refinement, we start with the same initial prompt that we used for the one-shot baseline in Appendix D.1. A limitation of our experiments is that we sample with temperature$= 0$ to focus on the

effect of iterating based on feedback rather than introducing variability. On subsequent generations, we prompt the model with the following template depending on the feedback it expects:

```
<Initial prompt from one-shot baseline for specific KernelBench problem.>

Here is your latest generation:
<Previously generated kernel G>

Your generated architecture ModelNew and kernel was evaluated on GPU and
    ↪ checked against the reference architecture Model.
Here is your Evaluation Result:

<Raw Compiler and Execution Feedback from stdout>

<'if correct:'>
Your kernel executed successfully and produced the correct output.
Here is your wall clock time: {runtime} milliseconds

<Profiler information if used and correct.>

Name your new improved output architecture ModelNew. Output the new code
    ↪ in codeblocks. Please generate real code, NOT pseudocode, make sure
    ↪  the code compiles and is fully functional. Just output the new
    ↪ model code, no other text, and NO testing code!
```

For the compiler and execution feedback, we handle timeouts and deadlocks explicitly with "Your kernel execution timed out", but do not provide any other information.

D.4 Few-Shot in Context Prompts

For Few-Shot experiments as outlined in Section 5.2.1. We provide more details about the in-context example in Appendix G. We sampled these experiments with temperature $= 0$.

```
<Initial Task prompt from one-shot baseline for Instruction>
<Initial pair of Reference PyTorch and CUDA kernel equiavlent for example
    ↪  add kernel from one-shot baseline for Instruction>

Example <i>
Here is an example architecture
<PyTorch reference architecture for No. i in-context example>

Here is an optimized verison with custom CUDA kernels:
<PyTorch architecture with Custom CUDA Kernel for No. i in-context
    ↪ example>

.. up to number of in-context sample times


Task:
Here is an example architecture:

<PyTorch reference architecture for specific KernelBench Problem>

Name your new improved output architecture ModelNew. Output the new code
    ↪ in codeblocks. Please generate real code, NOT pseudocode, make sure
    ↪  the code compiles and is fully functional. Just output the new
    ↪ model code, no other text, and NO testing code!
```

D.5 Hardware Case Study Prompts

Here we provide hardware information. This is used in Section 4.4 and elaborated more in H, sampled with temperature $= 0$.

```
<Initial Task prompt from one-shot baseline for Instruction>
<Initial pair of Reference PyTorch and CUDA kernel equiavlent for example add kernel from one-shot baseline for Instruction>

Here is some information about the underlying hardware that you should keep in mind.

The GPU that will run the kernel is NVIDIA <GPU NAME>.

- We have <x> GB GDDR6 with ECC of GPU Memory.
- We have <x> GB/s of Memory Bandwidth.
- We have <x> of RT Core Performance TFLOPS.
- We have <x> of FP32 TFLOPS.
- We have <x> of TF32 Tensor Core TFLOPS.
- We have <x> of FP16 Tensor Core TFLOPS.
- We have <x> of FP8 Tensor Core TFLOPS.
- We have <x> of Peak INT8 Tensor TOPS.
- We have <x> of Peak INT4 Tensor TOPS.
- We have <x> 32-bit registers per SM of Register File Size.
- We have <x> of Maximum number of registers per thread.
- We have <x> of Maximum number of thread blocks per SM.
- We have <x> KB of Shared memory capacity per SM.
- We have <x> KB of Maximum shared memory per thread block.



Here are some concepts about the GPU architecture that could be helpful:

- Thread: A thread is a single execution unit that can run a single instruction at a time.
- Thread Block: A thread block is a group of threads that can cooperate with each other.
- Shared Memory: Shared memory is a memory space that can be accessed by all threads in a thread block.
- Register: A register is a small memory space that can be accessed by a single thread.
- Memory Hierarchy: Memory hierarchy is a pyramid of memory types with different speeds and sizes.
- Memory Bandwidth: Memory bandwidth is the rate at which data can be read from or stored into memory.
- Cache: Cache is a small memory space that stores frequently accessed data.
- HBM: HBM is a high-bandwidth memory technology that uses 3D-stacked DRAM.

Here are some best practices for writing CUDA kernels on GPU

- Find ways to parallelize sequential code.
- Minimize data transfers between the host and the device.
- Adjust kernel launch configuration to maximize device utilization.
- Ensure that global memory accesses are coalesced.
- Minimize redundant accesses to global memory whenever possible.
- Avoid long sequences of diverged execution by threads within the same warp.
  #We added this to reference the specific GPU architecture
- Use specialized instructions based on the specific GPU architecture

You are given the following architecture:

<PyTorch reference architecture for specific KernelBench Problem>

Name your new improved output architecture ModelNew. Output the new code in codeblocks. Please generate real code, NOT pseudocode, make sure
```

```
    ↪  the code compiles and is fully functional. Just output the new
    ↪ model code, no other text, and NO testing code!
```

E KERNELS OF INTEREST

In this section we provide examples of interesting or notable kernel generations. We first expand on the discussion in Section 6, where we defined the following categories of optimizations: algorithmic optimizations, operator fusion, and using hardware features.

E.1 ALGORITHMIC OPTIMIZATIONS

***13x Speedup* on Level 1 Problem 11 by Claude-3.5 Sonnet**
The original torch operator is `torch.diag(A) @ B`, multiplying a diagonal matrix formed from the vector `A` with the matrix `B`. The model identifies an optimization in the special case of a diagonal matrix multiplication, where the diagonal matrix doesn't need to be explicitly constructed. Instead, each element of the vector `A` is directly multiplied with the corresponding row in matrix `B`, significantly improving performance:

```
__global__ void diag_matmul_kernel(
    const float* diag,
    const float* mat,
    float* out,
    const int N,
    const int M) {

    const int row = blockIdx.y * blockDim.y + threadIdx.y;
    const int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < M) {
        out[row * M + col] = diag[row] * mat[row * M + col];
    }
}
```

E.2 KERNEL FUSION

***2.9x Speedup* on Level 1 Problem 87 by DeepSeek-V3**
GeLU reference in torch:

```
0.5 * x * (1.0 + torch.tanh(math.sqrt(2.0 / math.pi) * (x + 0.044715 *
    ↪ torch.pow(x, 3.0))))
```

Optimized version fuses in a single kernel. There is also a small constant folding optimization, instead of computing `math.sqrt(2.0 / math.pi)` repeatedly, the kernel uses the precomputed value `0.7978845608028654f`:

```
__global__ void gelu_kernel(const float* x, float* out, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        float x_val = x[idx];
        float cdf = 0.5f * (1.0f + tanhf((0.7978845608028654f * (x_val +
            ↪ 0.044715f * x_val * x_val * x_val))));
        out[idx] = x_val * cdf;
    }
}
```

***1.3x Speedup* on Level 1 Problem 29 by Claude-3.5 Sonnet**
SoftSign reference in torch:

```
x / (1 + torch.abs(x))
```

Fused kernel:

```
__global__ void softsign_kernel(const float* input, float* output, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        float x = input[idx];
        float abs_x = abs(x);
        output[idx] = x / (1.0f + abs_x);
    }
}
```

***2.6x Speedup* on Level 2 Problem 13 by Claude-3.5 Sonnet**
Sequence of operators in torch:

```
x = torch.matmul(x, self.weight.T)  # Gemm
x = x / 2  # Divide
x = torch.sum(x, dim=1, keepdim=True) # Sum
x = x * self.scaling_factor  # Scaling
```

Fused kernel:

```
__global__ void fused_ops_kernel(
    const float* input,
    const float* weight,
    float* output,
    const float scaling_factor,
    const int batch_size,
    const int input_size,
    const int hidden_size
) {
    // Each thread handles one element in the batch
    const int batch_idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (batch_idx < batch_size) {
        float sum = 0.0f;

        // Compute matmul and divide for this batch element
        for(int h = 0; h < hidden_size; h++) {
            float elem = 0.0f;
            for(int i = 0; i < input_size; i++) {
                elem += input[batch_idx * input_size + i] *
                        weight[h * input_size + i];
            }
            // Divide by 2 as we go
            sum += (elem / 2.0f);
        }

        // Scale and store final result
        output[batch_idx] = sum * scaling_factor;
    }
}
```

Despite this good example, overall Level 2 generated kernels show insufficient fusion. It is expected that most Level 2 problems can be expressed in a single fused kernel.

***1.9x Speedup* on Level 3 Problem 49 by OpenAI-o1**
We have a hypothetical architecture of the attention mechanism where the softmax is replaced with a ReLU.

```
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
    att = F.relu(att)
```

The model found an optimization that fuses the scaling, masked fill, and ReLU but not anything else, resulting in a modest improvement of 1.9x.

```
__global__ void fused_masked_fill_scale_relu_kernel(
    const float* __restrict__ att,
    const float* __restrict__ bias,
    float* __restrict__ output,
    int total_elems,
    float scale,
    int T,
    float negative_infinity
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < total_elems) {
        float val = att[idx] * scale;
        int bias_idx = idx % (T * T);
        if (bias[bias_idx] == 0.0f) {
            val = negative_infinity;
        }
        if (val < 0.0f) {
            val = 0.0f;
        }
        output[idx] = val;
    }
}
```

E.3 HARDWARE FEATURES

***2.8x Speedup* on Level 1 Problem 96 by OpenAI-o1**
Torch reference for Cosine Similarity Loss

```
cosine_sim = torch.nn.functional.cosine_similarity(predictions, targets,
    ↪ dim=1)
return torch.mean(1 - cosine_sim)
```

The generated kernel uses shared memory for reduce redundant global memory accesses, improving data locality and increasing overall performance. This is a moderately complicated kernel with synchronization points and reductions that would be tricky for humans to get right.

```cpp
__global__ void cosine_similarity_loss_kernel(
    const float* __restrict__ predictions,
    const float* __restrict__ targets,
    float* __restrict__ losses,
    const int batch_size,
    const int input_size
) {
    // Each block handles one sample in the batch
    int sample_idx = blockIdx.x;
    if (sample_idx >= batch_size) return;

    // Shared memory for reductions
    extern __shared__ float sdata[];

    // Pointers to data for this sample
    const float* pred = predictions + sample_idx * input_size;
    const float* targ = targets + sample_idx * input_size;

    // Intermediate sums for dot product and norms
    float thread_dot = 0.0f;
    float thread_pred_norm_sq = 0.0f;
    float thread_targ_norm_sq = 0.0f;

    for (int idx = threadIdx.x; idx < input_size; idx += blockDim.x) {
        float p = pred[idx];
        float t = targ[idx];
        thread_dot += p * t;
        thread_pred_norm_sq += p * p;
        thread_targ_norm_sq += t * t;
    }

    // Reduction for dot product
    sdata[threadIdx.x] = thread_dot;
    __syncthreads();
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (threadIdx.x < s) {
            sdata[threadIdx.x] += sdata[threadIdx.x + s];
        }
        __syncthreads();
    }
    float dot_product = sdata[0];

    // Reduction for pred_norm_sq
    sdata[threadIdx.x] = thread_pred_norm_sq;
    __syncthreads();
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (threadIdx.x < s) {
            sdata[threadIdx.x] += sdata[threadIdx.x + s];
        }
        __syncthreads();
    }
    float norm_pred = sqrtf(sdata[0] + 1e-8f);

    // Reduction for targ_norm_sq
    sdata[threadIdx.x] = thread_targ_norm_sq;
    __syncthreads();
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (threadIdx.x < s) {
            sdata[threadIdx.x] += sdata[threadIdx.x + s];
        }
        __syncthreads();
    }
    float norm_targ = sqrtf(sdata[0] + 1e-8f);

    if (threadIdx.x == 0) {
        float cosine_sim = dot_product / (norm_pred * norm_targ + 1e-8f);
        losses[sample_idx] = 1.0f - cosine_sim;
    }
}
```

***1.9x Speedup* on Level 1 Problem 98 by Deepseek-R1**
Torch reference for Cosine Similarity Loss

```
self.loss_fn = torch.nn.TripletMarginLoss(margin=margin)
self.loss_fn(anchor, positive, negative)
```

Another example of a generated kernel using shared memory:

```
__global__ void triplet_margin_loss_kernel(
    const float* anchor,
    const float* positive,
    const float* negative,
    float* losses,
    float margin,
    int feature_size)
{
    extern __shared__ float shared_sums[];

    int batch_idx = blockIdx.x;
    int tid = threadIdx.x;

    int offset = batch_idx * feature_size;

    const float* a = anchor + offset;
    const float* p = positive + offset;
    const float* n = negative + offset;

    float a_p_sum = 0.0f;
    float a_n_sum = 0.0f;

    int stride = blockDim.x;
    for (int i = tid; i < feature_size; i += stride) {
        float diff_ap = a[i] - p[i];
        a_p_sum += diff_ap * diff_ap;
        float diff_an = a[i] - n[i];
        a_n_sum += diff_an * diff_an;
    }

    shared_sums[tid] = a_p_sum;
    shared_sums[blockDim.x + tid] = a_n_sum;

    __syncthreads();

    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            shared_sums[tid] += shared_sums[tid + s];
            shared_sums[blockDim.x + tid] += shared_sums[blockDim.x + tid
                ↪  + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        float d_ap = sqrtf(shared_sums[0]);
        float d_an = sqrtf(shared_sums[blockDim.x]);
        losses[batch_idx] = fmaxf(d_ap - d_an + margin, 0.0f);
    }
}
```

E.4 Iterative Refinement Examples

E.4.1 Iteratively Trying new Optimizations

We provide an example of a kernel that iteratively improves on its existing generation. In the following example, the model attempts new optimizations incorrectly, fixes them, and continue to attempt new optimizations, improving its kernel to faster than the `torch.compile` baseline (1.34ms) but short of the Torch Eager baseline (0.47ms).

Level 1, Problem 63: 2D convolution with square input and square kernel. DeepSeek-R1 with Execution and Profile Feedback

	Turn 1	Turn 2	Turn 3	Turn 4	Turn 5	Turn 6	Turn 7	Turn 8	Turn 9	Turn 10
Compiles?	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓
Correct?	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓
Runtime (ms)	9.1	-	1.57	-	1.83	1.43	-	**1.13**	-	1.46

Table 5: Iterative refinement trajectory of DeepSeek-R1 with execution feedback E and profiler feedback P on Problem 63, Level 1. Torch Eager baseline runs in 0.47ms and `torch.compile` runs in 1.34ms.

In this example, we see a $8\times$ speedup in average kernel runtime from its initial generation, where the model repeatedly (incorrectly) refines its kernel, fixes the compiler issues using feedback, then continues to attempt more optimizations. The first big jump in performance (Turn 1 $\rightarrow$ Turn 3) occurs because the model decides to launch thread blocks along an output channel dimension, when it originally computed these elements sequentially. The model then attempts to use shared memory in Turn 5, and continues using it, along with texture cache memory with the `__ldg` instruction in Turns 7 and 8.

E.4.2 Leveraging Feedback to Correct Kernel Code

Level 2, Problem 73: 2D Convolution with a BatchNorm and a scale factor. DeepSeek-R1 with Execution Feedback We provide an example of a kernel that the model struggles to generate correctly, and produces a correct kernel after iterative refinement using execution feedback.

	Turn 1	Turn 2	Turn 3	Turn 4	Turn 5	Turn 6	Turn 7	Turn 8	Turn 9	Turn 10
Compiles?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Correct?	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Runtime	-	-	-	-	-	-	-	-	-	**3.16**

Table 6: Iterative refinement trajectory of DeepSeek-R1 with execution feedback E on Problem 73, Level 2. Torch Eager baseline runs in 0.105ms and `torch.compile` runs in 0.156ms.

In the above example, the model continually produces either the wrong output tensor shape or the wrong values and iterates on its kernel using this feedback until the final turn, where it generates a functionally correct, albeit non-performant kernel. We provide another example below that explicitly leverages compiler feedback to fix compiler errors:

Level 2, Problem 23: 3D Convolution with a GroupNorm and return the mean across all but the batch dimension. DeepSeek-R1 with Execution Feedback

In the above example, the model attempts to use the `CUB` library, but incorrectly invokes function calls. The model is then able to correct these errors and write a slightly faster kernel in Turn 8.

E.4.3 Iterative Refinement Never Fixes the Error

Level 1, Problem 54: 3D Convolution square input and square kernel. DeepSeek-R1 with Execution and Profiler Feedback

	Turn 1	**Turn 2**	**Turn 3**	**Turn 4**	**Turn 5**	**Turn 6**	**Turn 7**	**Turn 8**	**Turn 9**	**Turn 10**
Compiles?	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓
Correct?	✗	✗	✓	✓	✗	✗	✓	✓	✗	✗
Runtime	-	-	11.4	1.36	-	-	1.39	**1.33**	-	-

Table 7: Iterative refinement trajectory of DeepSeek-R1 with execution feedback E on Problem 23, Level 2. Torch Eager baseline runs in 1.29ms and `torch.compile` runs in 0.719ms.

	Turn 1	**Turn 2**	**Turn 3**	**Turn 4**	**Turn 5**	**Turn 6**	**Turn 7**	**Turn 8**	**Turn 9**	**Turn 10**
Compiles?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Correct?	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Runtime	-	-	-	-	-	-	-	-	-	-

Table 8: Iterative refinement trajectory of DeepSeek-R1 with execution feedback E and profiler feedback P on Problem 54, Level 1. Torch Eager baseline runs in 4.47ms and `torch.compile` runs in 4.67ms.

This problem is particularly interesting because no model is able to consistently produce functional code for this kernel, even with different forms of feedback and profiling information. Interestingly, the example before is an arguably more difficult version of this kernel that fuses the 3D convolution with another operator, and the same model is able to generate functional code for this task. In the example above, the model consistently makes the same mistake and continually generates a functionally incorrect kernel with the same value errors.

F Iterative Refinement on Correctness

Here we show that fast_0 across iterative refinement 5.1.2 configurations at a turn budget of $N = 10$ compared to one-shot baseline 4.1. We find that models self-correct more effectively with execution feedback E, fixing issues especially related to execution errors. Notably, DeepSeek-R1 on Level 1 and 2 can generate a functional kernel on >90% of the tasks given 10 turns of iterative refinement. However, the remaining incorrect kernels almost always fail due to functional incorrectness, likely because correctness feedback is less granular than execution failure messages

Method	**Level 1**			**Level 2**			**Level 3**		
	Llama 3.1 70B	DeepSeek V3	DeepSeek R1	Llama 3.1 70B	DeepSeek V3	DeepSeek R1	Llama 3.1 70B	DeepSeek V3	DeepSeek R1
Single Attempt (Baseline)	26%	43%	67%	0%	6%	62%	0%	30%	8%
Iterative Refinement (w G)	27%	48%	72%	2%	7%	67%	0%	36%	14%
Iterative Refinement (w G+E)	**40%**	**53%**	95%	**7%**	8%	85%	18%	42%	**50%**
Iterative Refinement (w G+E+P)	36%	50%	**95%**	**7%**	**9%**	**92%**	8%	**44%**	42%

Table 9: **Leveraging execution feedback helps reduce errors:** Here we present the percentage of problems where the LM-generated Kernel is correct for iterative refinement. We note leveraging execution feedback helps the model achieve better correctness fast_0, which is the percentage of problems where the model has at least one correct generation up to turn $N = 10$. We note the various iterative refinement configurations, leveraging previous Generation G, Execution Result E, and Timing Profiles P.

G Few Shot Experiment

For this experiment, we provide in-context examples of optimization techniques such as fusion, tiling, recompute, and asynchrony to models during kernel generation. As described in Section 5.2.1, we provide three in-context examples: a fused GELU Hendrycks & Gimpel (2023), a tiled

matrix multiplication Mills (2024), and a minimal Flash-Attention Dao et al. (2022); Kim (2024) demonstrating effective shared memory I/O management. The prompt used for this experiment is described in Appendix D.4. The speedup of these kernels were computed over PyTorch Eager. We compare the performance of these few-shot kernels over the one-shot baseline below.

		Baseline			Few-Shot		
Model	**Level**	**fast$_1$**	**fast$_0$**	**Kernel Length (chars)**	**fast$_1$**	**fast$_0$**	**Kernel Length (chars)**
Llama 3.1-70B	1	3%	27%	301018	6%	27%	360212
	2	0%	0%	646403	0%	0%	566668
	3	0%	0%	404596	0%	4%	485332
OpenAI o1	1	10%	55%	343995	6%	39%	437768
	2	24%	56%	381474	16%	39%	432800
	3	12%	56%	260273	8%	22%	364551

Table 10: Comparison of the Section 4.1 baseline and few-shot prompting performance across models. We examine the **fast$_0$**, **fast$_1$**, and cumulative character length of generated kernels per level.

77% of matrix multiplication problems in Level 1 achieves a speedup over the one-shot baseline through tiling. The runtime comparison for each GEMM variant is presented below as Table **??**.

Problem Name	**Baseline (ms)**	**Few-Shot (ms)**	**Ref Torch (ms)**
3D Tensor Matrix Multiplication	20.9	7.71	1.45
Matmul for Upper-Triangular Matrices	14	5.39	2.98
Matrix Scalar Multiplication	1.19	0.811	0.822
Standard Matrix Multiplication	3.39	2.46	0.397
Matmul with Transposed Both	3.44	2.67	0.412
Matmul with Transposed A	3.61	2.99	0.384
4D Tensor Matrix Multiplication	366	338	36
Tall Skinny Matrix Multiplication	3.39	3.59	1.9
Matmul with Diagonal Matrices	0.221	0.237	2.83

Table 11: Performance comparison of the Section 4.1 baseline and few-shot prompting in level 1 matrix multiplication problems.

Few-shot kernels generated for the following problems in level 2 outperformed PyTorch Eager through aggressive shared memory I/O management.

Problem Name	**Baseline (ms)**	**Few-Shot (ms)**	**Ref Torch (ms)**
Conv2d InstanceNorm Divide	0.514	0.0823	0.0898
Gemm GroupNorm Swish Multiply Swish	0.124	0.0542	0.0891
Matmul Min Subtract	0.0651	0.0342	0.0397
Matmul GroupNorm LeakyReLU Sum	0.0935	0.0504	0.072
ConvTranspose3d Swish GroupNorm HardSwish	33.3	29.6	35.2
ConvTranspose2d Mish Add Hardtanh Scaling	0.235	0.209	0.243
ConvTranspose3d Add HardSwish	15.6	14.1	22.2
ConvTranspose2d Add Min GELU Multiply	0.365	0.349	0.4
ConvTranspose2d BiasAdd Clamp Scaling Clamp...	0.3	0.31	0.368
Conv2d GroupNorm Tanh HardSwish ResidualAdd...	0.124	0.129	0.154
Conv2d ReLU HardSwish	0.0681	0.0711	0.0768

Table 12: Performance comparison of the Section 4.1 baseline and few-shot prompting in level 2 for problems whose few-shot kernels outperform PyTorch Eager.

H CROSS-HARDWARE CASE STUDY

H.1 EVALUATION ACROSS DIFFERENT HARDWARE

To evaluate how generated kernels fare across different hardware platforms, we utilize a number of different NVIDIA GPUs that span different micro-architectures and capabilities. The specific details for each is provided in Table 13.

Provider	GPU Type	Memory	Power	Microarchitecture	FP16 TFLOPS	Memory Bandwidth
Baremetal	NVIDIA L40S	48 GB	300W	Ada	362.05	864 GB/s
Baremetal	NVIDIA H100	80 GB	700W	Hopper	989.5	3350 GB/s
Serverless	NVIDIA L40S	48 GB	350W	Ada	362.05	864 GB/s
Serverless	NVIDIA A100	42 GB	400W	Ampere	312	1935 GB/s
Serverless	NVIDIA L4	24 GB	72W	Ada	121	300 GB/s
Serverless	NVIDIA T4	16 GB	70W	Turing	65	300 GB/s
Serverless	NVIDIA A10G	24 GB	300W	Ampere	125	600 GB/s

Table 13: Specifications of different GPUs, including memory, power consumption, micro-architecture, FP16 TFLOPS, memory bandwidth, and their providers.

We ran the same set of kernels generated in Section 4.1 on a variety of hardware (as listed in Table 13). We computed the fast_1 speedup against the PyTorch Eager baseline profiled on that particular hardware platform in Table 14.

Level	GPUs	Llama-3.1-70b-Inst	DeepSeek-V3	DeepSeek-R1
1	**L40S**	3%	6%	12%
	H100	2%	7%	16%
	A100	3%	7%	16%
	L4	2%	4%	15%
	T4	3%	7%	22%
	A10G	2%	7%	12%
2	**L40S**	0%	4%	36%
	H100	0%	4%	42%
	A100	0%	4%	38%
	L4	0%	4%	36%
	T4	0%	4%	46%
	A10G	0%	4%	47%
3	**L40S**	0%	8%	2%
	H100	0%	10%	2%
	A100	0%	8%	2%
	L4	0%	6%	2%
	T4	0%	10%	2%
	A10G	0%	10%	0%

Table 14: KernelBench result across multiple hardware types: Speedup (fast_1) over Torch Eager comparison of GPUs across different models and levels. The kernels used across different GPUs are the same as the ones generated for Single Attempt **without** hardware/platform specific information.

Based on the increased variability in fast_1 score for DeepSeek R1 as described in Section 4.4 and Table 14, we plot the individual speedups for each problem (in Levels 1 and 2) across different GPUs. Speedup is computed against PyTorch Eager and there is a horizontal line at $y = 1.0$ to mark the cutoff for fast_1.

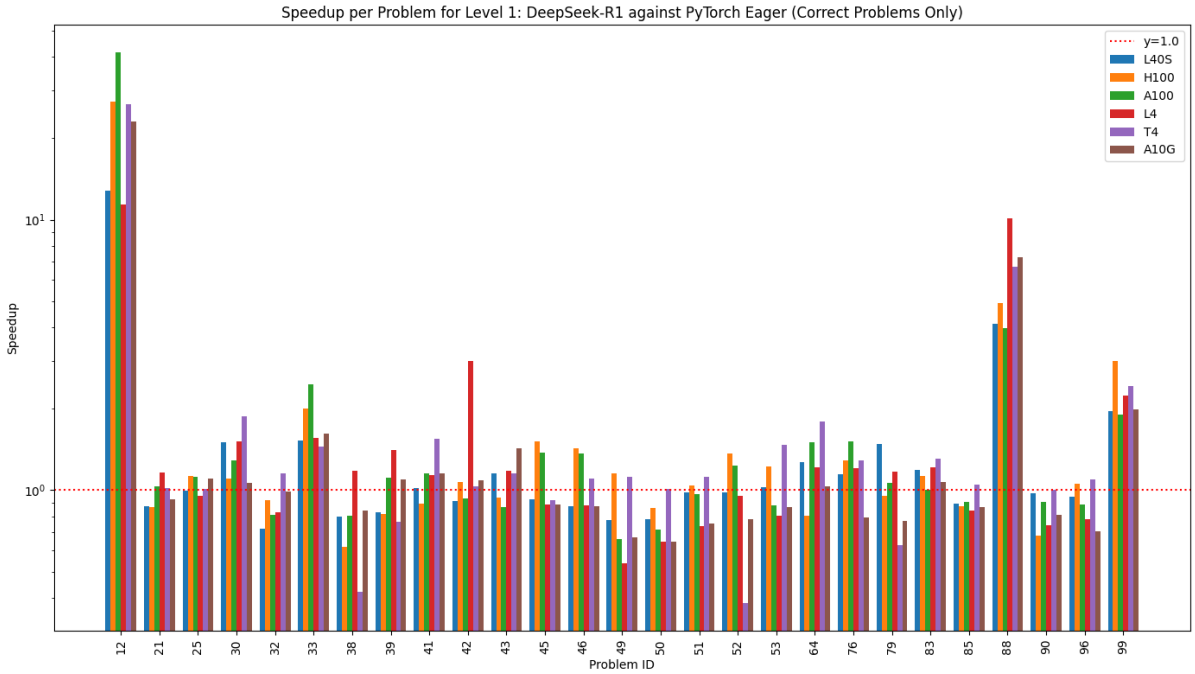


Figure 7: Speedup comparison across different GPUs for DeepSeek R1 on Level 1 (log scale).

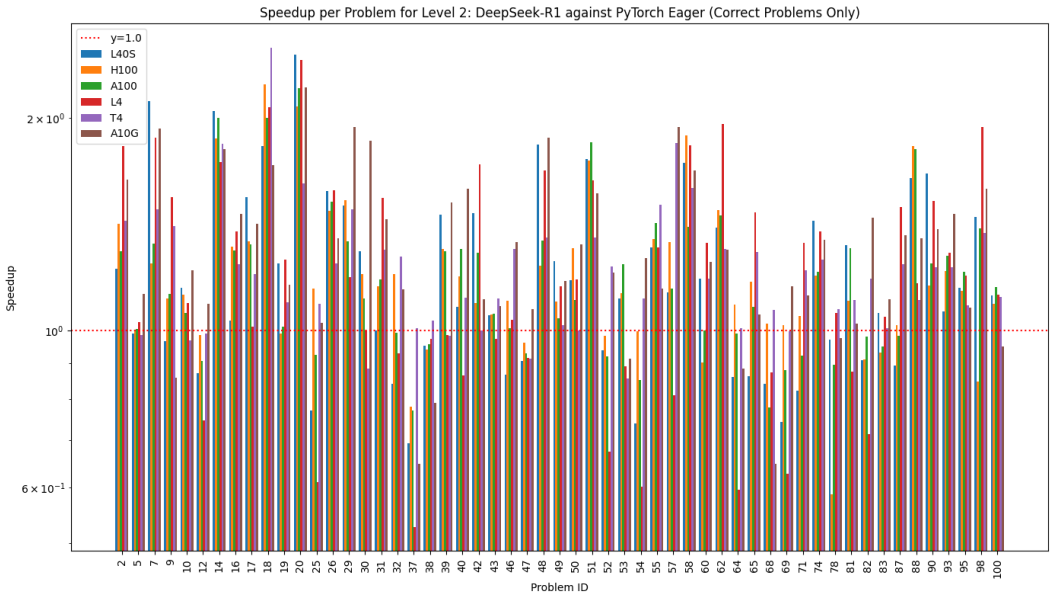


Figure 8: Speedup comparison across different GPUs for DeepSeek-R1 on Level 2 (log scale).

H.2 Effect of Providing Hardware Information

We provided hardware-specific information such as the GPU's micro-architecture, TFLOPS, and memory bandwidth (as shown in Table 13) to models during kernel generation. These generations were then profiled on their respective hardware platforms. The speedup in terms of fast_1 over PyTorch Eager is provided below in Table 15.

Level	GPUs	Llama-3.1-70b-Inst	DeepSeek-V3	DeepSeek-R1	OpenAI-o1
1	**L40S**	2%	5%	14%	13%
	H100	3%	6%	14%	9%
	A100	4%	4%	16%	9%
2	**L40S**	1%	4%	33%	25%
	H100	2%	2%	35%	33%
	A100	2%	1%	31%	32%
3	**L40S**	0%	2%	0%	10%
	H100	2%	8%	0%	12%
	A100	0%	0%	2%	8%

Table 15: Providing hardware specifications in context: Speedup ($fast_1$) over Torch Eager comparison of GPUs across different models and levels.

As described in Section 5.2.2, DeepSeek-R1 begins to generate kernels with warp matrix multiply-accumulate (`wmma`) instructions. We show the generated kernel for Problem 1 Level 1 when provided with H100 specifications in context below.

```cpp
#include <torch/extension.h>
#include <cuda_fp16.h>
#include <cuda_runtime.h>
#include <mma.h>

using namespace nvcuda;

__global__ void hgemm_kernel(const half* A, const half* B, half* C, int N) {
    const int WARPS_PER_BLOCK = 8;
    const int WMMA_M = 16;
    const int WMMA_N = 16;
    const int WMMA_K = 16;

    __shared__ half Ashared[WARPS_PER_BLOCK][WMMA_M][WMMA_K];
    __shared__ half Bshared[WARPS_PER_BLOCK][WMMA_K][WMMA_N];

    int warpId = threadIdx.x / 32;
    int laneId = threadIdx.x % 32;

    int tileM = (blockIdx.y * WARPS_PER_BLOCK + warpId) * WMMA_M;
    int tileN = blockIdx.x * WMMA_N;

    wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, half> c_frag;
    wmma::fill_fragment(c_frag, __float2half(0.0f));

    for (int k = 0; k < N; k += WMMA_K) {
        int loadAidx = laneId + (tileM + (laneId / WMMA_K)) * N + k + (laneId % WMMA_K);
        int loadBidx = laneId + (k + (laneId / WMMA_N)) * N + tileN + (laneId % WMMA_N);

        if (tileM + (laneId / WMMA_K) < N && k + (laneId % WMMA_K) < N)
            Ashared[warpId][laneId / WMMA_K][laneId % WMMA_K] = A[loadAidx];
        else
            Ashared[warpId][laneId / WMMA_K][laneId % WMMA_K] = __float2half(0.0f);

        if (k + (laneId / WMMA_N) < N && tileN + (laneId % WMMA_N) < N)
            Bshared[warpId][laneId / WMMA_N][laneId % WMMA_N] = B[loadBidx];
        else
            Bshared[warpId][laneId / WMMA_N][laneId % WMMA_N] = __float2half(0.0f);

        __syncthreads();

        wmma::load_matrix_sync(a_frag, &Ashared[warpId][0][0], WMMA_K);
        wmma::load_matrix_sync(b_frag, &Bshared[warpId][0][0], WMMA_N);
        wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

        __syncthreads();
    }

    int storeCidx = (tileM + (laneId / WMMA_N)) * N + tileN + (laneId % WMMA_N);
    if (tileM + (laneId / WMMA_N) < N && tileN + (laneId % WMMA_N) < N)
        C[storeCidx] = c_frag.x[laneId];
}

torch::Tensor hgemm_cuda(torch::Tensor A, torch::Tensor B, int N) {
    auto C = torch::zeros({N, N}, A.options().dtype(torch::kFloat16));

    const int WARPS_PER_BLOCK = 8;
    dim3 grid((N + 15) / 16, (N + 15) / (16 * WARPS_PER_BLOCK));
```

I HIGH-THROUGHPUT EVALUATION SYSTEM

I.1 SINGLE-SHOT EXPERIMENTS: BATCHED KERNEL GENERATION

Given the high volume of GPU kernels to evaluate, we build a fast and highly-parallelized evaluation system, where we separate into the kernel generation and evaluation process into 3 stages, as shown in Figure 10.

- **Inference:** We query LMs in parallel and store the generated kernel.
- **CPU Pre-Compile:** We compile the model-generated kernels with `nvcc` for a specified hardware into a binary, parallelized on CPUs and each kernel binary is saved to their individual specific directory for caching.
- **GPU Evaluation:** With the kernel binary already built on CPU, we focus on evaluating multiple kernels in parallel across multiple GPU devices. However, to ensure accurate kernel timing, we only evaluate one kernel at time on one device.

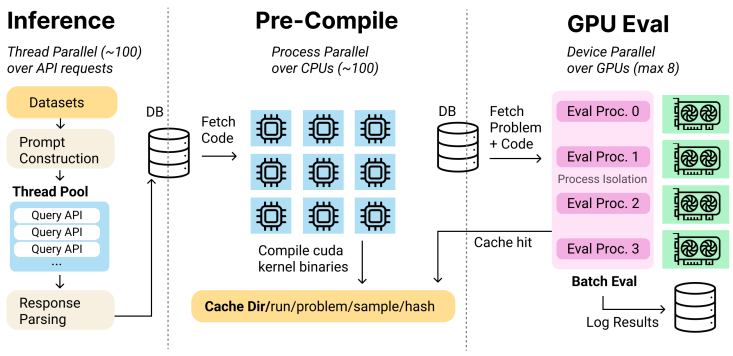


Figure 10: **KernelBench provide a high throughput kernel generation and evaluation system**. We parallelized generation, compilation, and evaluation of kernels across CPUs and GPUs.

I.2 ITERATIVE REFINEMENT EXPERIMENTS: GPU ORCHESTRATOR SYSTEM

Based on the single-shot system, we also design a platform to handle multiple iterative refinement experiments at once. We treat each iterative refinement experiment as a finite state machine, where the states are LM-based kernel generation, pre-compilation, kernel execution, and profiling. The transitions are based on environment feedback, and can change based on different experiment setups.

Our system was run on a node with 8 available GPUs. Unlike the single-shot system, batching each generation and kernel execution is highly inefficient – thus, we design a pipelined, multiprocessing system with a GPU orchestrator with the following characteristics:

- **CPU Parallelism:** The orchestrator spawns multiple independent processes that each handle an independent task in KernelBench. These processes run the multi-turn state machine logic for the iterative refinement experiments – only the kernel execution state requires acquiring a GPU.
- **Acquiring GPUs:** The GPU orchestrator keeps a separate process running that handles which processes can acquire a GPU using semaphores. Processes can request a GPU from this process when it is ready to execute and evaluate kernel code. We try to minimize process control over a GPU to maximize resource throughput, given a system with a limited number of available GPUs.
- **Pre-compiling on the CPU:** To avoid processes hogging GPU time, we pre-compile kernels with `nvcc` on the CPU for a specified hardware into a binary. We also did this same trick for the single-shot system, but for separate reasons.

- **Evaluating Kernels on the GPU:** The only state where the finite state machine uses the GPU is for kernel execution and profiling. We found that waiting on GPUs is the primary bottleneck in the orchestrator, so we designed the orchestrator to maximize device occupancy.

The system generally supports overlapping the generation of kernel code and the execution of already-generated kernel code. There are also several unavoidable errors such as CUDA illegal memory accesses and deadlocks due to faulty kernel generations that the orchestrator solves by releasing and spawning new processes when encountered, and we wrote specifically handlers to ensure these errors are properly captured without crashing the orchestrator itself.

I.3 UI: Visualizing Kernel Generation Trajectories

To qualitatively observe the generated and compare them across techniques, we design an interface to easily visualize them. We provide this as part of the KernelBench framework.

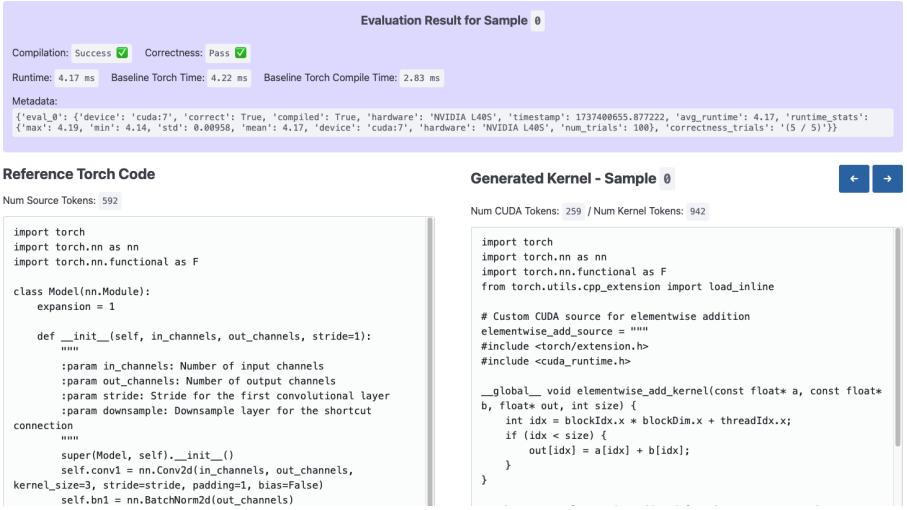


Figure 11: **We provide a visual interface for kernel inspection**. This allows us to easily examine kernel content, its performance, and compare across various techniques and configurations.