AI Coding Agents for Hardware-Optimized Code

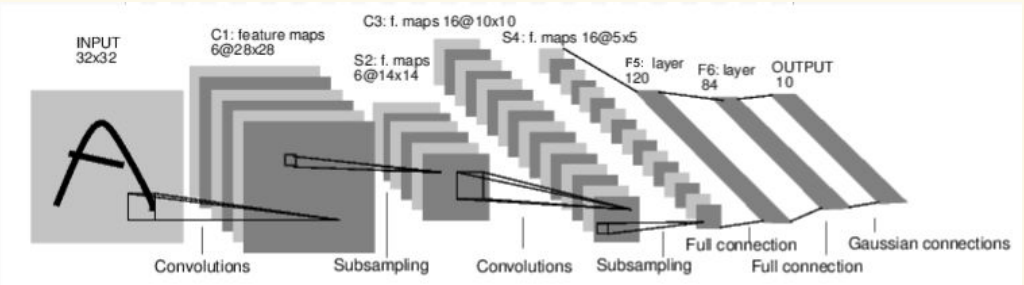
PIDS: Pranoy Dutta, Yiran (Irene) Wang, Dhairya Gupta, Shashank Iswara

Background - software for GPU’s

- Hardware is fast!
- Practical speedups require a robust software stack
- Writing custom kernels is hard!
- We have a talent shortage

Background - ML Workloads

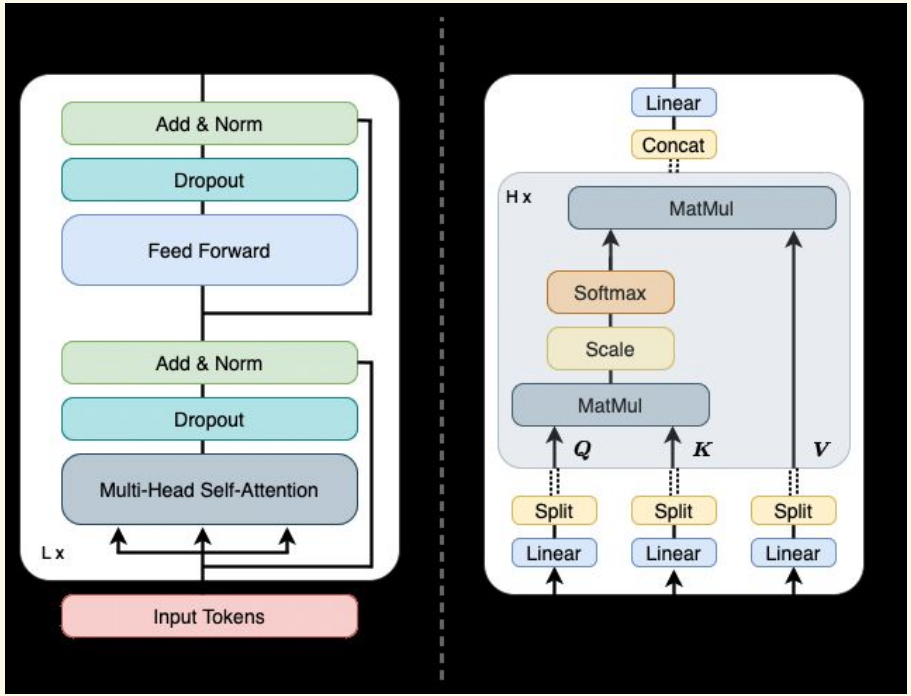
- ML Workloads:
  - Compute intensive
  - Memory intensive
  - Highly parallelizable
- Each operation = Separate GPU kernel
  - Matrix multiply, pooling, relu...
  - Kernel launch overhead
  - Memory overhead
- We might want to fuse operators, replace multiple operators with one, or make algorithmic changes



Simple Convolutional Neural Network for letter recognition

```
def forward(self, input):
    c1 = F.relu(self.conv1(input))
    s2 = F.max_pool2d(c1, (2, 2))
    c3 = F.relu(self.conv2(s2))
    s4 = F.max_pool2d(c3, 2)
    s4 = torch.flatten(s4, 1)
    f5 = F.relu(self.fc1(s4))
    f6 = F.relu(self.fc2(f5))
    output = self.fc3(f6)
    return output
```

Background - ML Scale



Transformer encoder layer architecture (left) and schematic overview of a multi-head self-attention block (right). Input tokens go through L encoder layers and H self-attention heads.

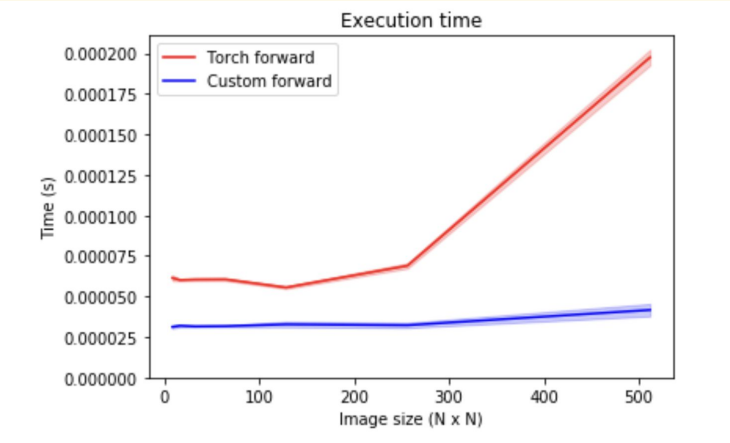
Background - How do we currently optimize kernels

- **1 - Naive PyTorch**
  - Greedily launches a kernel for each PyTorch operator
- **2 - torch.compile**
  - New functionality built into PyTorch that can implement some basic speedups by creating an execution graph
- **3 - Tools like Numba or Triton**
  - Can achieve better performance than the naive PyTorch and torch.compile
  - Requires programmers to learn new complicated frameworks
- **4 - Fully Custom Kernels**
  - Provides best potential for speedups
  - Requires specialized knowledge
  - Code is less likely to be robust

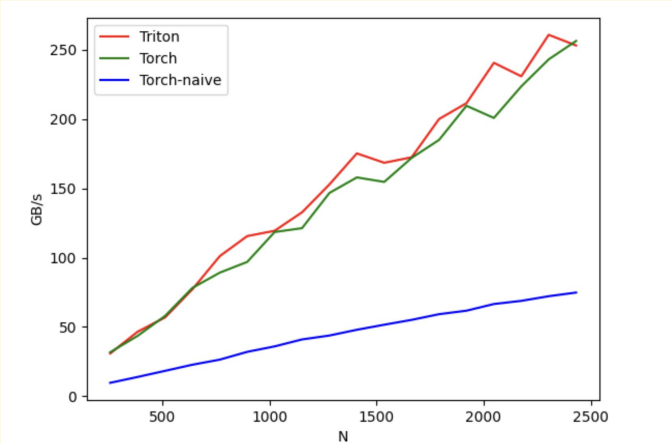
So why bother with writing custom kernels?

- Anecdotal evidence of ML researchers having who need to spend hours or days optimizing their architectures to make them performant
- ML researchers often have to rapidly test many different architectures, so there is not time to write custom kernels for each iteration

Torch vs Custom Kernels



Torch vs Triton Kernels



So why bother with writing custom kernels?

- Porting algorithms across platforms is a pain point
- The initial FlashAttention kernel was 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

Problem Statement

How can we better utilize LLMs to generate custom CUDA kernels?

Relevant Papers

KernelBench: Can LLMs Write Efficient GPU Kernels?

KernelBench is an open source testing framework created by researchers at Stanford and Princeton. It aims to test the skill of LLMs in generating and optimizing GPU kernels.

gpucc: An Open-Source GPGPU Compiler

gpucc is an open-source, LLVM-based compiler designed to efficiently translate CUDA and GPGPU code into optimized machine code for GPU architectures, offering a modular alternative to proprietary GPU compilers like NVCC.

Baldur: Whole-Proof Generation and Repair with Large Language Models

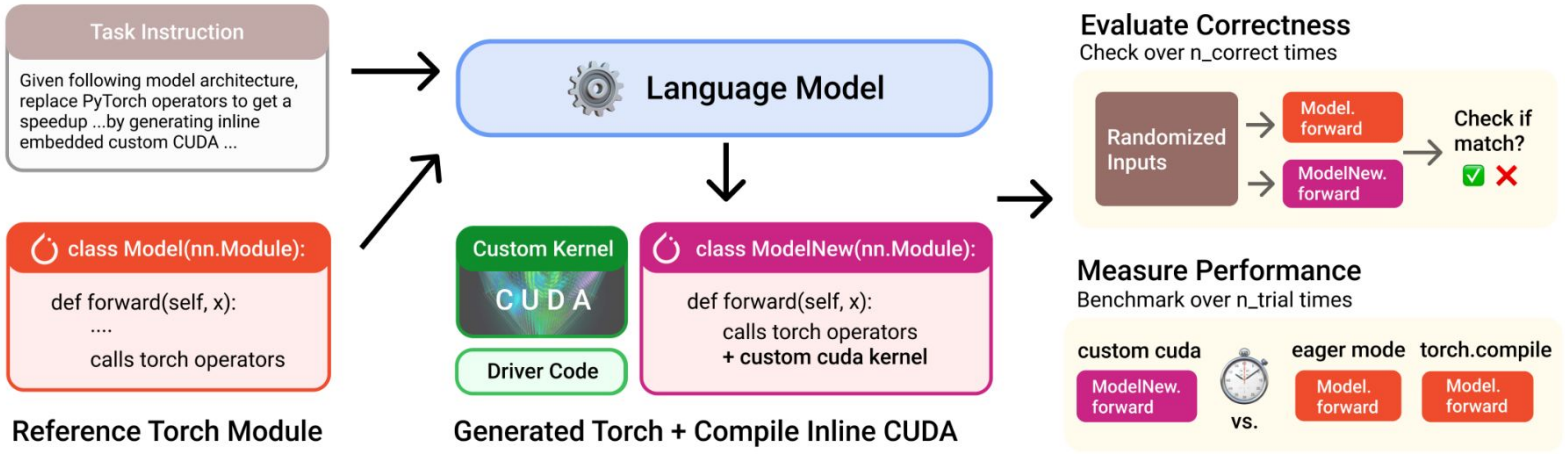
Baldur is a system for generating proofs of theorems in a single-shot. LLM-generated code is untrustworthy but if they generate a proof, we can machine-check it.

Kernel Bench

So what does KernelBench do?

- If these custom kernels are needed for ML workloads but they are hard for us to write…
- What’s our solution to everything in 2025?
- **LLMs**
- KernelBench creates a framework for and tests all the leading LLMs at the task of generating custom CUDA kernels for PyTorch modules

Method Overview



LLM Tasks Inputs

- **Reference implementation in PyTorch**
  - follows a typical AI researcher's workflow: define a Model class derived from torch.nn.Module
  - The Model includes __init__(), forward(), and relevant helper functions using PyTorch operations.
  - Specifies input sizes
- **Natural Language prompt**

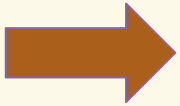
LLM Sample Tasks Inputs

```python
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Simple model that performs a single matrix multiplication (C = A * B)
        ↪ 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
```

```
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 ).
only limited by your imagination .

Here \'s an example to show you the syntax of inline embedding custom CUDA operators in
torch :
<EXAMPLE PyTorch Module>
<EXAMPLE Optimzed CUDA Kernel>
<INPUT PyTorch Module>

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 !
```

LLM Tasks Outputs

```python
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Simple model that performs a single matrix multiplication (C = A * B)
        ↪ 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
```



```
# 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;
}
"""
```

Task Levels

- **Level 1 (100 tasks): Single primitive operation**
  - Convolutions, matrix multiplication, loss computations
  - Little scope for speed ups because PyTorch already calls well optimized cuBLAS kernels
  - These kernels are closed source
- **Level 2 (100 tasks): Operator sequences**
  - Combine primitive operations
    - a combination of a convolution, ReLU, and bias
  - More scope but still limited scope for speedups because PyTorch compiler is usually good at identifying operator fusion
- **Level 3 (50 tasks): Full ML architectures**
  - These tasks are full model architectures
  - We would hope that the LLM has the best scope on these tasks

Verification

- Deciding whether or not something is equivalent is undecidable for traditional algorithmic methods (formally)
  - Because the Halting Problem exists, we can’t test equivalence either
- Approximating correctness with randomized inputs is most practical
  - Best for AI kernels because control flow is simpler, numerical correctness is important
- For more complex kernels, there should be a higher threshold for randomized input testing
  - Fuzzing could be a possibility

Metrics

- Two main ways of measuring generated Kernels:
  - Correctness
  - Performance (Speed compared to original kernel)
- Generate X (5) sets of inputs for the kernels and ensure they match
- These metrics are on *each problem*. To effectively compare outputs, we need to unify them.
- Unifying Metric: *fast-p*
  - Ratio of Correctness & Speedup past a threshold *p* (fast-0 is just correctness)

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

Results

- In order to get a full-view of model performance, results are kept for multiple passes
- Main metric is one-shot (pass@1)
- Error analysis
- Multi-pass with additional context could help

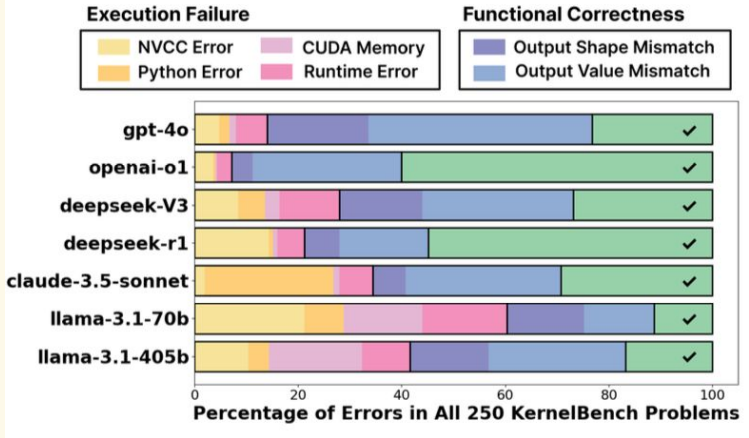
One-Shot Baseline

- Using the NVIDIA L40S GPU for all Profiling
- PyTorch Eager is the default execution mode (run immediately as the interpreter reads them)
- Torch.compile is the optimized approach by generating a graph
  - Similar to content in this class! Does things like operator fusion, etc.
- LLMs are *not* great (yet)

- Reasoning models make less execution failures, correctness is usually similar

fast-1 score (compared to Pytorch Eager and torch.compile)

$\mathbf{fast}_1$ 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%

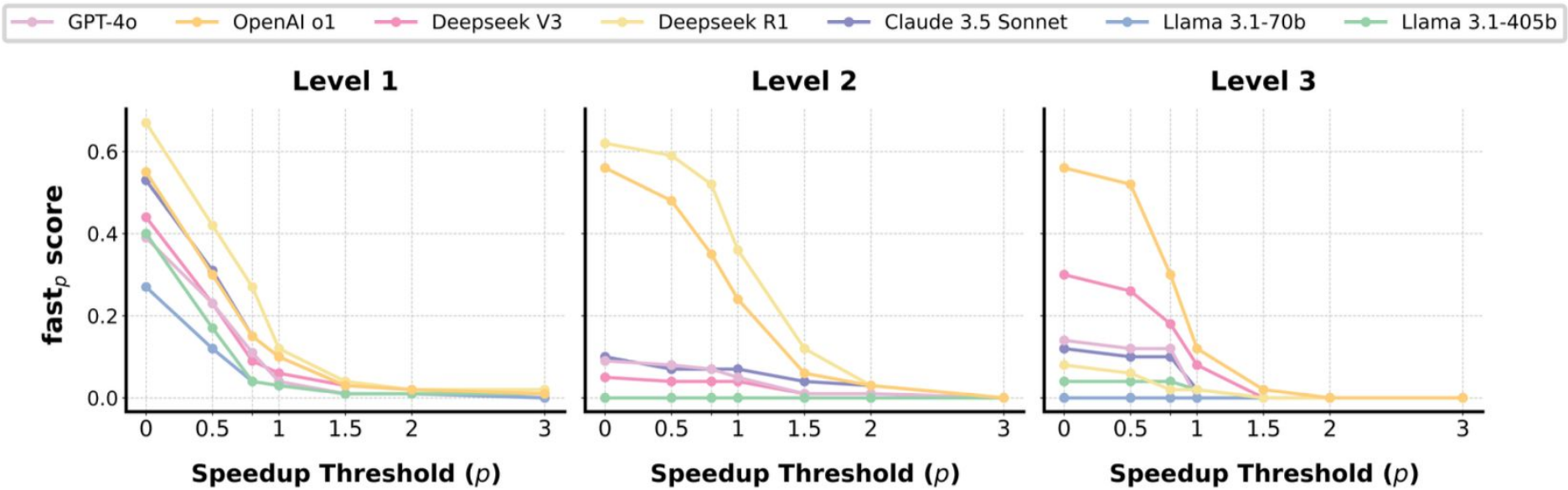


More one-shot performance

Key areas for improvement:

- Agentic processing with live feedback
- Tool-calling to test validity during inference time

These are compared to PyTorch Eager



Generalizing across Hardware

- Code generated is meant to be run across different architectures (without specific architectural details)
- Some models have high variation even for the same problems

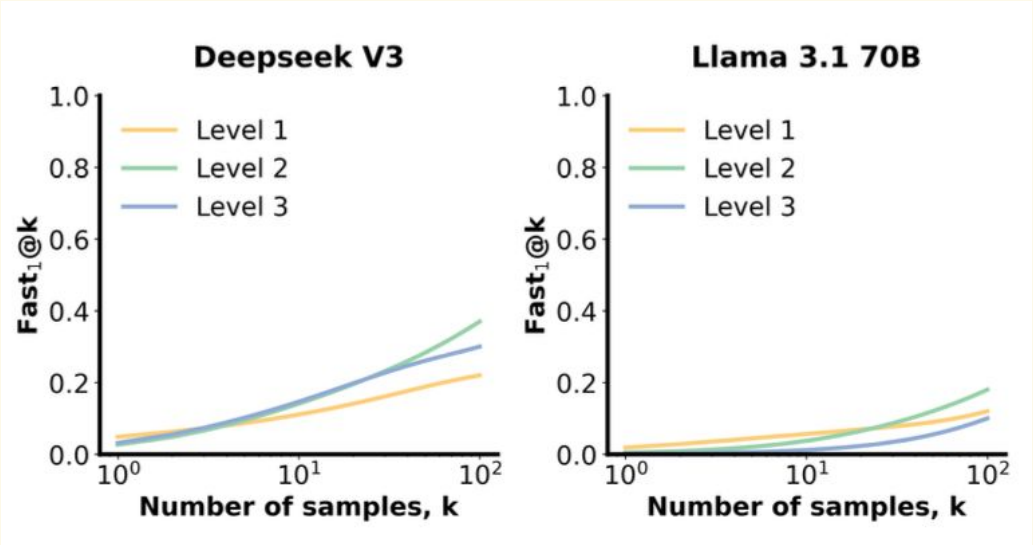
- **Key areas for improvement:**
  - Using SOTA GPU architectures
  - Testing with Tensor Processing Chips (Google TPU, Groq, etc.)
  - Giving specific architectural information at generation time

fast-p with p = 1

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%

Multiple Passes

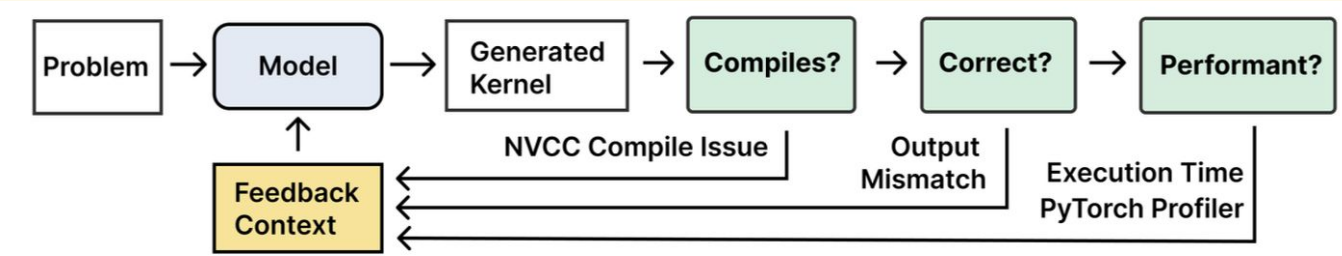
- Models will generate different results for the same inputs, testing to see how many samples we need for each model
- **fast-p@k** (fast-p at k samples)

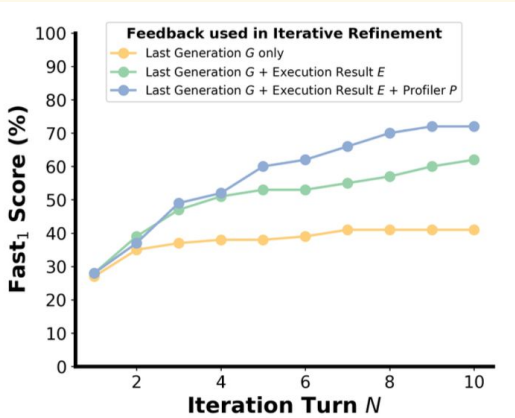


Refinement and In-Context Training

- With "*vibe coding*" becoming popular, people often go back and forth with models
- Need a way to test this iterative refinement
  - **fast-p@N**

- **fast-p@N** measures fast-p at N "turns"
  - Turns are essentially iterations of code generation





Comparison of Techniques

"**G**" is previous generation - (new context window if not included)
"**E**" is the execution result
"**P**" is the timing profile

Area to explore: attaching context before the model runs– kernel's GPU performance, etc.

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%**

Insights for Future Works

- A key area to visit is verification
  - Approximating correctness is not ideal for industry-grade software
  - Using tools like PyProf to characterize kernel

- For us: visiting new SOTA models that utilize mixed thinking
- Approaching with tool calling
- Convergence rate (how many iterations it takes for the LM to reach a working solution with a threshold of correctness)
- Utilizing existing compiler tools like Triton
- More sophisticated prompt engineering

Leaderboard

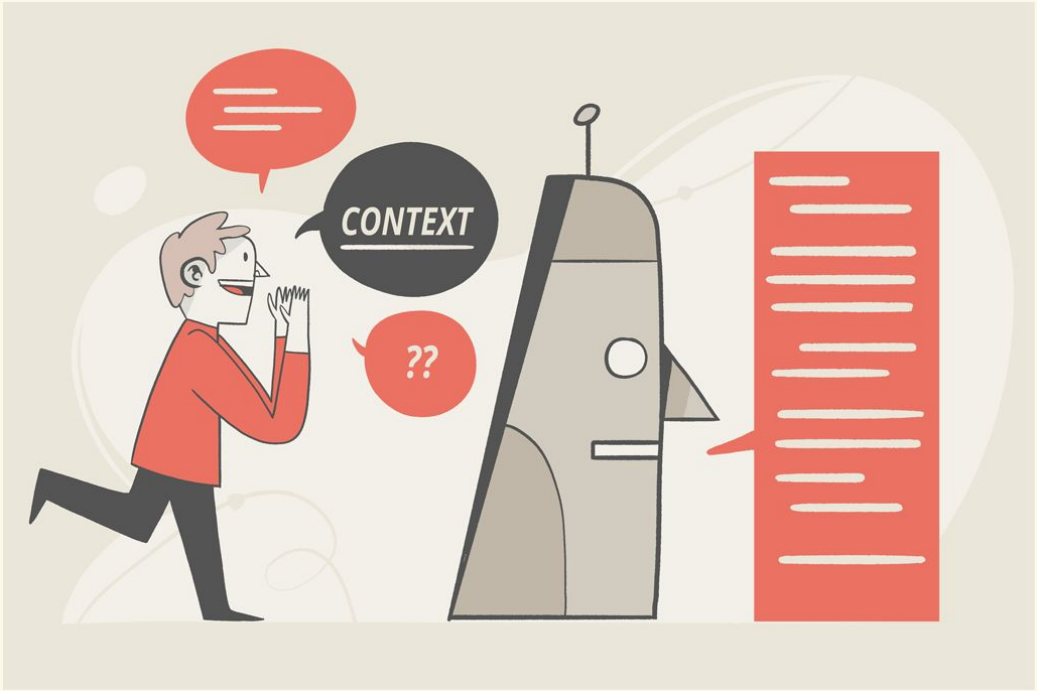
We need ways to fill this in

Level 2: 1_Conv2D_ReLU_BiasAdd					
Level 2: 2_ConvTranspose2d_BiasAdd_Clamp_Scaling_Cl…	1.10 (gpt-4o)				
Level 2: 3_ConvTranspose3d_Sum_LayerNorm_AvgPool_GE…	1.02 (gpt-o1)				
Level 2: 4_Conv2d_Mish_Mish	0.79 (gpt-o1)	0.66 (deepseek-coder)			
Level 2: 5_ConvTranspose2d_Subtract_Tanh	0.96 (gpt-o1)				
Level 2: 6_Conv3d_Softmax_MaxPool_MaxPool					
Level 2: 7_Conv3d_ReLU_LeakyReLU_GELU_Sigmoid_BiasA…	2.33 (claude-3.5-sonnet)	2.14 (gpt-o1)			
Level 2: 8_Conv3d_Divide_Max_GlobalAvgPool_BiasAdd_…	1.33 (gpt-4o)	1.03 (gpt-o1)			
Level 2: 9_Matmul_Subtract_Multiply_ReLU	1.89 (gemini-1.5-flash)	0.82 (gpt-o1)	0.78 (deepseek-coder)		
Level 2: 10_ConvTranspose2d_MaxPool_Hardtanh_Mean_T…	0.98 (claude-3.5-sonnet)	0.92 (gpt-o1)	0.88 (deepseek-coder)		
Level 2: 11_ConvTranspose2d_BatchNorm_Tanh_MaxPool_…					
Level 2: 12_Gemm_Multiply_LeakyReLU					
Level 2: 13_ConvTranspose3d_Mean_Add_Softmax_Tanh_S…	1.02 (claude-3.5-sonnet)				
Level 2: 14_Gemm_Divide_Sum_Scaling	3.17 (claude-3.5-sonnet)	2.66 (gpt-4o)	1.33 (gpt-o1)		
Level 2: 15_ConvTranspose3d_BatchNorm_Subtract					
Level 2: 16_ConvTranspose2d_Mish_Add_Hardtanh_Scali…	1.13 (gpt-o1)	1.12 (claude-3.5-sonnet)	1.05 (gpt-4o)	0.80 (deepseek-coder)	

Prompt Engineering

- KernelBench techniques for generating code use very little context

- Using Retrieval Augmented Generation (RAG) for code docs and optimization techniques
- Providing broad information on the purpose of the code being generated (i.e. including more information about data types, SPMM, etc.)



Triton

- Pythonic programming language and compiler for writing kernels
- Triton compiles down to PTX (same as CUDA)
- Why is Triton preferred:
  - Eliminates the boilerplate CUDA syntax
  - Focus on algorithmic implantation
  - Includes simple optimizations like operator fusion and other techniques to reduce the number of kernel launches
  - LLMs often fail due to syntax errors

gpucc: An Open-Source GPGPU Compiler

Jingyue Wu Artem Belevich Eli Bendersky Mark Heffernan Chris Leary
Jacques Pienaar Bjarke Roune Rob Springer Xuetian Weng Robert Hundt

Google, Inc., USA

{jingyue,tra,eliben,meheff,leary,jpienaar,broune,rspringer,xweng,rhundt}@google.com

NVIDIA CUDA Compiler Driver: NVCC

CUDA code (C++ code extended with CUDA-specific syntax)

Parallel Thread Execution intermediate code for NVIDIA GPUs

GPU-specific binary executable code (e.g. *.cubin* or embedded GPU code)

Motivation: Problems with NVCC

- **Closed-source design:** hinders understanding of internal optimizations, delays bug resolution, and limits visibility into language features.
- **Limited customization/innovations**: developers and researchers cannot easily experiment with or modify compilation passes
- **Monolithic architecture:** difficult to modularize or extend (e.g. integration with modern toolchains like Clang and LLD)

Solution: GPUCC

- First fully-functional open-source CUDA compiler
- Built on LLVM, providing modern infrastructure and modular design.
- Full support for C++11 and C++14, with improved handling compared to NVCC.
- Includes both general-purpose and CUDA-specific LLVM passes for performance tuning.
- Leverages common code paths for CPU and GPU, enabling cross-architecture optimizations.
- Deployed internally at Google for large-scale CPU and GPU compilation.

Mixed-Mode CUDA code

```
template <int N>
void host(float *x) {
  float *y;
  cudaMalloc(&y, 4*N);
  cudaMemcpy(y, x, ...);
  kernel<N><<<16, 128>>>(y);
  ...
}
```

```
template <int N>
__global__ void kernel(
    float *y) {
  ...
}
```

CPU/host

GPU/device

Mixed-Mode CUDA code Difficulties

- One code path must be emitted twice with different semantics
- Symbol linking issues across host/device
- Template instantiation needs consistent qualifier handling and overload resolution
- C++ features valid on CPU but not GPU can cause bugs
- Memory synchronization need to contain at runtime

Mixed-Mode CUDA code Solution

The CUDA code has functions marked with CUDA-specific attributes like:

- `__global__` → kernel (device code, launched by host)
- `__device__` → device-only functions (can be called from kernels)
- `__host__` → host-only (optional, default for normal functions)

Clang recognizes these and builds **separate ASTs** (abstract syntax trees) for host and device functions

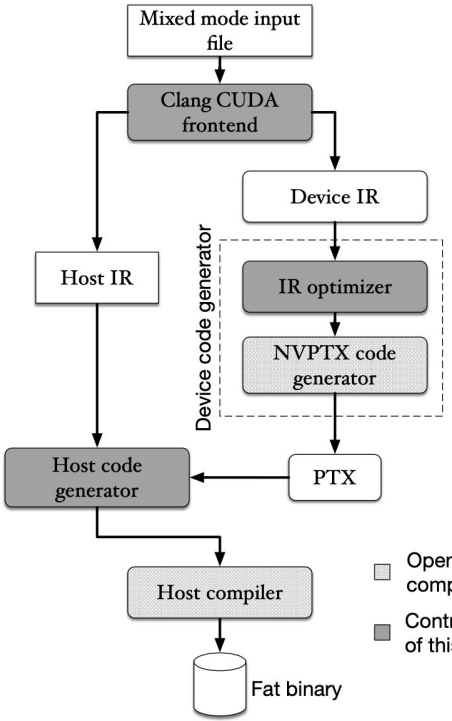


Figure 1: Overview of the compilation flow and execution of a mixed mode input file containing CPU and GPU code.

Major Optimizations in gpucc

- Straight-line scalar optimizations
- Inferring memory spaces
- Loop unrolling and function inlining
- Memory-space alias analysis
- Speculative execution
- Bypassing 64-bit divisions

Memory space	Attribute	Type qualifier
shared	`.shared`	`__shared__`
global	`.global`	`__global__`
local	`.local`	N/A
constant	`.const`	`__constant__`

Table 2: *Representation of memory spaces in PTX and CUDA.* Column **attribute** shows the memory space attributes PTX's `ld` and `st` instructions can optionally take. Column **type qualifier** shows the CUDA type qualifiers that programmers can use to annotate a variable residing in each memory space.

Insight

Our Project

LLM generated DSL
↓
DSL to CUDA compiler
↓
CUDA to binary compiler (NVCC)

Takeaway from GPUCC

General knowledge background on GPU compilation and optimization problem

Separating problems into modular subproblems

Custom optimizations to individual subproblems

Solution: GPUCC

- the first fully-functional, open-source, high performance CUDA compiler
- based on LLVM and supports C++11 and C++14
- developed and tuned several general and CUDA-specific optimization passes

Results highlight (compared with nvcc)

- up to 51% faster on internal end-to-end benchmarks
- on par on open-source benchmarks
- compile time is 8% faster on average and 2.4x faster for pathological compilations

Baldur: Whole-Proof Generation and Repair with Large Language Models

Outline

1. Motivation (Problems with LLM Codegen)
2. Current approaches to automated verification
3. Baldur
    a. Full proof generation
    b. Proof repair
4. Evaluation

Problems with LLM Code Gen.

- There are many tools to generate code with AI
  - Cursor, Windsurf, Claude Code, etc.
- Lots of code will be generated by AI in the future.
- How can we trust it?

Formal Verification.

New problem: Formal verification is hard.

- Manual proof writing is time consuming, expensive, and difficult.
  - Proof of the CompCert C compiler is 3x the length of the actual code.

Current approaches to automated verification.

- “Hammers”
  - Iteratively apply known math facts via heuristics.
  - Has trouble with inductive reasoning.
- Search-based Neural theorem provers
  - Given partial proof + current proof state (goal, list of assumptions), predict next line of proof
  - Computationally expensive search

Baldur's contributions

- Fine-tune LLMs on proofs, and generate whole proofs in a single shot.
- Proof repair task – when given proof assistant error message, LLMs perform better at fixing invalid proof.
- Performs better than search-based approaches with lower computational cost.

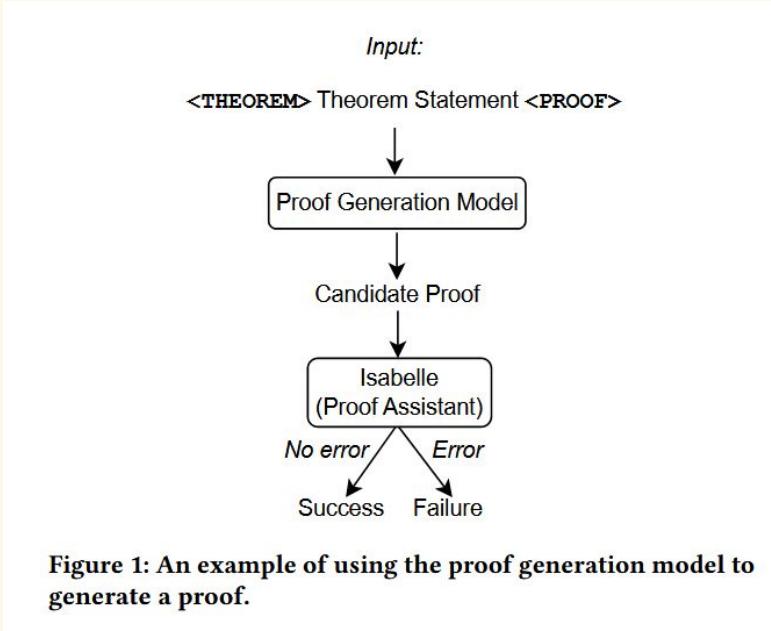


Figure 1: An example of using the proof generation model to generate a proof.

Proof Generation

- Given theorem statement, generate a full candidate proof.
- If Isabelle (proof assistant) accepts proof, it is correct, else give it back to the LLM in proof repair mode.

Training data for fine-tuning

- Create a new dataset which matches theorem statements to whole proofs
  - Takes existing neural theorem prover datasets and concatenates proof steps into single proof.

```
lemma fun_sum_commute:
  assumes "f 0 = 0" and "∧x y. f (x + y) = f x + f y"
  shows "f (sum g A) = (∑a∈A. f (g a))"
```

```
by (induct A rule: infinite_finite_induct)
(simp_all add: assms)
```

Proof Repair

Input: theorem statement, incorrect proof, error message
Output: candidate proof

```
lemma fun_sum_commute:
  assumes "f 0 = 0" and "∧x y. f (x + y) = f x + f y"
  shows "f (sum g A) = (Σa∈A. f (g a))"
```

```
proof (induct A)
case (insert x A)
thus ?case
  by (simp add: assms(2))
qed simp
```

```
Step error: Unable to figure out induct rule
At command "proof" (line 1)
```

```
by (induct A rule: infinite_finite_induct)
(simp_all add: assms)
```

Creating proof repair dataset.

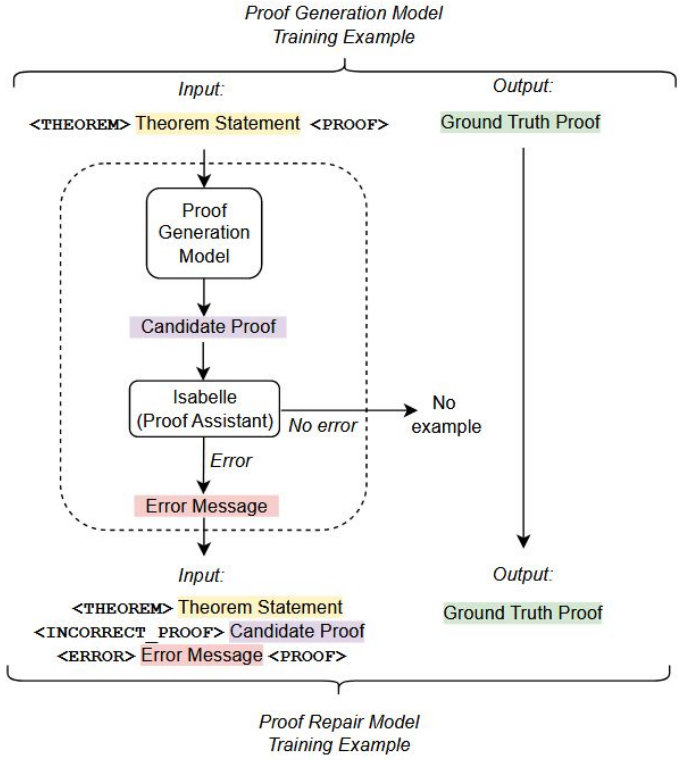


Figure 3: Training data creation for the proof repair model.

Results / Eval

Experiment setup

LLM (pretty outdated, 2023)

- Minerva, based on PaLM, pretrained on math.
- 8 billion, 62 billion parameter models.
- Max context length of 2048
- Fine-tuning: batch size of 32, steps: 100,000.
- Model overfits at 50,000 to 70,000 steps.

Machine

- Training: 64 TPUv3 cores, across 8 hosts, for 62B model 256 TPUv3 across 32 hosts.
- Inference: 32 inference servers each with 8 TPUv3 cores.

PISA dataset

- Isabelle/HOL
- Test set: 3,000 randomly chosen theorems

How effective are LLMs at generating whole proofs?

PISA benchmark (what % of theorems can they fully generate proofs for?)

- Sledgehammer: 25.6%
- Search-based: 39.0%
- Baldur-8B (16 samples): 34.8%
- Baldur-8B (64 samples): 40.7%
- Baldur-62b w/ context: 47.9%

Can LLMs be used to repair proofs?

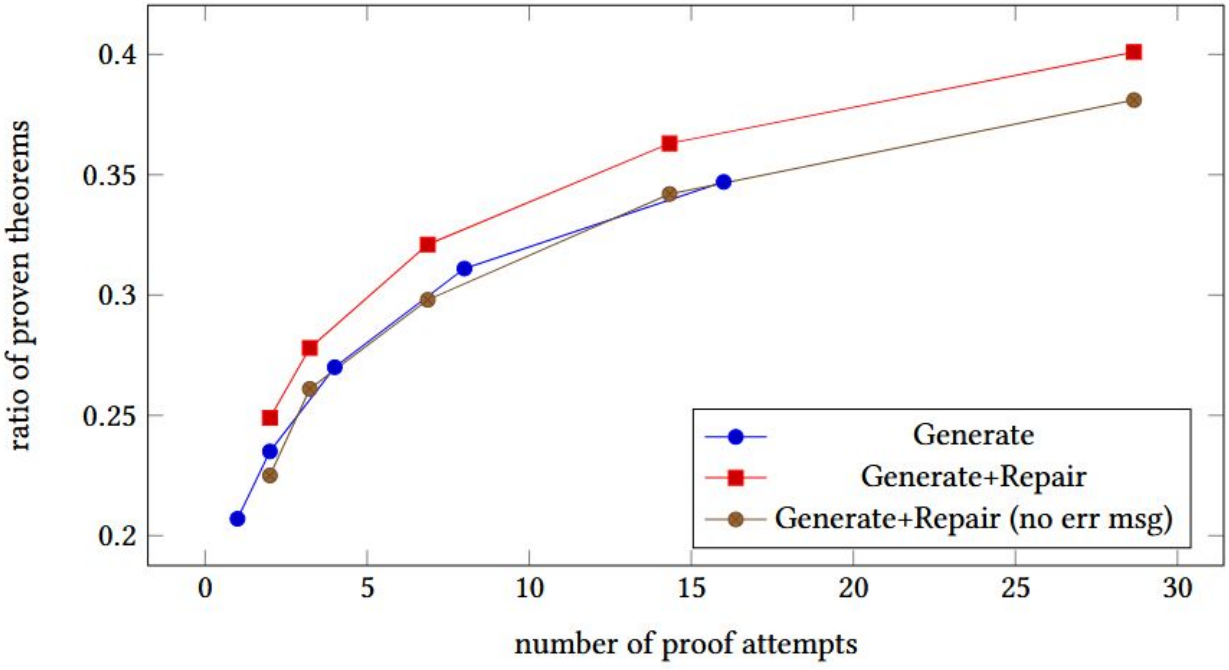


Figure 4: Ratio of theorems proven vs inference cost.

Can LLMs benefit from using the context of the theorem?

Model	16 samples	64 samples
Baldur 8b generate	34.8%	40.7%
Baldur 8b generate + repair	36.3%*	—
Baldur 8b w/ context	40.9%	47.5%
Baldur 62b w/ context	42.2%	47.9%
Baldur 8b w/ context ∪ Thor	—	65.7%

Figure 5: Proof rate of different models.
***The repair approach uses half the number of samples, and then one repair attempt for each sample.**

Does the size of the LLM affect proof synthesis effectiveness?

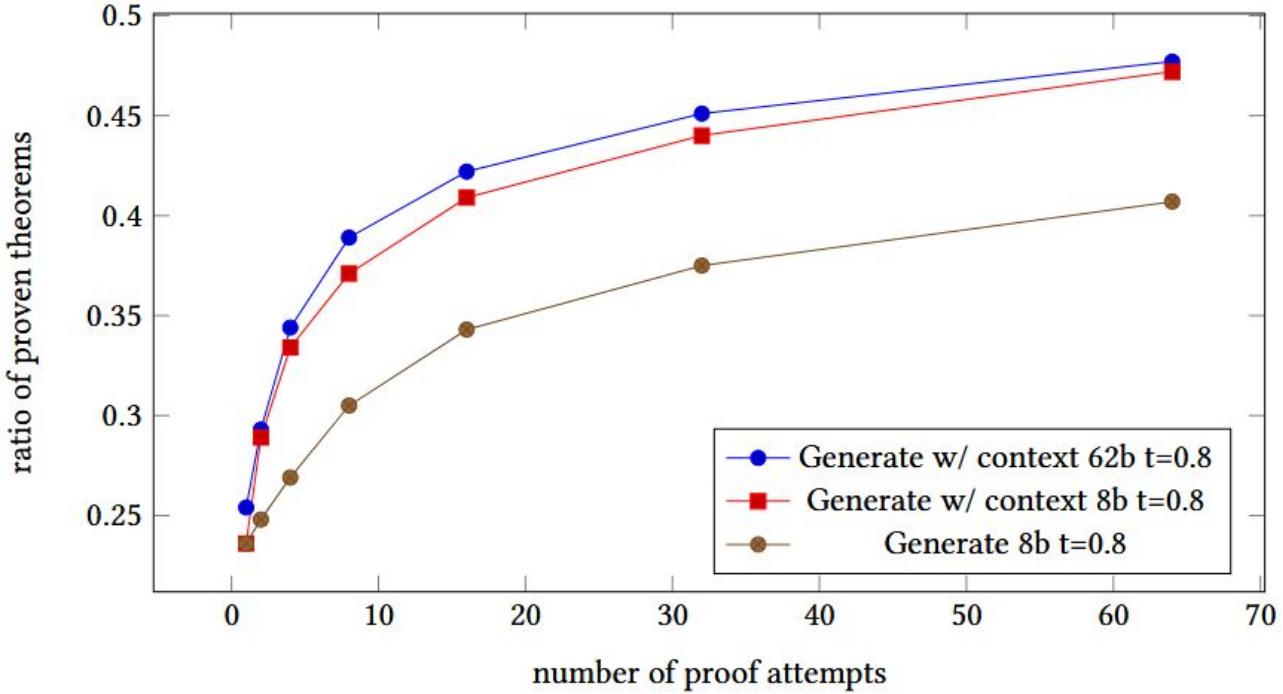


Figure 6: Ratio of theorems proven vs. inference cost for models with different sizes and temperatures.

Discussion

- This paper was 2 years ago. LLMs have gotten *much better* at coding and math.
- When generating code, why not generate a proof?
- For our project: maybe we can use GPUVerify (?)

Citations

1. https://arxiv.org/html/2502.10517v1
2. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45226.pdf#:~:text=In%20this%20paper%2C%20we%20present,times%20in%20paral%02lel%20build%20environments
3. https://llvm.org/devmtg/2015-10/slides/Wu-OptimizingLLVMforGPGPU.pdf
4. https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
5. https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.hellotech.com%2Fblog%2Fwhats-a-gpu-what-gpu-do-you-have&psig=AOvVaw1L8Chm_x7ssYxzqOuSmR9k&ust=1744777794530000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCLDPxPmZ2YwDFQAAAAAdAAAAABAE
6. https://scalingintelligence.stanford.edu/blogs/kernelbench/
7. https://www.researchgate.net/figure/Transformer-encoder-layer-architecture-left-and-schematic-overview-of-a-multi-head_fig1_352992757
8. https://alexdremov.me/speed-up-pytorch-with-custom-kernels-but-it-gets-progressively-darker/
9. https://blog.allardhendriksen.nl/cwi-ci-group/20201023170553-benchmarking_custom_convolutions_versus_pytorch_convolutions/
10. https://www.techradar.com/news/computing-components/graphics-cards/amd-vs-nvidia-who-makes-the-best-graphics-cards-699480

THANK YOU

Instructions for paper presentations: This is a rough outline of what I expect. If you believe a different organization works better for you, feel free to adapt. Everyone on your team must present although the presentation times do not have to be equal.

1) Tell us about the problem and list the papers you selected for presentation.
2) Start with a brief history of work on the problem, and if you can, give us the big picture of what has been done on it. Then say what particular area(s) you focused on and why you selected the papers you will present.
3) Tell us the most important points we should take away from the papers you read. You do not have to spend equal amounts of time on all three papers - if one paper has a lot more insights in your opinion, feel free to spend more time on that.
4) Remember that I am looking for **insights**. A long narrative that just reproduces material from the paper is not interesting.
5) Since you do not have too much time for the presentation, you do not have to show us a lot of experimental results from each paper. You can tell us what the benchmarks or applications are, something about data sizes if that is relevant, what SOTA system(s) they compared against, and how much better their results are.
6) At the end, you should have a slide that tells us what the most important take-away points are from the papers you read and your presentation. If there are other good papers you did not have time to present, list them at the end of your presentation.

some suggestions for good presentations

1) Spend a bit of time in the beginning to introduce your audience to the problem you want to discuss. See for example the sparsity presentation today, in which they spent 5 minutes giving us some background on sparse tensor presentations.

2) If possible, tell the audience why they should care about what you will talk about. The AI Scientist group did this very well by telling you right at the beginning about the AI-generated paper that made waves recently.

3) At the end, present some interesting open questions like the LLM code generation team did last Thursday.

4) If possible, relate what you are discussing to concepts we have covered in class. The DeepSeek team and the sparse tensor guys today did this well.

****

Now that I have said good things about every team (sincerely!), let me give you guys some advice on what can be improved.

1) When presenting experimental results, always specify the machine the results are collected on.

2) Have a small table with the sizes of data in your data sets. Otherwise it is hard to know what small and large mean.

3) When presenting a graph, always start by saying what the x-axis and y-axis are, whether higher is better or lower, and what the different colors mean if you have colored lines. You don't have to spend a lot of time on each graph but say what the main takeaways etc. from each one.