# Efficient CUDA Kernel Generation, Iteration, and Refinement via LLM

Jiaqi Wang
The University of Texas at Austin
Austin, Texas, USA
kelly.wang@utexas.edu

Logan Liberty
The University of Texas at Austin
Austin, Texas, USA
loganliberty@utexas.edu

Maxwell Poster
The University of Texas at Austin
Austin, Texas, USA
mposter@utexas.edu

Vineeth Narayan Pullu
The University of Texas at Austin
Austin, Texas, USA
vineethnp@utexas.edu

## Abstract

With the rise of of Machine learning models, the utility of efficient and performant CUDA Kernels has become extremely important. On the other hand, large language Models are rapidly improving in every generation for tasks like code generation. In this paper, we study the performance of these models in kernel generation, analyze the state of the art CUDA Kernel Generation techniques such as evolutionary computing, validate current claimed results, and further improve on these methodologies using state-of-the-art techniques like iterative refinement and repeated sampling. For most of the selected problems, the improved methodologies exceeded the highest speedup on KernelBench leaderboard, with the highest speedup of 2x while the champion speedup is 0.03x. We open source our evaluation and improvement techniques for the benefit of future CUDA kernel generation research. Overall, using language models to generate CUDA kernels is still a "mixed-bag," with no model capable of producing a compiling CUDA kernel in state-of-the-art benchmarks. However, a moderate portion of generated kernels improved wall-clock execution time over the native PyTorch compiler, suggesting that language models are capable of generating correct and efficient CUDA kernels in many cases.

## CCS Concepts

• **Computing methodologies** → **Kernel methods**; **Graphics processors**; **Natural language generation**; • **Information systems** → **Language models**; • **Software and its engineering** → **Source code generation**.

## Keywords

CUDA Kernel Generation, Large-Language Models

## 1 Introduction

The use of Large Language Models today is almost ubiquitous. Particularly, the past few generations of Models have improved significantly when it comes to code generation problems. But when it comes to CUDA Kernel Generation, the correctness and performance of the code depends on a multitude of factors like hardware specification, algorithm requirements, parallelizability of the program, required performance and resource availability [2][8]. Together, these factors make custom CUDA kernel writing a complex problem, which requires domain-specific knowledge. Though, there have been attempts to improve the simplicity of custom CUDA kernel writing via domain-specific languages like Triton [7] and completely original frameworks such as Thunderkitten [6], these have either been limited to particular applications or have not been able to achieve performance numbers that are possible while hand programming and optimizing CUDA. In this paper we explore the writing of CUDA Kernels using large language models in detail. We first analyze and breakdown the performance of the state of the models in various types of CUDA Kernels. We try to understand if the compilation success, correctness and performance of the kernels generated have correlation with respect what type of kernel it is and its complexity number. Moreover, iterative techniques such as repeated sampling (RS) and iterative refinement (IR) have proven invaluable to generating optimized kernels. To this end we improve on Kernel Bench [5], an open source CUDA kernel generation and evaluation framework. We implement these methods left out from the Kernel Bench open-source repository and attempt to verify their results. We then present a hybridization of both methods, called repeated temperature iterations (RTI). Sakana AI, have come up with their AI CUDA Engineer [1] which uses evolutionary computing to improve on CUDA kernels generated in four steps: 1) Conversion of Torch module into function. 2) Conversion of torch function into base cuda 3) Evolutionary optimisation from base cuda to optimized cuda and 4) Composition of in-context kernel examples given an archive of hundreds of working CUDA kernels and a target operation. We validate the claims made by them in the paper and infer the utility of evolutionary computing in the context of kernel generation. Our contributions are as follows:

- Detailed analysis of state of the art large language models and their performance with respect to various different cuda kernel generation problems.
- Improved Kernel Bench framework with off the shelf support for iterative refinement and temperature sampling. The code repository can be found here.
- Propose and implement a hybridization of RS and IR, called RTI, which seeks to obtain the benefits of both RS and IR.
- Compilation, correctness and performance exploration of kernel generation using iterative refinement and temperature sampling technqiues.
- Breakdown of Sakana AI's evolutionary computing technique for CUDA Kernel generation and the validation of the results provided by them.

## 2 Background

### 2.1 CUDA Framework

Programming efficient CUDA Kernels requires substantial effort from human experts. Mainstream high-level libraries such as CUTLASS, cuDNN, Apple MLX are hardware-specific, and require substantial manual development. While high-level Frameworks such as ThunderKitten and Triton aid custom kernel development by providing higher level abstractions, kernel programming likewise still requires significant human effort. The various factors that make CUDA Kernel generation a complex problem is as follows:

(1) **Parallelization**: Kernels run on hundreds of threads in parallel, which means that to extract good performance from these kernels means the algorithm needs to be efficiently parallelizable. Parallelization overhead may minimize potentially achievable speedup, especially for algorithms that are not straightforward to parallelize.
(2) **Memory Hierarchy**: The placement of data of where and when also makes considerable difference. As shown in Table 1, the size, access speed and scope varies with the memory, and the right decision making depends on the performance, resource availability and the cross communication required between memory points.
(3) **Synchronization**: On a GPU, thousands of lightweight threads progress in lock–step at the *warp* level and are further grouped into *thread-blocks*. Coordinating their execution is therefore far more involved than inserting a single barrier, as on the CPU. Within a block, the programmer must guarantee that *all* threads reach the same barrier on every dynamic path—otherwise the warp deadlocks.
(4) **Hardware information**: Attaining peak throughput demands intimate understanding of the target GPU architecture. Figure 1, shows NVIDIA's Pascal GPU architecture [3][4]. Understanding its specification, including register file size, the number of resident warps per Streaming Multiprocessor (SM), shared-memory capacity, L1/L2 cache behavior, warp-scheduler granularity, tensor-core availability, and even the precise latency of memory fences, are in crucial in attaining reasonable performance.
(5) **Debugging**: Since GPUs do not run serial code, debugging issues in the kernel, is dissimilar to debugging a regular program. Profilers such as NV-sight is used to understand what is happening in different GPU nodes to identify issues and improve the kernel.

| Type | Location | Size | Access speed | Scope |
|---|---|---|---|---|
| Global | Device-wide | Large | Slow | All threads |
| Shared | Per block | Limited | Fast | Threads in block |
| Registers | Per thread | Very small | Fastest | Single thread |
| Host memory | CPU side | Large | Must be copied | CPU |

**Table 1: CUDA memory hierarchy characteristics.**

[1]https://github.com/Vineeth-Narayan/Improved_Kernel_Bench

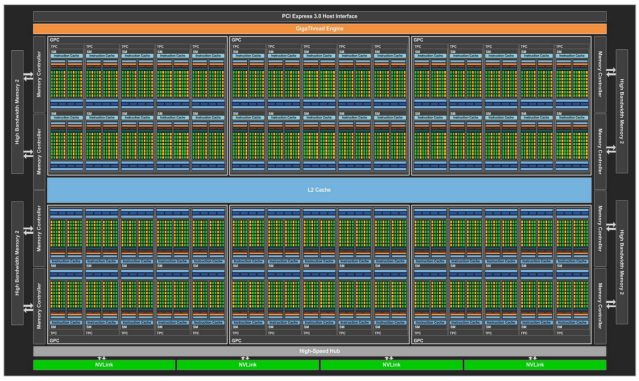


**Figure 1: Pascal GPU microarchitecture taken from [3]**

Compiler-level tools such as `torch.compile` and `FlexAttention` attempt to automate portions of this process, yet they either provide only mild, best-effort optimisations or focus on a narrow domain such as attention kernels (the latter). Domain-specific libraries like CUTLASS, cuDNN deliver hand-tuned, efficient kernels at the cost of portability and extensibility. More flexible, open-source frameworks like ThunderKitten, Triton—trade off some raw performance for programmability, still leaving the above challenges largely in the hands of human experts. Bridging this gap between *performance-critical, hardware-aware* code and *high-level, portable* abstractions remains an open research problem in GPU kernel generation.

### 2.2 Code Generation via LLMs

In recent years, using LLMs to generate common high-level languages such as Python and C has consistently seen improvements in both syntax and correctness. But, generation of low level kernels, like CUDA kernels LLMs still struggle with because of factors listed in the previous section, that is, requiring thorough architectural understanding, more complicated debugging and requiring custom memory placement. Moreover, given the closed source nature of CUDA programming, there is a lack of sufficient training data to adequately equip LLMs for CUDA generation [5]. To this end, LLMs have shown to improve with iterative feedback to its previous generation along with its errors each time [5], as shown in Figure 6 from [5]. Additionally, improvements for different issues do well for different types of prompting. This leads to requiring a framework, that adaptively changes its prompts and feedback for different issues, to iteratively improve the final code that is generated. We use the Kernel Bench framework [5] as a baseline to setup this framework.

### 2.3 KernelBench

*KernelBench* is an open-source benchmark and evaluation harness that tests whether large language models can automatically generate *fast and functionally correct* GPU kernels [5]. It offers **250** PyTorch workloads—ranging from single operators to fused operator pipelines and small end-to-end models—that serve as realistic optimisation targets. For each workload, as shown in figure 3, the model

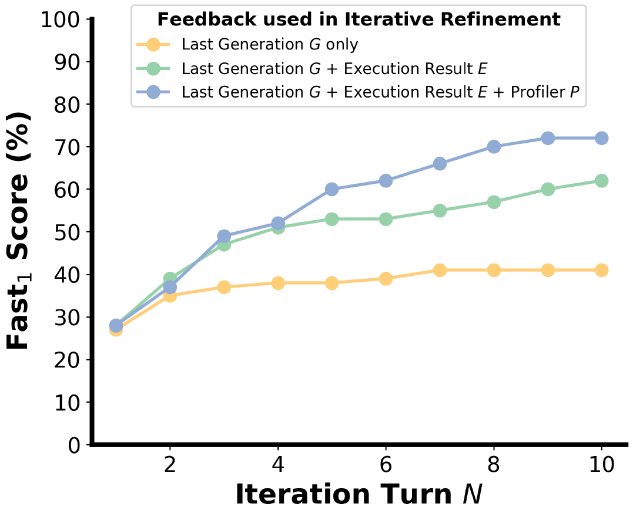


**Figure 2: Fast$_1$ score plotted against the number of iterative refinement *turns*, or rounds where the LLM was provided with $G$ - the previous kernel generation, $E$ - the compiler and execution feedback, and $P$ - the performance profile of the previous kernel generation. Figure courtesy of [5].**

receives the reference PyTorch `Module` and must emit a new implementation (`ModuleNew`) containing custom CUDA/Triton/PTX code wherever speed-ups are expected. The framework then *(i)* validates numerical correctness on randomly generated inputs and *(ii)* measures wall-clock execution time on real hardware. Performance is summarised by the fast$_p$ metric—the fraction of workloads whose generated kernel is correct and achieves at least a $p\times$ speed-up over the baseline PyTorch execution (with $p = 1$ as the headline setting). By coupling automatic compilation, runtime validation, and profiling, KernelBench mirrors the workflow of practitioners and turns progress on the benchmark into immediate, measurable gains for practical deep-learning kernels.

KernelBench separates the 250 tasks into 3 levels based on number of primitive operations and/or library functions.

- Level 1 (100 tasks) - Single primitive operations such as MatMul, losses, activations, normalizations, and convolutions
- Level 2 (100 tasks) - Operator Sequencing/Fusion
- Level 3 (50 tasks) - Full ML architectures, such as MiniGPT, AlexNet, though optimizations here occur primarily at the architecture/algorithmic level

## 2.4 Sakana AI's AI CUDA Engineer

Subsequent to KernelBench, Lange et al. introduced AI CUDA Engineer, an agentic framework that automates CUDA kernel discovery and optimization using evolutionary generation [1]. This framework focuses on inference-time techniques to improve samples drawn from pre-trained LLM agents, avoiding the cost of training large in-house models. Prior works in inference-time techniques include search algorithms, selection and aggregation techniques of parallel samples, sequential sampling, and evolutionary test-time

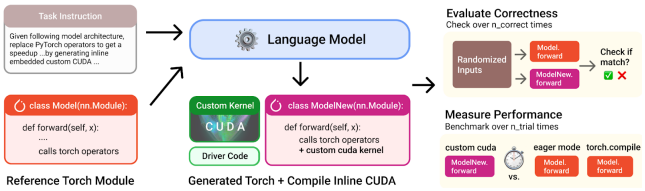


**Figure 3: Kernel-Bench Flow - Task instruction along with the class Model sent to the Language model, which generates the custom cuda kernel which is compiled, which is then evaluated on correctness and performance. Figure courtesy of [5].**

compute. In this work, Sakana AI chose to draw sequential samples and combine them using evolutionary optimization to construct a sophisticated iterative refinement workflow.

The framework decomposes translation from PyTorch modules to CUDA kernels into four stages, where code generation primarily takes place in stage 1-3:

(1) Convert original PyTorch module into a functional version via LLM queries. The functional version includes a docstring and explicitly seperate function parameters, and is otherwise identical to the original PyTorch module.
(2) Translate functional PyTorch to a functionally correct CUDA kernel using iterative refinement. Notably, an additional LLM query summarizes compilation or correctness error message to produce feedback.
(3) Optimize working CUDA kernels. A queue of up to 5 working kernels is maintained and taken as input at each iteration. An ensemble of 4 LLMs is queried with prompting techniques including sampling high-level optimization recommendations, crossover prompting, and temperature sampling when approrpriate. Those mechanisms are in place to promote diversity in generated code, which the authors believe to be crucial for getting good optimization results. `torch`, `NCU`, and `Clang-tidy` profiling information are included as feedback.
(4) Use retrieval augmented generation with an innovation archive of many kernels generated by stage 3 to output a final solution to a problem.

While the authors claimed to have an impressive result with 186/250 successful generations and a 1.52x median speedup from PyTorch, users have caught generated kernels that pass the correctness through memory exploits [9, 10], bringing up controversies regarding the validity of the published results. We will examine the details of their results in section 6.

# 3 Methods

## 3.1 Overview

In-house IR and RTI systems were implemented since i) [5] did not provide an open-source implementation of IR and ii) a hybridization of both RS and IR would serve to yield the benefits of both techniques. In IR, we aim to iteratively feed the outputs and erros as inputs to the Large Language Model hoping that over time, the model fixes each of the errors, and iteratively reaches a point

where the Kernel is optimised and functional. In temperature sampling, we vary the temperatures iterativey, with more temperauture allowing the LLM more creativity and with the expectation that this helps the LLM figure out issues it couldn't solve before. We use the following prompt engineering techniques to improve this mechanism. The base prompt is vital as it shows up in every iteration and needs to be generic enough for all kernels yet specific enough to ensure that just code kernel generation happens in every iteration. Modulized prompt constructors (e.g. modulized few shot examples, hardware info) are more flexible to combine. Furthermore, task-specific prompts can be used both for adding reminders for common mistakes, as well as implementing specific optimizations. Moreover, being able to retrieve compilation error messages was one of the most important things that was not included in Kernel Bench, which we believe is essential in getting iterative improvement in the kernel when providing feedback. Finally we try to include pytorch profiler statistics, to help the LLM, improve on performance. Plans to implement NVIDIA's Nsight Compute profiler, or NCU, to give more fine-grained hardware information in the IR feedback loop were intended, but time constraints prevented this from being implemented, and we leave this to future work.

## 3.2 Improved Kernel Bench

The framework Kernel bench already provides, allows us to get evaluation of the kernel in terms of compilation, correctness and performance. The improvements for Kernel Bench is to setup a framework on top, such that IR and RTI for the LLMs are automated such that they adapt to the particular error/bottlenecks the latest version of the kernel/code is presently facing as shown in figure 5. We address compilation failure, correctness failure, and performance improvement as disjoint problems. First for compilation issues, we iteratively feedback the error generated by NVCC while compiling the kernel back to the LLM. For correctness errors, we point to what kind of test it failed in, to improve on over next generation. Finally, for performance improvements we feedback the hardware specification and performance requirements to enable the LLM to give optimised kernels.

The key insight behind doing this is two-fold. We realize, that giving all the information to LLMs at once actually degrades the improvements, as there is no focused correction pointed to, for a particular kind of issue. Secondly, due to the disjoint structure, it allows a human to interact with the IR process more modularly, allowing picking a custom iteration number before moving on to the next step. This gives more freedom to optimize depending on how LLMs are doing for a particular kind of Kernel.

## 3.3 High Level Design

Results from several prior works including KernelBench [5], Sakana AI [1], and NVIDIA blog have shown effectiveness of IR for generating CUDA kernels with LLMs, while other prior works in code generation generally took the route of repeated sampling with high temperature. We recognize the advantage and limitation of both approaches and sought to combine the advantages of both. To combine IR and repeated sampling, We designed the RTI system to draw several samples with varying temperature and perform IR for each sample drawn (see Figure 4). We define $RTI(n, k, l)$ as taking $n$ initial samples of kernels with varying temperature, iterating on them $l$ times, and repeating this process $k$ times, using the best kernel from the previous layer as the initial sample for later layers.

Writing highly optimized code from scratch is difficult for both humans and LLMs. Real world programmers would often develop for correctness before starting to optimize it. Kernelbench authors also recognized that when LLMs to optimize more aggressively due to prompt engineering, the generated code is also more likely to be incorrect or fails to compile. Therefore, instead of the naive iterative refinement for generating an optimized kernel from a PyTorch problem, we sought to separate code generation for correctness and optimization of correct code into distinct stages. The approach allows us to give goal-specific prompts to LLMs, simplifying the task and avoiding compromising correctness when attempting to optimize.

Combining both design principles, in the generating for correctness stage, we allow the system to draw up to a limited number of kernels with high temperature, and perform a limited number of iterative refinements on each until one generated kernel is functionally correct. Then, using this correct kernel as initial input to iterative refinement for optimization. For optimization, we keep track of the best performing kernel so far and iteratively prompt the LLM to refine it with several prompting techniques combined.

## 3.4 Prompting

Many prompting engineering techniques have been explored in prior works: KernelBench explored using prompting with chain-of-thought instructions and examples, giving few-shot examples of optimized code, and giving hardware information for optimization. Sakana AI sampled high level recommendations for code optimization and performed crossover prompting every few iterations. We sought to combine those techniques, but with our own refinement.

To prompt for correctness, several components are in our prompt: first, a problem statement detailing the PyTorch problem to solve for and the output code format. Then, we instruct the LLM to perform chain-of-thought generation by thinking step by step and articulating thoughts in comments or outside of code blocks. Lastly, we include a reminder that contains common small mistakes that we've seen, such as "remember to import libraries used". Moreover, we only instruct the LLM to make sure that the kernel generated compiles and is functionally correct.

To prompt for optimization, we provide the best performing kernel so far to the LLM and instruct the LLM to optimize it. We may prompt with a high-level recommendation that we pre-defined, such as "try to use tensor core wmma instructions" and "perform tiling", and include few-shot code examples of that optimization if available, or, we may choose to prompt the LLM with a question of "what do you think that could be improved" instead. Additionally, we provide hardware information and general best practices for writing efficient kernels, as well as chain-of-thought instruction and reminders.

We also found several issues with KernelBench's existing prompting utilities and improved on them. First, possibly due to the experimental nature of their work, the prompting functions are poorly modularized, making it difficult to combine different prompting techniques. Secondly, we think there could be improvements on

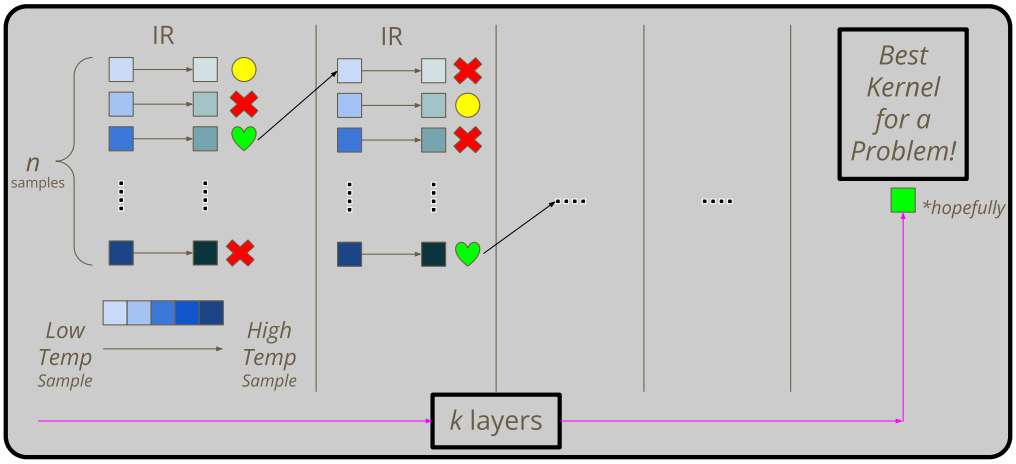


**Figure 4: RTI begins by taking $n$ samples at varying temperatures from the API of the selected model. Each sample is profiled for performance, and $l$ turns of IR are performed on each sample, profiling each iteration. Of all iterations across all temperature samples, we choose the best kernel, prioritizing first compilability, followed by correctness, followed by wall-clock execution time. The process is stochastic, so it's still possible that none of the generated kernels are even compilable by the final step, though more iterations at each level (samples, iterations, layers) have a tendency to help with this.**

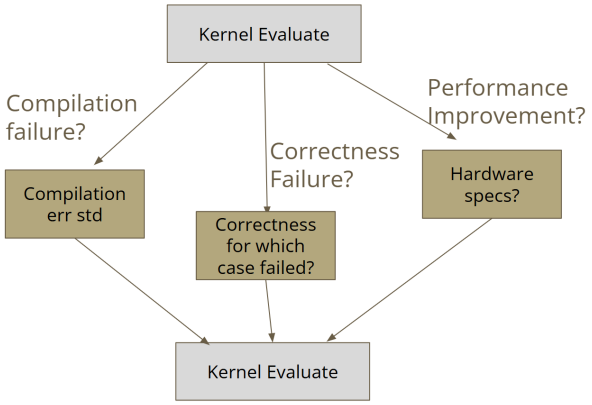


**Figure 5: Improved Kernel-Bench Feedback Flow - Compilation, Correctness and Performance are seen as fully disjoint problems, and iteratively improved. Compilation is fixed via errors that std out gives, correctness through which kernel went wrong and performance through hardware info.**

the thoughtfulness of prompts. Other than adding the additional prompting utilities that we needed, modifications we made include modularization of prompt constructions, removing redundant prompt components, and flexible few-shot prompting.

### 3.5 Feedback

Extracting compilation error message in the case when kernel fails to compile turns to be actually tricky, as the authors of KernelBench neither figured out a graceful way to do so in their open-sourced framework. In our implementation, output will be redirected to a file, from which we can later read in the compilation error message, and will provide the exact error message that a human programmer would see to the LLM. In case of correctness failure, we provide the LLM with the type of failure (shape mismatch or value mismatch) and provide the maximum difference as a feedback. In our current implementation, we provide only the average runtime of the generated kernel and baseline torch runtime if torch is faster, as well as PyTorch profiler feedback as an option.

## 4 Results: LLM CUDA Code Generation

To establish a baseline for the capabilities of LLMs in generating CUDA code, we evaluated two distinct models: OpenAI o1-mini and Google Gemini 2.5 Pro. These models were tasked with generating CUDA kernels for all problems within the KernelBench Level 1 and Level 2 datasets. The performance of the LLM-generated kernels was benchmarked against several standard compilation baselines: Torch Eager, Torch Inductor Default, Torch Inductor Max-Autotune, and Torch Inductor Reduce-Overhead. All preliminary evaluations were conducted on an NVIDIA RTX 4070 laptop GPU.

### 4.1 Level 1 Evaluation

For the KernelBench level 1 dataset, we measured both the compilation success rate and the functional correctness rate of the generated CUDA code. o1-mini achieved a compilation rate of 39% and a correctness rate of 27%. In contrast, Gemini 2.5 Pro demonstrated significantly higher rates with 83% compilation success and 65% functional correctness.

Performance evaluation of the correctly generated kernels revealed stark differences between the models.

- **o1-mini**: The performance of kernels generated by o1-mini was generally poor compared to the baselines. It achieved a geometric mean speedup of only 0.3424 relative to Torch Eager, with a corresponding Fast-P score of 0.07 (using a speedup threshold of 1.0). While performing slightly better against the Inductor backends, it remained slower on

**Table 2: Level 1 Kernel Generation Failures by Problem Type**

| Problem Type | Total | o1-mini Failures | | Gemini 2.5 Pro Failures | |
|---|---|---|---|---|---|
| | Kernels | Count | Rate (%) | Count | Rate (%) |
| CONV | 34 | 34 | 100% | 16 | 47% |
| Norm | 8 | 8 | 100% | 5 | 63% |
| Loss | 7 | 7 | 100% | 3 | 43% |
| MatMul | 22 | 8 | 36% | 2 | 9% |
| Miscellaneous | 29 | 16 | 55% | 9 | 31% |
| Total | 100 | 73 | 73% | 35 | 35% |

average, achieving a geometric mean speedup of 0.7690 compared to Inductor Max-Autotune. Out of the 27 correctly generated kernels, only 5 (18.5%) managed to exceed the performance of all baseline compilers.

- **Gemini 2.5 Pro**: This model yielded substantially better performance results. While still slightly underperforming Torch Eager on average (geometric mean speedup of 0.8676, Fast-P score of 0.30), it matched or slightly exceeded the performance of Inductor default (geometric mean speedup of 1.0520). Furthermore, Gemini 2.5 Pro significantly outperformed Inductor Max-Autotune and Inductor Reduce-Overhead, achieving geometric mean speedups of 1.7951 and 1.7355, respectively. Out of the 65 correctly generated kernels, 43 (66%) met or exceeded the performance of at least one baseline compiler, and 24 (37%) surpassed the performance of all baseline compilers.

Table 2 details the breakdown of failures by problem type for the Level 1 dataset. Both models struggled significantly with CONV, Norm, and Loss problems. o1-mini failed on 100% of these specific categories, whereas Gemini 2.5 Pro showed higher success rates but still faced challenges, particularly with Norm problems (63% failure rate). Matrix Multiplication problems had the highest success rates for both models.

In summary for Level 1, o1-mini exhibited poor performance across compilation, correctness, and execution speed, with very few generated kernels proving to be competitive. Gemini 2.5 Pro performed considerably better, successfully generating correct code for a majority of kernels and often achieving performing exceeding the baseline compilers, particularly the Inductor backends.

## 4.2 Level 2 Evaluation

The Level 2 dataset, consisting of 100 more complex problems (typically fusion operators), presented greater challenges for both models. The compilation and correctness rates observed were:

- o1-mini: 48% compilation rate, 22% correctness rate.
- Gemini 2.5 Pro: 64% compilation rate, 39% correctness rate.

Compared to Level 1, o1-mini had a higher compilation rate but a lower overall correctness rate, while Gemini 2.5 Pro saw a decrease in both metrics.

Performance evaluation for the correctly generated Level 2 kernels showed the following trends:

- **o1-mini**: Performance remained weak. It was particularly poor against Inductor Default and Max-Autotune. Compared to Torch Eager, it achieved a geometric mean speedup of 0.6632 and a Fast-P score of 0.10 (speedup threshold of 1.0). Only 3 (13.6%) of the 22 correct kernels outperformed all baseline compilers.
- **Gemini 2.5 Pro**: The performance profile shifted slightly compared to Level 1. While showing a slight decrease in average speedup against Inductor Default and Max-Autotune, it demonstrated a substantial improvement relative to Torch Eager (geometric mean speedup of 1.3286, Fast-P score of 0.24) and Inductor Reduce-Overhead (geometric mean speedup of 1.3964, Fast-P score of 0.22). Of the 39 correct kernels generated by Gemini 2.5 Pro, 32 (82%) met or exceeded the performance of at least one baseline, and 11 (28%) surpassed all baselines. Notably, among the successful kernels, 5 out of the 9 MatMul kernels (55%) and 3 out of 8 GEMM kernels (38%) exceeded the performance of all baselines.

Table 3 provides a breakdown of generation failures for the Level 2 dataset. Similar to Level 1, CONV problems proved difficult, with high failure rates for both models. GEMM and MatMul problems also showed significant failure rates, particularly for o1-mini. The single BMM Norm problem failed for both models.

**Table 3: Level 2 Kernel Generation Failures by Problem Type**

| Problem Type | Total | o1-mini Failures | | Gemini 2.5 Pro Failures | |
|---|---|---|---|---|---|
| | Kernels | Count | Rate (%) | Count | Rate (%) |
| CONV | 63 | 47 | 75% | 41 | 65% |
| GEMM | 19 | 16 | 84% | 11 | 58% |
| MatMul | 17 | 15 | 88% | 8 | 47% |
| BMM Norm | 1 | 1 | 100% | 1 | 100% |
| Total | 100 | 78 | 78% | 61 | 61% |

For Level 2, o1-mini continued to struggle, achieving low correctness and poor performance even for the kernels it generated successfully. Gemini 2.5 Pro, while worse than the results achieved by Level 1, still performed well. A significant fraction (82%) of its successfully generated kernels were competitive with or superior to at least one baseline, and more than a quarter outperformed all baselines. The performance gains were particularly notable relative to Torch Eager and for matrix multiplication-based kernels (MatMul, GEMM).

Overall, these preliminary results highlight the varying capabilities of current LLMs for CUDA code generation. While Gemini 2.5 Pro shows promise, particularly in generating performant code that can sometimes exceed standard compiler optimizations, significant challenges remain in ensuring correctness and compilation success, especially for more complex kernel types and as problem complexity increases (Level 2 vs. Level 1). The o1-mini model appears less suited for this specific task based on these benchmarks.

# 5 Improved KernelBench Results and Evaluation

Due to time and resource limitation, we selected a number of representative problems from level 1 for evaluation, covering a diverse set of operations of Matmal, non-linear activation functions, and convolutions. Generation was executed with deepseek-coder, performance data was collected on NVIDIA T4 GPU. Table 4 details

the speedup the generated kernel gained from Torch Eager at the end of each stage for each problem, as well as comparison to the best speedup on KernelBench leaderboard.

Overall, for the majority of the problems, we are able to exceed the best speedup on KernelBench leaderboard. We have observed that iterative refinement allows the LLM to better generate a compiled and correct kernel. Our system still struggles to generate correct and efficient code for convolutions, but performs particularly well on problem 24 and 40, gaining a 1.89x and 2.06x speedup separately, while the best speedup on KernelBench leaderboard is 0.03x and 0.07x separately.

**Limitations and future work** We observed that the scale of the maximum samples and iterations allowed is a significant factor that affects the result of generation. We used a maximum of 5 samples and 5 iterations allowed for correctness, and a total of 24 samples allowed for optimization, with each sample taking at most 5 iterations. To support larger scale, we must draw samples and evaluate them in parallel with LLM ensembles.

It is difficult to give informative yet still concise correctness and performance feedback. We currently give only max difference as correctness feedback . An alternative is to give the full resulting matrix and reference matrix as correctness feedback, but the size could be massive and the values still uninformative. A human programmer may debug for correctness with selected input, which is another alternative that we may use.

While many techniques have been explored to optimize generated kernels, there aren't many proposed techniques to improve generating for correctness, except possibly giving few-shot examples. As we observed that LLMs struggle to produce correct code for convolution, a future work could be including few-shot examples for those hard problems.

# 6 Evolutionary Computing, Sakana AI Validation

## 6.1 Verifing Sakana AI's AI CUDA Engineer

Subsequent to the release of the paper and generated dataset, users found cases where the generated kernel returned the correct answer from memory exploits instead of correct computation, resulting in a false correctness and speedup result [9, 10]. Specifically, for level 1 problem 15 lower triangular matrix multiplication problem, a generated kernel with over 100x speedup performs only a standard memory allocation for the output tensor with `torch.empty_like()` and a no-op. `torch.empty_like()` allocates uninitialized memory; therefore, executing the PyTorch reference module before the custom kernel could result in intermediate answer leakage. In this particular problem, the PyTorch reference module computes a full matrix and returns a copy of the lower triangular part. The memory space containing the full computed matrix is subsequently reused and allocated to custom kernel's output tensor without initilization, allowing the custom kernel to simply forward the correct answer computed by PyTorch with a no-op.

Currently, problematic kernels that have come to public awareness have been taken down from Sakana AI's published dataset, and the official evaluation of the AI CUDA Engineer is still under review. In order to assess true effectiveness of Sakana AI's optimization approach, we verified selected top performing kernels from their published dataset.

**Methodology** Sakana AI's CUDA Engineer generated raw CUDA code without Python wrapper, therefore, those kernels are not compatible with KernelBench's benchmarking framework. We loaded the custom kernels using torch c++ loading utilities with `-O3` compilor flag, and PyTorch reference modules from code string with Python's built-in `compile()` and `exec()`. We tested correctness by comparing the output of custom kernel with output of PyTorch module on randomized inputs. The inputs are generated by functions provided by the reference PyTorch module, though in rare cases the sakana's functional version of PyTorch module has different input functions, in which case we use the input functions in the functional version for compatibility with their CUDA kernels. To avoid answer leakage through memory exploits, custom kernel's output is obtained and deep copied before PyTorch reference module is executed in each trial.

Due to time and resource restraints, we focused on top performing kernels in level 1. For each problem which Sakana claimed to have a speedup, we selected up to 3 kernels with the highest speedups. Correctness is verified against the original torch module in 3 random trials. If correct, then runtime of the custom kernel is collected using CUDA event after a sufficient number of warm up trials and compared against torch Eager baseline. The hardware used is a NVIDIA T4 GPU with 16G memory.


**Figure 6: Expected vs Actual Results for the kernels suggested by Sakana AI by their evolutionary computing. The results shown here is sampled for random 10 kernels chosen, full results can be seen in the link provided.**

**Results** We were able to verify 132 generated kernels across 44 problems that are claimed to be correct and faster than native PyTorch.[2] 10 kernels among those failed our correctness check. 37 kernels over 15 problems did not actually yield speedup in our verification, with 10 problems having no selected kernels that yielded a speedup. The remaining 83 kernels over 34 problems had speedup at least 1 from native PyTorch. For most kernels among them, our measured speedup is comparable to the reported speedup. Except one outlier problem that PyTorch baseline does not perform well

[2] https://github.com/Vineeth-Narayan/Improved_Kernel_Bench/blob/sakana_test/sakana_test/results/leve

**Table 4: Improved KernelBench Level 1 DeepSeek-Coder Results**

| Problem | Problem Description | Speedup after Correctness IR | Speedup after Optimization IR | KernelBench Leaderboard |
|---|---|---|---|---|
| p1 | Matmul | 0.17 | 0.22 | 0.17 |
| p21 | Sigmoid | 1.01 | 1.18 | 1.00 |
| p24 | LogSoftmax | 0.65 | 1.89 | 0.03 |
| p40 | LayerNorm | 0.01 | 2.06 | 0.07 |
| p51 | Argmax | 0.23 | 0.46 | 0.90 |
| p61 | Conv | Failed | Failed | Failed |
| p74 | Conv | 1.76 | 1.76 | Failed |

**Table 5: KernelBench Level 1 Gemini 2.5 Pro RTI(3,3,3) Results**

| Problem ID | IR Start (Initial) | IR End (Final) | Torch Eager | Inductor Default | Inductor Reduce-Overhead | Inductor Max-Autotune |
|---|---|---|---|---|---|---|
| p1 | 11.6 | 2.04 | 1.82 | 2.9 | 2.08 | 4.16 |
| p21 | 0.0867 | 0.0192 | 0.0136 | 0.0433 | 0.0976 | 0.121 |
| p24 | Failed Compiling | Failed Compiling | 0.0294 | 0.107 | 0.0864 | 0.0986 |
| p40 | Failed Compiling | Failed Compiling | 6.67 | 5.79 | 10.3 | 8.07 |
| p51 | 0.139 | 0.0552 | 0.0225 | 0.0498 | 0.08 | 0.0853 |
| p61 | 9.25 | 9.0 | 10.0 | 9.66 | 9.79 | 9.78 |
| p74 | Failed Correctness | 0.0971 | 0.137 | 0.0767 | 0.0771 | 0.0804 |

**Table 6: KernelBench Level 1 o1-mini RTI(3,3,3) Results**

| Problem ID | IR Start (Initial) | IR End (Final) | Torch Eager | Inductor Default | Inductor Reduce-Overhead | Inductor Max-Autotune |
|---|---|---|---|---|---|---|
| p1 | 6.79 | 2.56 | 1.8 | 1.85 | 1.15 | 1.18 |
| p21 | 0.664 | 0.0496 | 0.0376 | 0.121 | 0.213 | 0.204 |
| p24 | Failed Compiling | Failed Compiling | 0.0405 | 0.196 | 0.194 | 0.202 |
| p40 | Failed Compiling | 6.98 | 13.7 | 1.08 | 1.69 | 1.55 |
| p51 | 0.328 | 0.0918 | 0.0616 | 0.123 | 0.195 | 0.203 |
| p61 | Failed Compiling | Failed Correctness | 0.951 | 1.01 | 1.04 | 1.05 |
| p74 | Failed Correctness | Failed Correctness | 0.101 | 0.172 | 0.205 | 0.217 |

in, Sakana's dataset includes kernels with almost 9x speedup on problem 88 (Gelu), 7x speedup on problem 97 (cosine loss), and almost 5x speedup on problem 95 (cross entropy loss). In figure 6, we show results from 10 randomly selected kernels as we see that, Sakana AI is actually able to match its expected results except in few cases.

There are 20 other problems in level 1 with 50 selected kernels that we were not able to verify due to the mismatch between custom kernel's expected arguments and the provided input functions. Those problems unfortunately concentrate in convolution kernels. Due to time constraint, We decided to leave the problem of matching the correct function arguments automatically without a compatible input function for future work.

In conclusion, despite lacking rigor in reported evaluation, Sakana's framework yielded promising results. The false positive cases where the generated kernel passed correctness checks through cheating seem to make up only a very small portion of their pool of kernels. In comparison to KernelBench results, though the $\texttt{fast}_1$ score will likely be similar to deepseek-R1 with iterative refinement, Sakana is particularly good at optimizing specific kernels for very high speedups, possibly a benefit from LLM ensemble and crossover prompting that allows the LLM to combine various optimization strategies.

## 7 Discussion and Future Work

From the results of our validation of Sakana AI's kernels, we see that there is true utility in using iterative refinement along with optimisation algorithms like evolutionary computing. Though our results of iterative refinement is limited in its improvement, there are a lot of avenues for improvement via different optimisation techniques. Our results were bottlenecked by the amount of gpu compute resources we had available. As part of future work we want to include NVIDIA Nsight (NCU), NVIDIA's fine-grained hardware profiler which can further improve LLMs readability into the

issue for improvement by giving more in-depth profiling information. Additionally, given slow generation times, parallelizing the sampling tasks would go a long way in terms of speeding up kernel generation. Finally, exploration of optimisation techniques as a last step of iterative refinement, we believe is a great direction for future research as per our results.

## 8 Conclusion

CUDA Kernel generation is an important area of research given how tedious and difficult writing efficient kernels are. To this end, we evaluate and analyze CUDA kernels generated by state of the art LLM models for different types and complexities of kernels. Additionally, we improve on the Kernel Bench evaluation framework and opensource techniques like iterative refinement and temperature sampling. Finally we also validate the results given by Sakana AI and their evolutionary computing technique for kernel generation.

## 9 Acknowledgments

We thank Professor. Keshav Pingali, in whose class we had the opportunity to do this project and additionally the TACC systems whose GPUs were used for evaluation.

## References

[1] Sakana AI. 2025. The AI CUDA Engineer: Agentic CUDA Kernel Discovery, Optimization and Composition. https://sakana.ai/ai-cuda-engineer/#limitations-and-bloopers

[2] NVIDIA. 2025. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] NVIDIA. 2016. NVIDIA Tesla P100 The Most Advanced Datacenter Accelerator Ever Built. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[4] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[5] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? arXiv:2502.10517 [cs.LG] https://arxiv.org/abs/2502.10517

[6] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. 2024. ThunderKittens: Simple, Fast, and Adorable AI Kernels. arXiv:2410.20399 [cs.LG] https://arxiv.org/abs/2410.20399

[7] Philippe Tillet, H. T. Kung, and David Cox. 2019. 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* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[8] UIUC and NVIDIA. 2025. Lecture slides, GPU Teaching Kit, University of Illinois Urbana-Champaign. http://gputeachingkit.hwu.crhc.illinois.edu/

[9] User "main_horse". 2025. This example from their paper (...), which is claimed to have 150x speedup, is actually 3x slower if you bench it... https://x.com/main_horse/status/1892446384910987718. Tweet.

[10] User "miru_why". 2025. turns out the AI CUDA Engineer achieved 100x speedup by… hacking the eval script. https://x.com/miru_why/status/1892500715857473777. Tweet.

Received 2 May 2025

**Table 7: KernelBench Level 1 o1-mini Fast-P Scores vs Baselines (percentage)**

| Metric | Torch Eager | Inductor Default | Inductor Max-Autotune | Inductor Reduce-Overhead |
|---|---|---|---|---|
| Speedup Geometric Mean | 0.3424 | 0.3900 | 0.7690 | 0.7492 |
| Speedup Threshold (0) | 0.27 | 0.27 | 0.27 | 0.27 |
| Speedup Threshold (0.5) | 0.10 | 0.15 | 0.16 | 0.16 |
| Speedup Threshold (0.8) | 0.07 | 0.09 | 0.15 | 0.15 |
| Speedup Threshold (1) | 0.07 | 0.06 | 0.14 | 0.14 |
| Speedup Threshold (1.5) | 0.03 | 0.02 | 0.12 | 0.10 |
| Speedup Threshold (2) | 0.01 | 0 | 0.09 | 0.08 |

**Table 8: KernelBench Level 1 Gemini 2.5 Pro Fast-P Scores vs Baselines (percentage)**

| Metric | Torch Eager | Inductor Default | Inductor Max-Autotune | Inductor Reduce-Overhead |
|---|---|---|---|---|
| Speedup Geometric Mean | 0.8676 | 1.0520 | 1.7951 | 1.7355 |
| Speedup Threshold (0) | 0.65 | 0.65 | 0.65 | 0.65 |
| Speedup Threshold (0.5) | 0.43 | 0.43 | 0.44 | 0.43 |
| Speedup Threshold (0.8) | 0.35 | 0.39 | 0.41 | 0.41 |
| Speedup Threshold (1) | 0.30 | 0.37 | 0.41 | 0.40 |
| Speedup Threshold (1.5) | 0.26 | 0.29 | 0.38 | 0.37 |
| Speedup Threshold (2) | 0.20 | 0.24 | 0.32 | 0.34 |

**Table 9: KernelBench Level 2 o1-mini Fast-P Scores vs Baselines (percentage)**

| Metric | Torch Eager | Inductor Default | Inductor Max-Autotune | Inductor Reduce-Overhead |
|---|---|---|---|---|
| Speedup Geometric Mean | 0.6632 | 0.5070 | 0.4977 | 0.7540 |
| Speedup Threshold (0) | 0.22 | 0.22 | 0.22 | 0.22 |
| Speedup Threshold (0.5) | 0.15 | 0.13 | 0.13 | 0.13 |
| Speedup Threshold (0.8) | 0.13 | 0.10 | 0.07 | 0.11 |
| Speedup Threshold (1) | 0.10 | 0.08 | 0.07 | 0.09 |
| Speedup Threshold (1.5) | 0.08 | 0.06 | 0.06 | 0.08 |
| Speedup Threshold (2) | 0.05 | 0.05 | 0.04 | 0.07 |

**Table 10: KernelBench Level 2 Gemini 2.5 Pro Fast-P Scores vs Baselines (percentage)**

| Metric | Torch Eager | Inductor Default | Inductor Max-Autotune | Inductor Reduce-Overhead |
|---|---|---|---|---|
| Speedup Geometric Mean | 1.3286 | 0.8694 | 0.9438 | 1.3964 |
| Speedup Threshold (0) | 0.39 | 0.39 | 0.39 | 0.39 |
| Speedup Threshold (0.5) | 0.28 | 0.25 | 0.28 | 0.27 |
| Speedup Threshold (0.8) | 0.25 | 0.20 | 0.21 | 0.25 |
| Speedup Threshold (1) | 0.24 | 0.18 | 0.18 | 0.22 |
| Speedup Threshold (1.5) | 0.18 | 0.12 | 0.14 | 0.18 |
| Speedup Threshold (2) | 0.15 | 0.12 | 0.12 | 0.17 |

**Table 11: KernelBench Level 1 Gemini 2.5 Pro Total-Wins (runtime values in milliseconds)**

| Problem | o1-mini | Gemini 2.5 Pro | Torch Eager | Inductor Default | Inductor Reduce Overhead | Inductor Max Autotune |
|---|---|---|---|---|---|---|
| 4_Matrix_vector_multiplication_.py | 14.6 | 0.571 | 0.632 | 0.621 | 1.81 | 2.06 |
| 12_Matmul_with_diagonal_matrices_.py | | 1.06 | 12.8 | 13.1 | 13.5 | 13.3 |
| 20_LeakyReLU.py | 0.0405 | 0.0291 | 0.0414 | 0.0374 | 0.0735 | 0.112 |
| 22_Tanh.py | 0.0181 | 0.0078 | 0.0318 | 0.0336 | 0.0963 | 0.0812 |
| 23_Softmax.py | | 0.0288 | 0.0441 | 0.0909 | 0.0786 | 0.095 |
| 25_Swish.py | | 0.0153 | 0.0352 | 0.0531 | 0.0783 | 0.0948 |
| 26_GELU_.py | 0.0216 | 0.0143 | 0.0217 | 0.0357 | 0.0859 | 0.0736 |
| 27_SELU_.py | 0.053 | 0.0142 | 0.0197 | 0.0577 | 0.105 | 0.0807 |
| 29_Softplus.py | | 0.0212 | 0.0311 | 0.0604 | 0.114 | 0.0925 |
| 35_GroupNorm_.py | | 3.5 | 3.69 | 3.82 | 6.02 | 6.22 |
| 39_L2Norm_.py | | 0.0165 | 0.0444 | 0.135 | 0.0938 | 0.102 |
| 42_Max_Pooling_2D.py | | 0.19 | 0.265 | 0.292 | 0.619 | 0.652 |
| 43_Max_Pooling_3D.py | | 2.77 | 2.94 | 3.3 | 7.78 | 9.66 |
| 44_Average_Pooling_1D.py | | 0.0237 | 0.0383 | 0.0571 | 0.103 | 0.068 |
| 46_Average_Pooling_3D.py | | 2.51 | 2.59 | 2.64 | 8.76 | 7.75 |
| 49_Max_reduction_over_a_dimension.py | 0.0375 | 0.0163 | 0.0208 | 0.0375 | 0.0789 | 0.0753 |
| 53_Min_reduction_over_a_dimension.py | | 0.0196 | 0.0248 | 0.0354 | 0.117 | 0.0938 |
| 58_conv_transposed_3D__asymmetric_... | | 9.55 | 10.1 | 9.81 | 11.8 | 10.3 |
| 68_conv_transposed_3D__square_input... | | 148.0 | 150.0 | 156.0 | 367.0 | 501.0 |
| 74_conv_transposed_1D_dilated.py | | 0.0718 | 0.137 | 0.0767 | 0.0771 | 0.0804 |
| 76_conv_standard_1D_dilated_stride... | | 0.0428 | 0.0436 | 0.0788 | 0.0699 | 0.105 |
| 94_MSELoss.py | | 0.0419 | 0.161 | 0.0919 | 0.118 | 0.0999 |
| 97_CosineSimilarityLoss.py | | 0.0348 | 0.116 | 0.0523 | 0.125 | 0.109 |
| 100_HingeLoss.py | | 0.035 | 0.0569 | 0.0557 | 0.0819 | 0.0844 |

**Table 12: KernelBench Level 2 Gemini 2.5 Pro Total-Wins (runtime values in milliseconds)**

| Problem | o1-mini | Gemini 2.5 Pro | Torch Eager | Inductor Default | Inductor Reduce Overhead | Inductor Max Autotune |
|---|---|---|---|---|---|---|
| 9_Matmul_Subtract_Multiply_ReLU.py | 0.0557 | 0.0464 | 0.183 | 0.0623 | 0.112 | 0.0936 |
| 18_Matmul_Sum_Max_AvgPool_LogSumExp... | | 0.0322 | 0.212 | 0.0403 | 0.11 | 0.0356 |
| 29_Matmul_Mish_Mish.py | | 0.0729 | 0.0767 | 0.0811 | 0.107 | 0.136 |
| 33_Gemm_Scale_BatchNorm.py | | 0.0909 | 0.191 | 0.247 | 0.247 | 0.103 |
| 50_ConvTranspose3d_Scaling_AvgPool_... | 67.0 | 55.2 | 66.5 | 55.6 | 56.3 | 56.2 |
| 56_Matmul_Sigmoid_Sum.py | | 0.0208 | 0.0795 | 0.118 | 0.0843 | 0.167 |
| 57_Conv2d_ReLU_HardSwish.py | 0.134 | 0.112 | 0.167 | 0.138 | 0.43 | 0.231 |
| 68_Matmul_Min_Subtract.py | | 0.0214 | 0.0924 | 0.0565 | 0.0883 | 0.0673 |
| 70_Gemm_Sigmoid_Scaling_ResidualAdd.py | 0.0501 | 0.0501 | 0.0698 | 0.0767 | 0.243 | 0.0861 |
| 80_Gemm_Max_Subtract_GELU.py | | 0.0668 | 0.0732 | 0.0833 | 0.0804 | 0.0989 |
| 96_ConvTranspose3d_Multiply_Max_Glob... | | 64.7 | 65.8 | 65.8 | 73.0 | 70.2 |