Agenda

8:30–8:45 Introduction

8:45–10:00 LLMLift

- 8:40–9:00 Talk: LLMLift: Verified Code Transpilation with LLMs
- 9:00–10:00 Hands-on: Translating Python to Accelerator DSL

10:00–10:30 Coffee Break ☕

10:30–11:50 Autocomp

- 10:30–10:50 Talk: Autocomp: A Portable Code Optimizer for Tensor Accelerators
- 10:50–11:50 Hands-on: Building a Trainium Code Optimizer

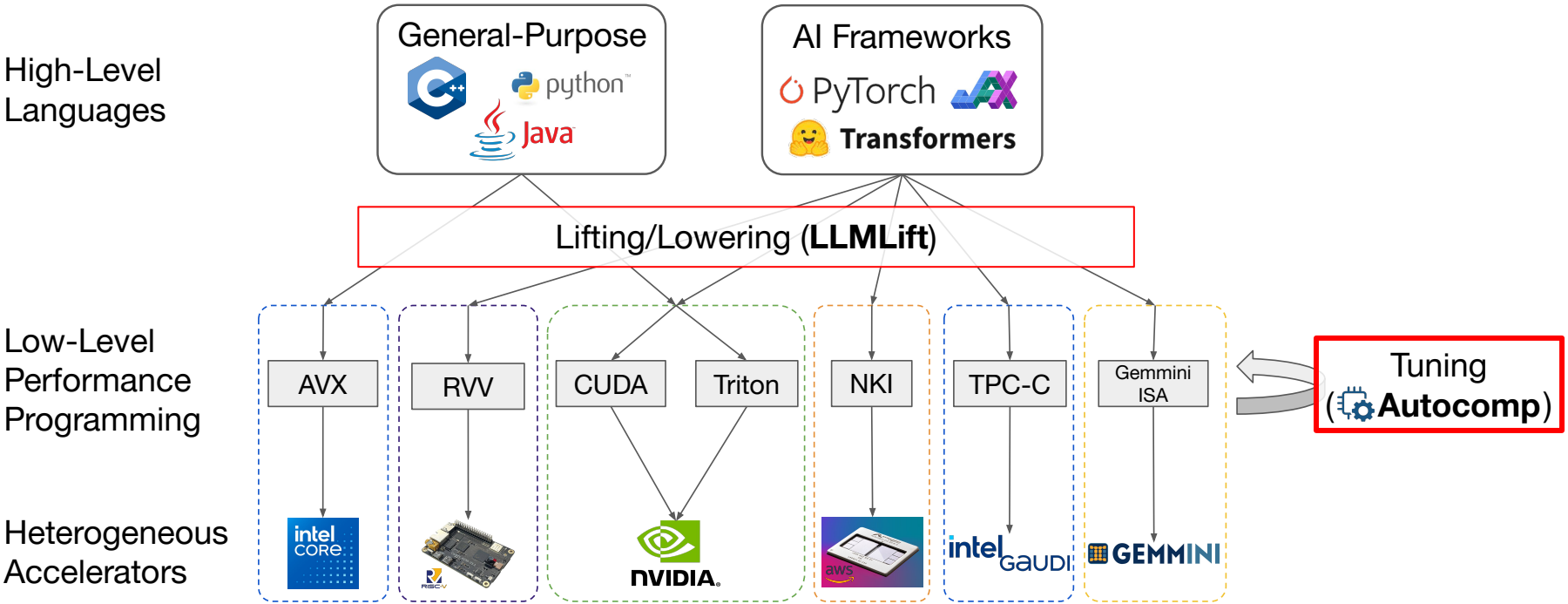
11:50–12:00 Q&A and Future Directions

Autocomp + Trainium Tutorial

ASPLOS 2026

Charles Hong, Sahil Bhatia, Alvin Cheung, Yakun Sophia Shao

The solution: LLMLift + Autocomp



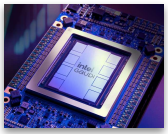
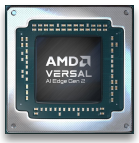
Hundreds of models × dozens of hardware backends

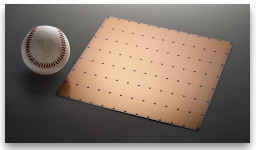
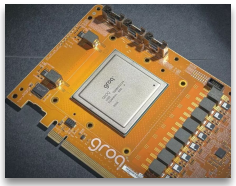
= a combinatorial problem!

OpenAI LLaMA deepseek Qwen Gemma Mistral AI KIMI MINIMAX

×



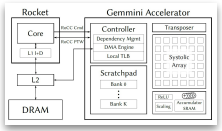




Autocomp: The kernel optimizer for any AI hardware

The old way: Adapting traditional compilers to new hardware requires massive engineering effort

The Autocomp way: Target a new hardware platform just by modifying a structured prompt

Just change:
- Accelerator ISA
- Optimization Menu
- Generation Rules



Gemmini
Academic Accelerator

Trainium 1
Industry Accelerator

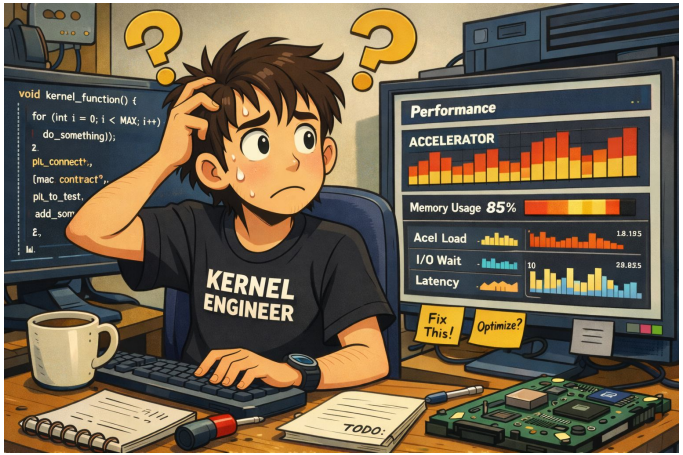
Canaan K230
RVV Dev Board

NVIDIA L40S
GPU

Your Hardware
?

Why not LLMs?

1. LLM doesn't know how to program **specialized accelerator ISA**
2. Code optimization is a highly challenging reasoning problem
3. Need **runtime feedback/profiling** data to know which optimizations to apply



The **Autocomp** Approach

Problem 1: LLM doesn't know how to program accelerator ISA.

Observation: DSLs can be taught in context (our ISLAD'24 paper).

Problem 2: Code optimization is a challenging reasoning problem.

Observation: Each individual optimization (tiling, double-buffering, etc.) is well-understood. Applying the right ones in the right order is the problem!

Problem 3: Need runtime feedback/profiling data to know which optimizations to apply.

Observation: Run code and provide profiling data in context. Search widely and throw out bad search paths.

In-Context Learning

- Early LLM days: we found that providing **both ISA+code examples** is critical to increasing % of functionally correct generated code

	pass@k		
	k=1	k=10	k=50
Zero-shot	0.33%	3.33%	16.7%
One-shot ICL	44.67%	84.42%	99.79%
One-shot ICL (NL-annotated)	46.0%	88.81%	99.98%
No ISA, One-shot ICL (NL-annotated)	1%	9.12%	29.29%

TABLE I
`gpt-4-turbo` TRANSLATED CODE CORRECTNESS FOR MATRIX-VECTOR MULTIPLICATIONS, WITH NATURAL LANGUAGE DESCRIPTIONS FOR FUNCTIONS IN THE INPUT CODE.

- Condensing ISA, hardware architecture info, code examples into reasonably sized prompt is a bit of an art. Currently working on turning this into a repeatable, automated process

Problem 1: LLM doesn't know how to program accelerator ISA.

Observation: DSLs can be taught in context (our ISLAD'24 paper).

Problem 2: Code optimization is a challenging reasoning problem.

Observation: Each individual optimization (tiling, double-buffering, etc.) is well-understood. Applying the right ones in the right order is the problem!

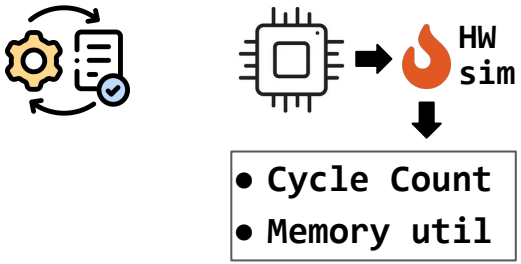
Problem 3: Need runtime feedback/profiling data to know which optimizations to apply.

Observation: Run code and provide profiling data in context. Search widely and throw out bad search paths.

Plan-then-Implement: 2-Phase Optimization Pass

Hardware Feedback:

Correctness + Performance



1. Plan `Prompt`

Accelerator <architecture> and <ISA>.
Original code: <code>
<opt menu> (a list of optimization strategies)
Select **one** and describe how to apply it (what variables/instructions/loops to apply to).

Sample!

2. Implement `Prompt`

Accelerator <architecture> and <ISA>.
Original code: <code>
Plan from phase 1: <plan>
Take the selected optimization, and implement it.

Sample!

Problem 1: LLM doesn't know how to program accelerator ISA.

Observation: DSLs can be taught in context (our ISLAD'24 paper).

Problem 2: Code optimization is a challenging reasoning problem.

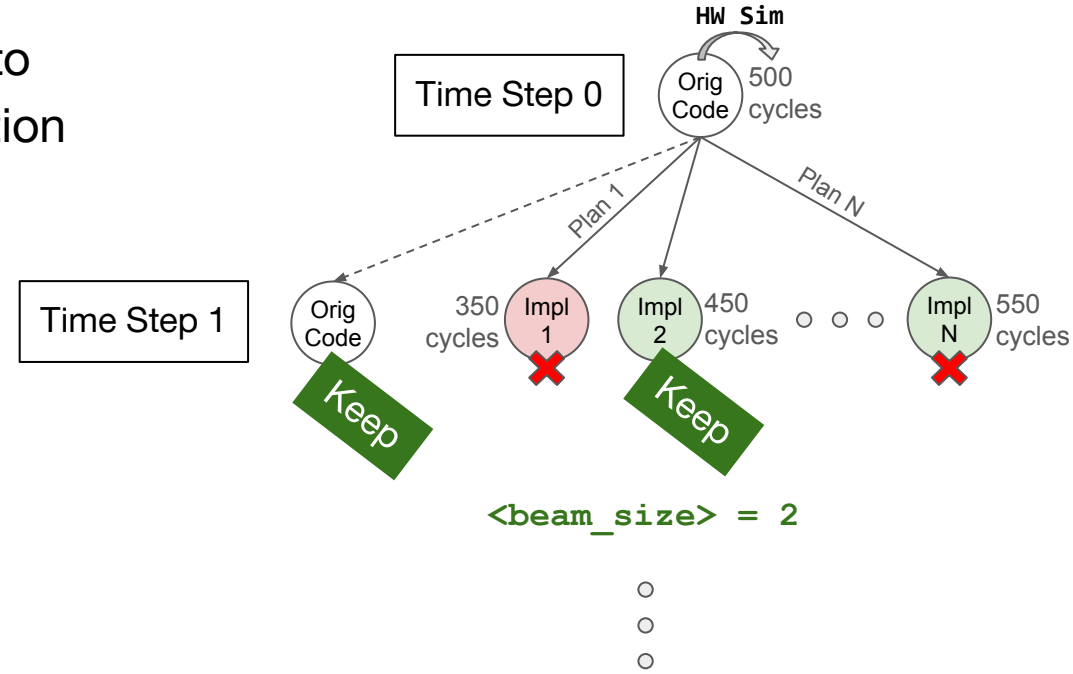
Observation: Each individual optimization (tiling, double-buffering, etc.) is well-understood. Applying the right ones in the right order is the problem!

Problem 3: Need runtime feedback/profiling data to know which optimizations to apply.

Observation: Run code and provide profiling data in context. Search widely and throw out bad search paths.

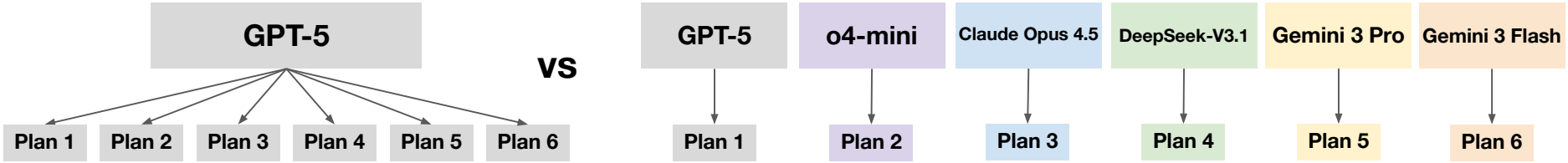
LLM-Integrated Beam Search

- Iteratively optimize
  - Go from correct solution to **faster, also correct** solution
- Use test cases to eliminate functionally incorrect code
- Keep `<beam_size>` best implementations at each time step



Increasing Plan/Code Diversity

- Sampling gives us different choices, but we noticed similar plans were often being regenerated
- **Optimization Menu Dropout:** Each time a plan is generated, randomly "drop out" some of the suggested optimizations
  - We used 70% probability of being dropped
- **LLM Ensembling**: When sampling plans/code, divide samples between multiple different models



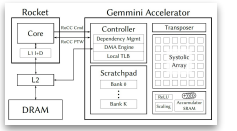
Experimental Results

Back to the motivation…

The old way: Adapting traditional compilers to new hardware requires massive engineering effort

The Autocomp way: Target a new hardware platform just by modifying a structured prompt

Just change:
- Accelerator ISA
- Optimization Menu
- Generation Rules



Gemmini
Academic Accelerator

Trainium 1
Industry Accelerator

Canaan K230
RVV Dev Board

NVIDIA L40S
GPU

Your Hardware
?

Evaluation: Gemmini

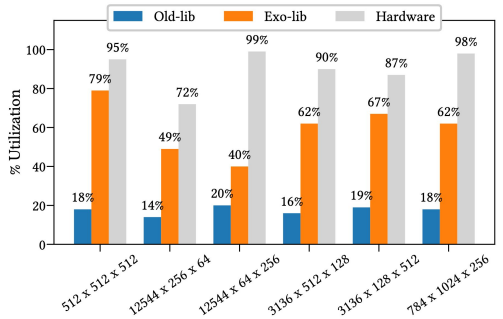
- Prior work used Exo scheduling DSL to **hand-optimize** Gemmini code
  - Gemmini: open-source tensor accelerator generator
- Let's see if we can beat **human experts** on **key workloads (GEMM/Conv)**!

Autocomp **vs.**

exo

in app.py

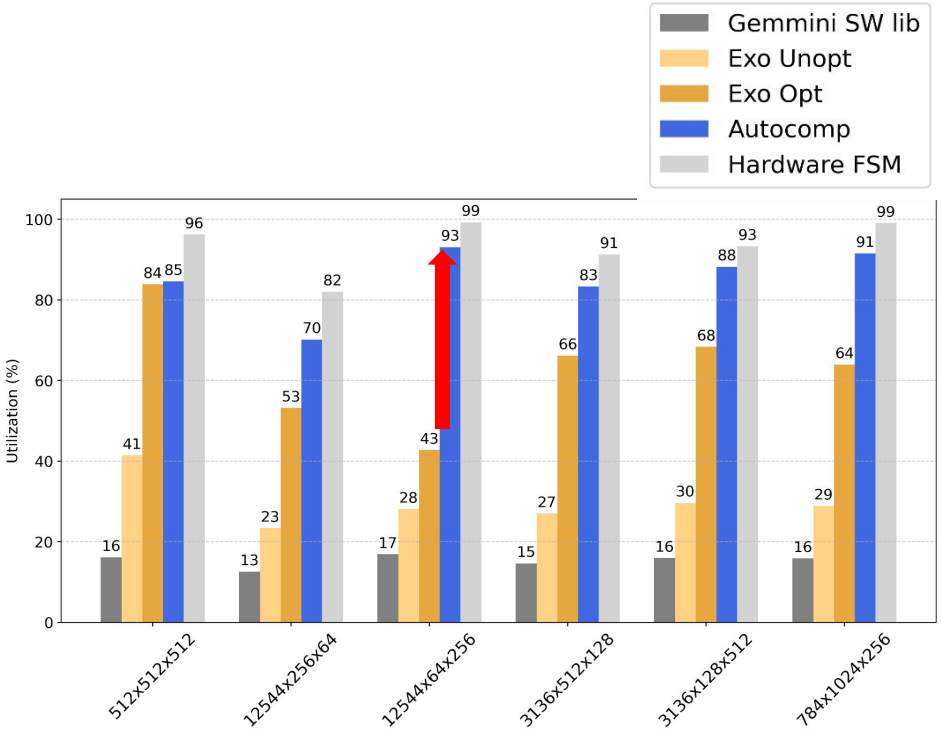
```
def gemm(...):
    res: R[...] @ ACCUMULATOR
    a  : R[...] @ SCRATCHPAD
    b  : R[...] @ SCRATCHPAD
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Load C to res
            for ko in seq(0, 8):
                # Load A to a
                for ii in seq(0, 16):
                    for ki in seq(0, 16):
                        a[...] = A[...]
                ... # Load B to b
                # Matmul of a and b
                for ii in seq(0, 16):
                    for ji in seq(0, 16):
                        for ki in seq(0, 16):
                            res[..]+=a[..]*b[..]
            ... # Store res to C
```



(a) MATMUL utilization (as a percentage of peak FLOPS). X axis labels are the size of matrices in N x M x K.

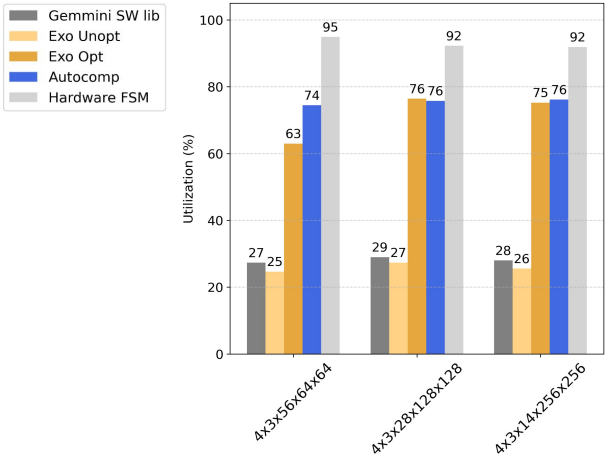
GEMM Optimization Results

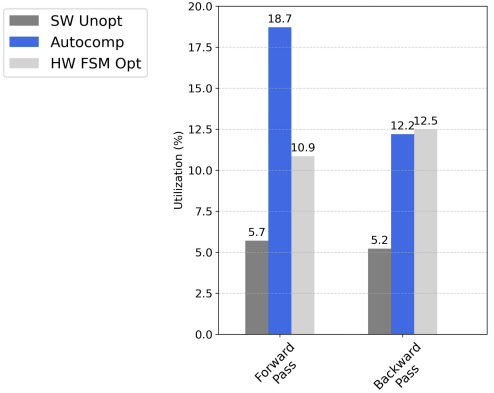
- **Highlights:**
  - **5.6x** speedup over Gemmini software library
  - **1.4x** speedup over previous best hand-optimized code
- We can explore much more of the search space than a human can



Evaluation: Convolution/TinyMPC

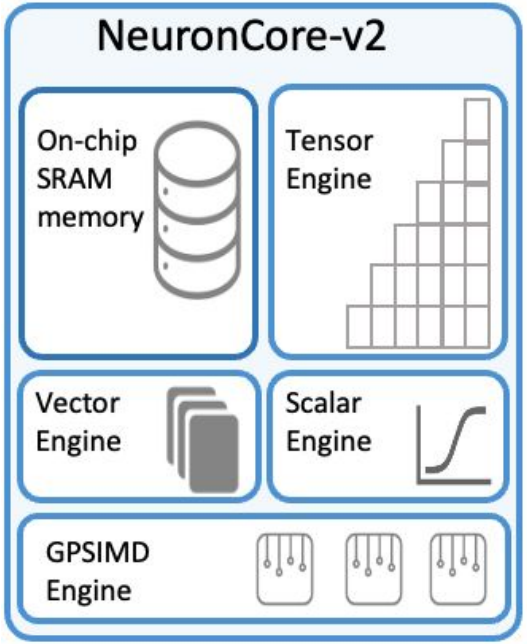
- Also optimized other workloads
  - Convolution
  - Robotics model-predictive control (sequence of small linear algebra kernels)
- See paper for details!





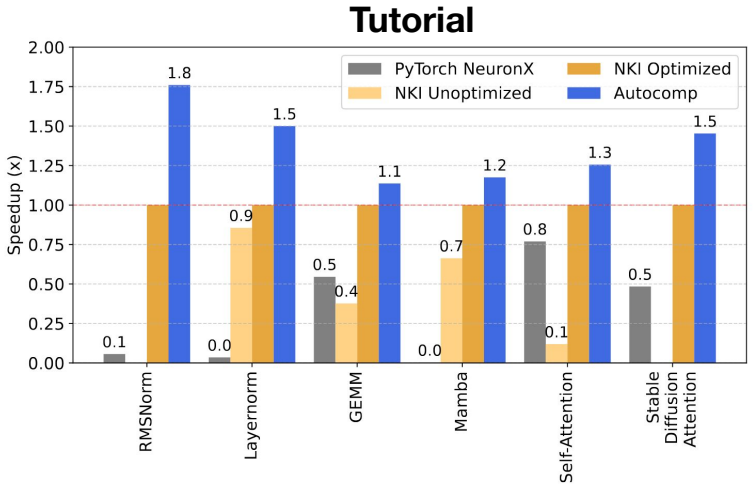
AWS Trainium

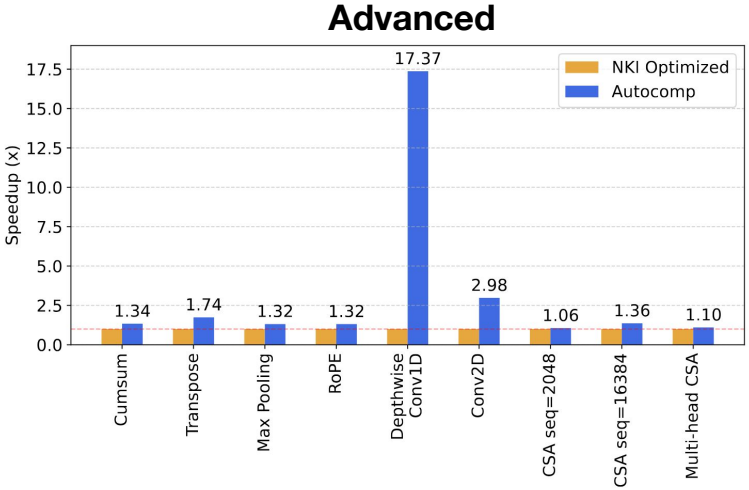
- Systolic array-based architecture with tensor engine, vector engine, scalar engine
- Kernels written in hardware specific DSL called Neuron Kernel Interface (NKI)
- The tutorial environment gives you access to a `trn1.2xlarge` instance



Evaluation: Trainium

- Benchmark: AWS's own Trainium NKI kernel examples
- Tutorial:
  - **1.36x** faster than NKI Optimized (starting from NKI Unoptimized)
  - **13.52x** faster than PyTorch
- Advanced (examples for production users):
  - **1.9x** faster than baseline

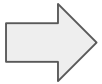




Trainium Conv1D

- Key optimization (~10x speedup): Add a constant-size inner loop to the main convolution loop
- Small/static loop bound seems to enable coalescing of tensor_reduce() operations (profiler summary shows reduced # of VectorEngine insts)
    - Coalescing can also be done manually

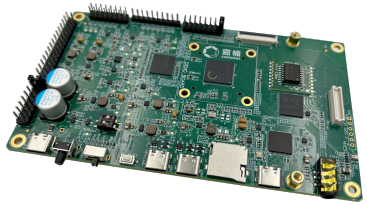
```
# convolution loop
for o in nl.affine_range(out_image_size):
    img_tile  = img_local[i_p, i_f_a + o]    # (C_T
    filt_tile = filt_psum[i_p, i_f_win]      # (C_T
    prod = nisa.tensor_tensor(img_tile,
                              filt_tile,
                              op=np.multiply)
    out_sb[i_p, o] = nisa.tensor_reduce(np.add,
                                        prod,
                                        axis=[1])
```



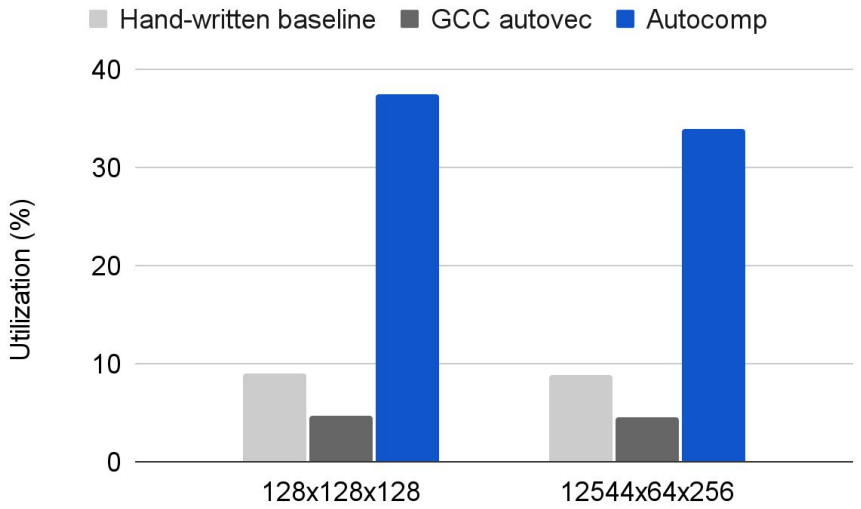
```
# --- Phase B + Phase C (optimized): block the big 1D output dim ------
# We tile the out_image_size dimension into fixed-size blocks.
OUT_TILE = 64
NUM_FULL_BLOCKS = out_image_size // OUT_TILE

# Process full blocks of OUT_TILE outputs
for b in nl.affine_range(NUM_FULL_BLOCKS):
    base_out = b * OUT_TILE
    for o in nl.affine_range(OUT_TILE):
        # identical to original per-output compute, now grouped by blocks
        img_tile  = img_local[i_p, i_f_a + (base_out + o)]   # (C_TILE_SIZE
        filt_tile = filt_psum[i_p, i_f_win]                  # (C_TILE_SIZE
        prod = nisa.tensor_tensor(img_tile,
                                  filt_tile,
                                  op=np.multiply)
        out_sb[i_p, base_out + o] = nisa.tensor_reduce(np.add,
                                                       prod,
                                                       axis=[1])
```

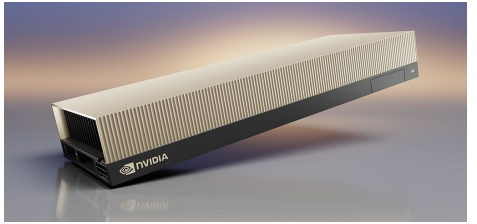
See our [blog post](#)!

Evaluation: RISC-V Vector (RVV)



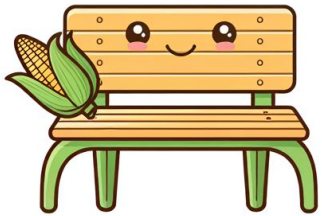
- Optimize GEMM on an RVV 1.0 compliant dev board (Canaan K230)
- Optimization menu similar to that for Gemmini, with the addition of:
  1. “Prefetching”
  2. “Register blocking”
  3. “Maximize LMUL”
  4. “Fuse instructions”
- **Outperforms GCC auto-vectorization by ~8x!**





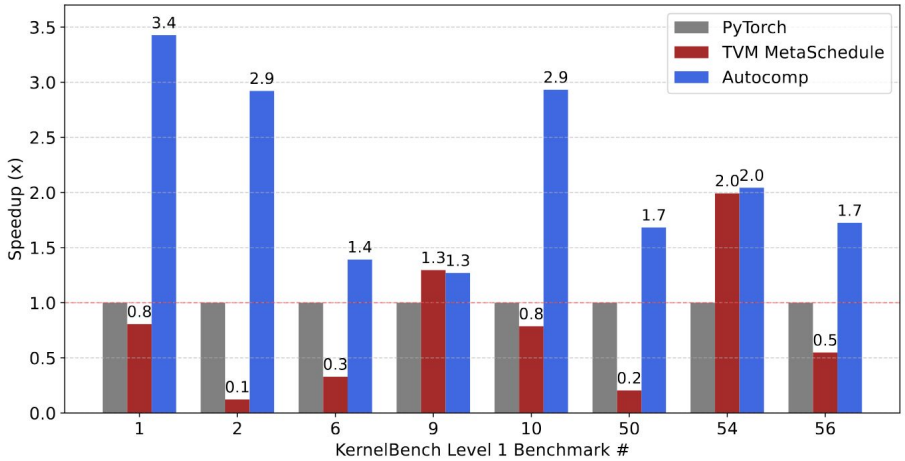
Evaluation: NVIDIA L40S GPU

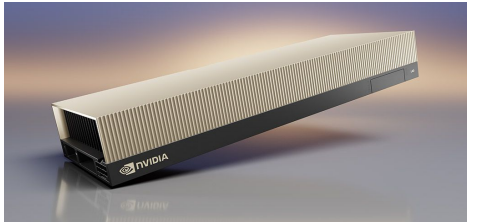
- Modify **optimization menu** to generate/optimize inline CUDA
  - e.g., “Use shared memory to reduce global memory bandwidth usage” or “Minimize divergent branches within warps”
- Demonstrates Autocomp’s **portability**: just had to change a few prompts!
- Benchmark: Stanford’s KernelBench
  - Level 1: Single operators
  - Level 2: Multiple operators
  - Level 3: Full model architectures



Comparison to TVM (SOTA Auto-Scheduler)

- We optimize PyTorch GEMM/conv kernels from KernelBench level 1
- Compare against PyTorch and TVM MetaSchedule
- **2.05x** faster than PyTorch, **3.80x** faster than TVM



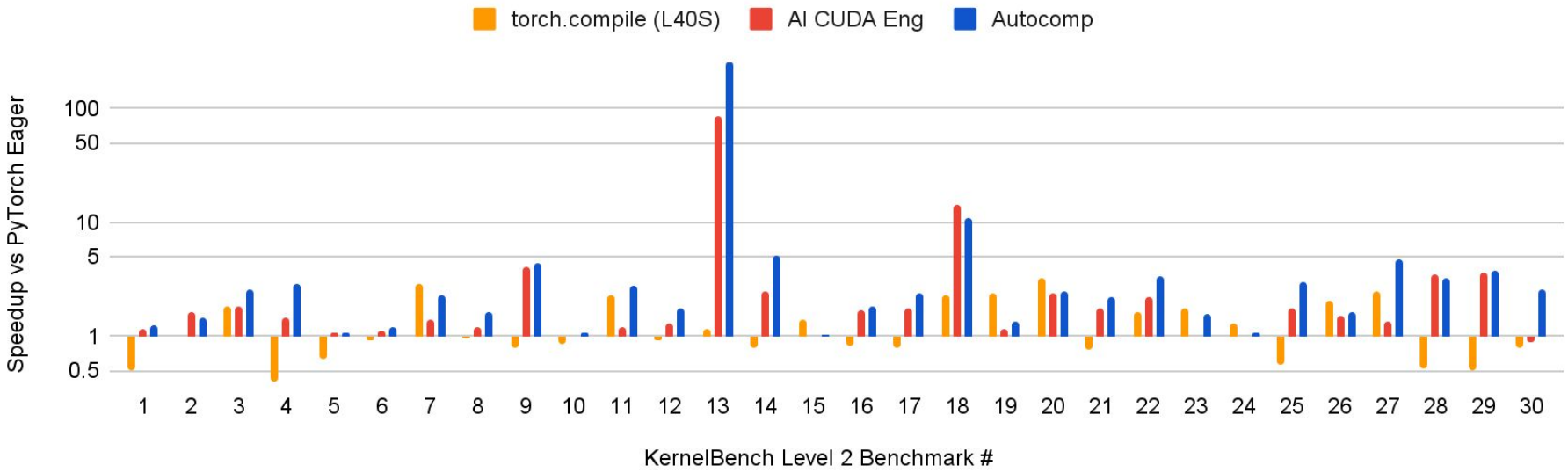


Evaluation: NVIDIA L40S GPU (cont.)

- Next: **KernelBench** level 2/3
  - Level 2: Sequences of operators (e.g., Conv + Bias + ReLU, Matmul + Scale + Sigmoid)
  - Level 3: Full model architectures (e.g. AlexNet, ResNet, ViT)
- Compare against another LLM search baseline: AI CUDA Engineer

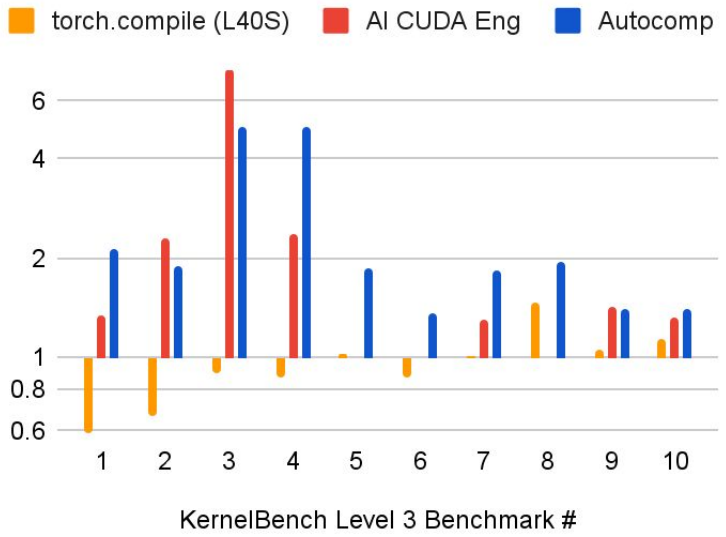
KernelBench Level 2/3

- State-of-the-art results!
  - **2.61x speedup** on first 30 kernels of level 2 (AI CUDA Eng: 1.91x)



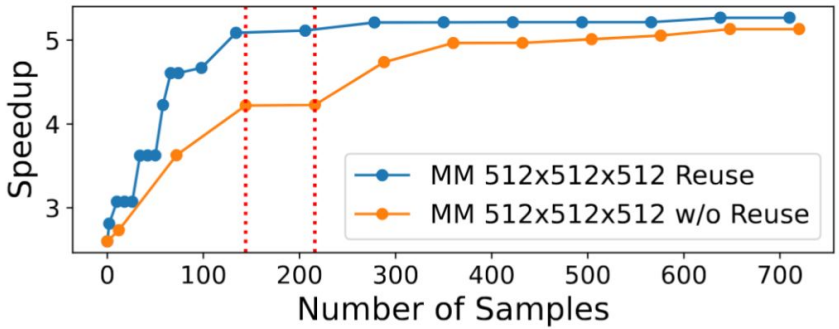
KernelBench Level 2/3

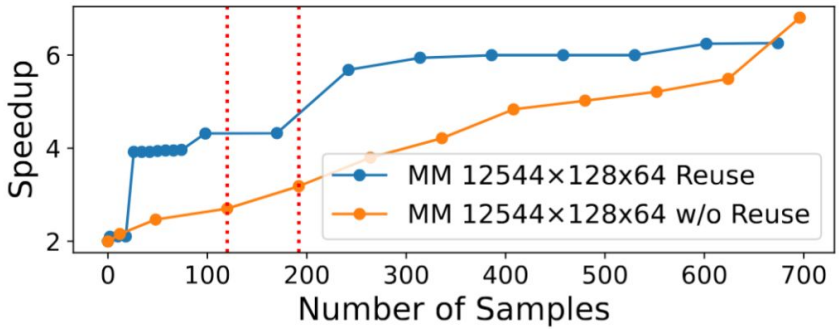
- State-of-the-art results!
  - **2.61x speedup** on first 30 kernels of level 2 (AI CUDA Eng: 1.91x)
  - **2.11x speedup** on first 10 kernels of level 3 (AI CUDA Eng: 1.63x)



Schedule Reuse

- Each optimization run **provides valuable insight** into best strategy
- How do we leverage this knowledge in future runs?
- One way: **remember** the **sequence of optimizations** selected, and **reapply** it to similar GEMMs (i.e. square, column-dominant, row-dominant)
    - Then start beam search from here
- **Improves perf under a ~100-sample budget by 24%**



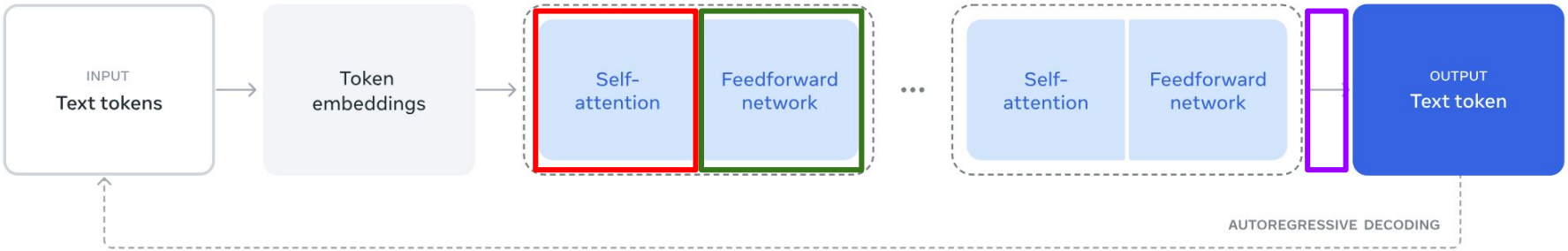


Can Autocomp make a real-world difference?

A real-world problem

- Optimizing **Llama-3.2-1B** on **Trainium trn1.2xlarge** using AWS's `neuronx-distributed-inference` (NXDI) library
- Optimize 3 kernels independently: 1) attention, 2) MLP, 3) LM head

Llama Architecture



Latency Breakdown (Batch=32)

Attention (27%)	MLP (34%)	LM Head (18%)	Other (21%)

Characterizing the 3 kernels

<table>
<tr>
<td>Grouped query attention (GQA):

<code>softmax(Q @ K.T / sqrt(d)) @ V</code>

Note: same K/V shared across multiple Q heads.</td>
<td>SwiGLU MLP/FFN:

<code>(SiLU(x @ gate.T) * (x @ up.T)) @ down.T</code>

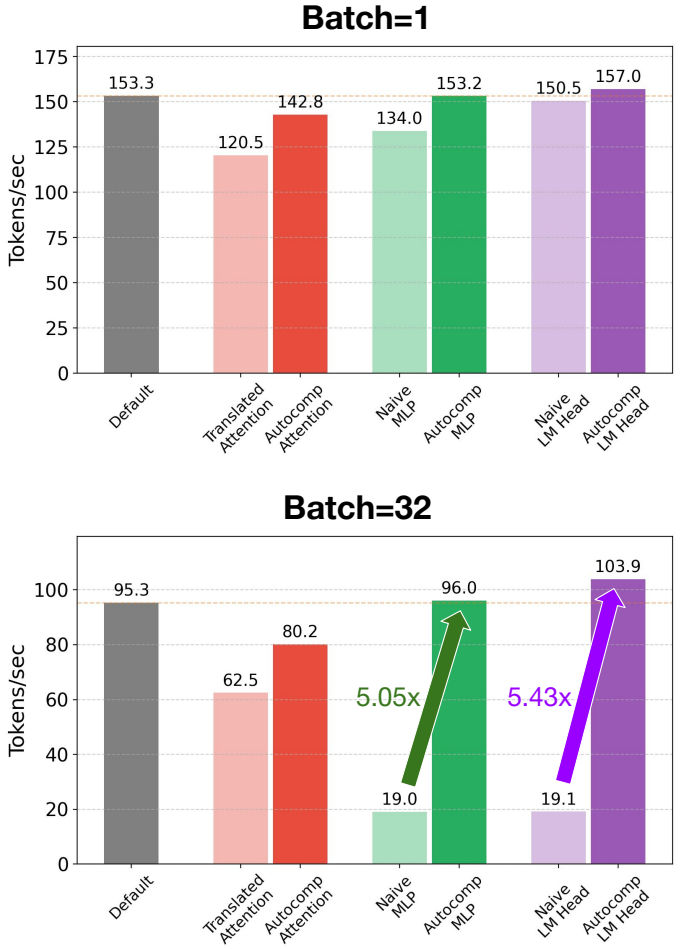
Three GEMMs, one activation function.</td>
<td>LM head (output projection):

<code>hidden @ vocab_weights.T</code>

Short & wide GEMM.</td>
</tr>
<tr>
<td>Procedure: Translate PyTorch to NKI, then optimize NKI</td>
<td colspan="2">Procedure: Start with naive NKI implementation (based on tutorials/reference code), then optimize</td>
</tr>
</table>

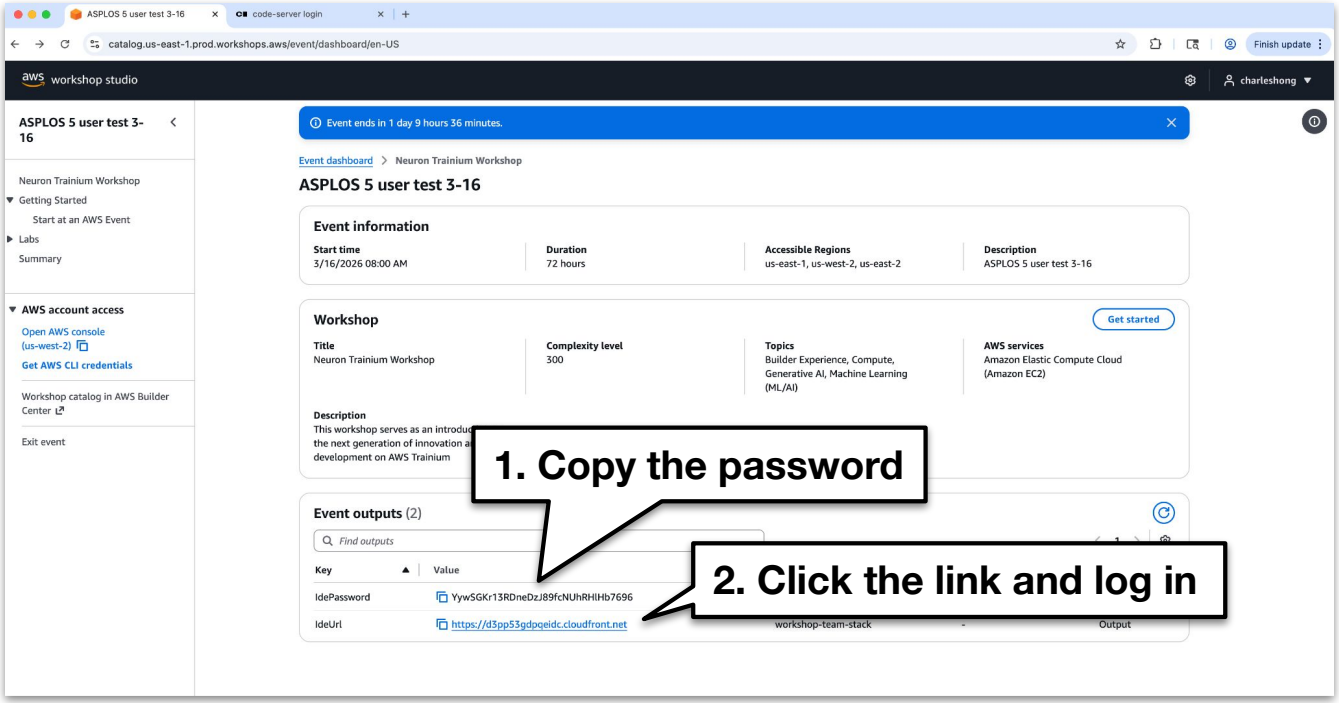
Llama-3.2-1B Performance

- How do our kernel implementations affect **end-to-end inference** performance (tokens/sec)?
- Autocomp **LM head** increases overall throughput by **1.02x** (batch=1) and **1.09x** (batch=32)
- Autocomp **MLP** matches NXDI performance, speeding up naive implementation by **1.14x** (batch=1) and **5.05x** (batch=32)
- Not yet able to match NXDI's **attention** performance



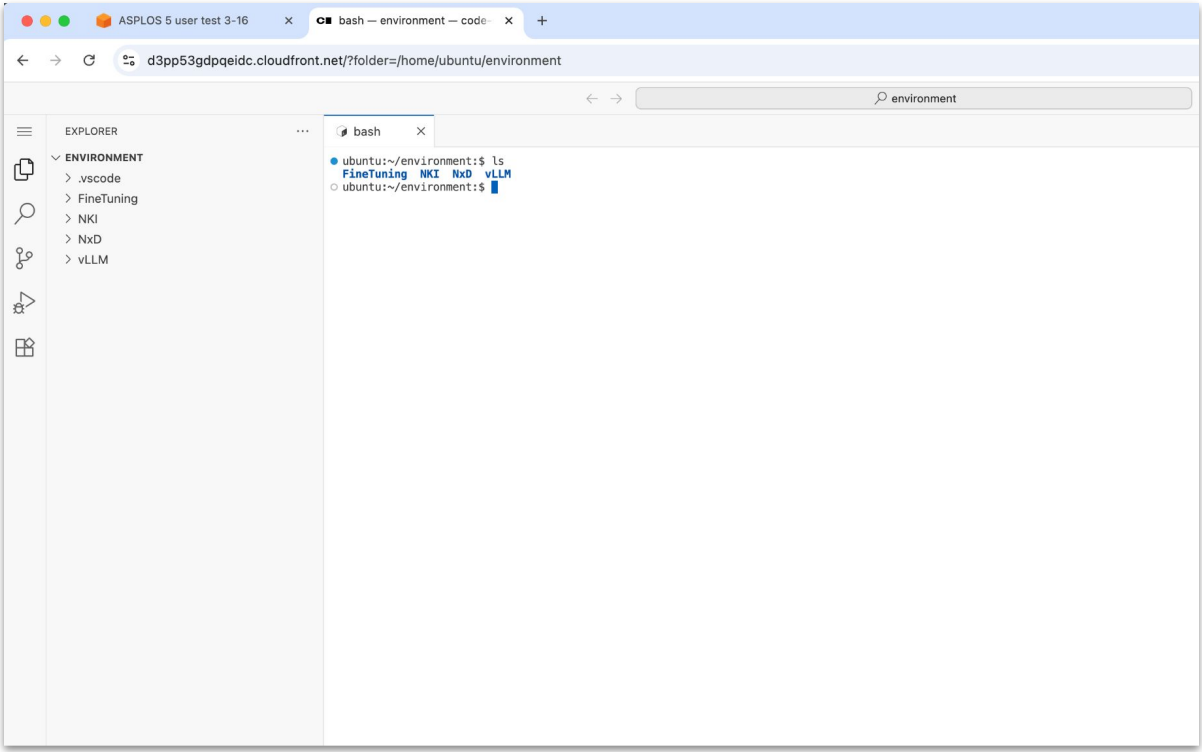
Hands-On: Getting Started

AWS Tutorial Environment (if you weren't here earlier)



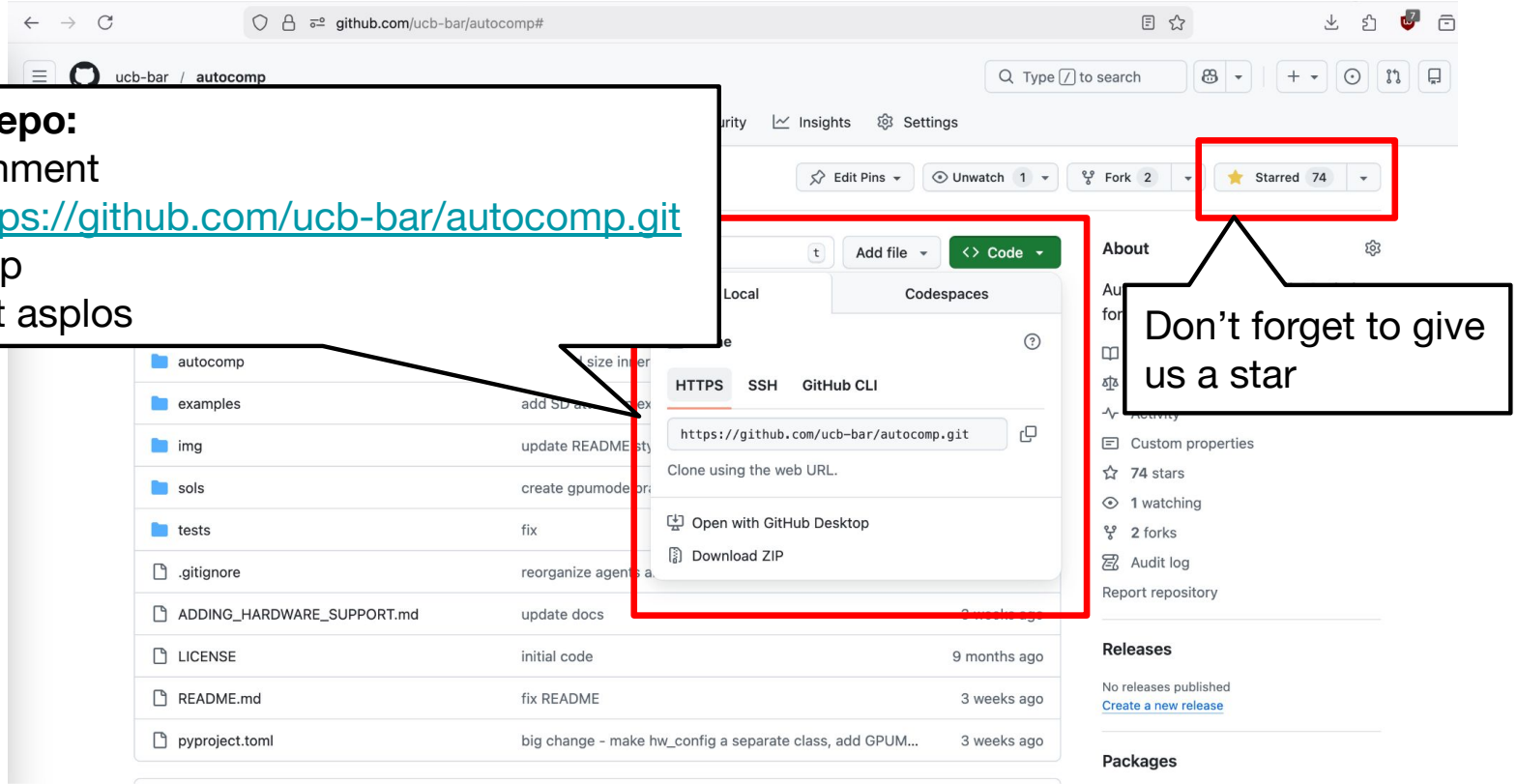
AWS Tutorial Environment

Looks something like this:



Clone Autocomp into the instance

Clone the repo:
cd ~/environment
git clone https://github.com/ucb-bar/autocomp.git
cd autocomp
git checkout asplos



Autocomp repo organization

autocomp/

- **autocomp/**
  - **agents/** - Hardware-specific agents (Saturn, Gemmini, Trainium, CUDA)
  - **backend/** - Hardware-specific evaluation scripts
  - **hw_config/** - Defines hardware configuration (e.g. Trainium instance type)
  - **search/** - Logic for Autocomp beam search
  - **common/** - LLM utils, logging, etc.
- sols/
- tests/

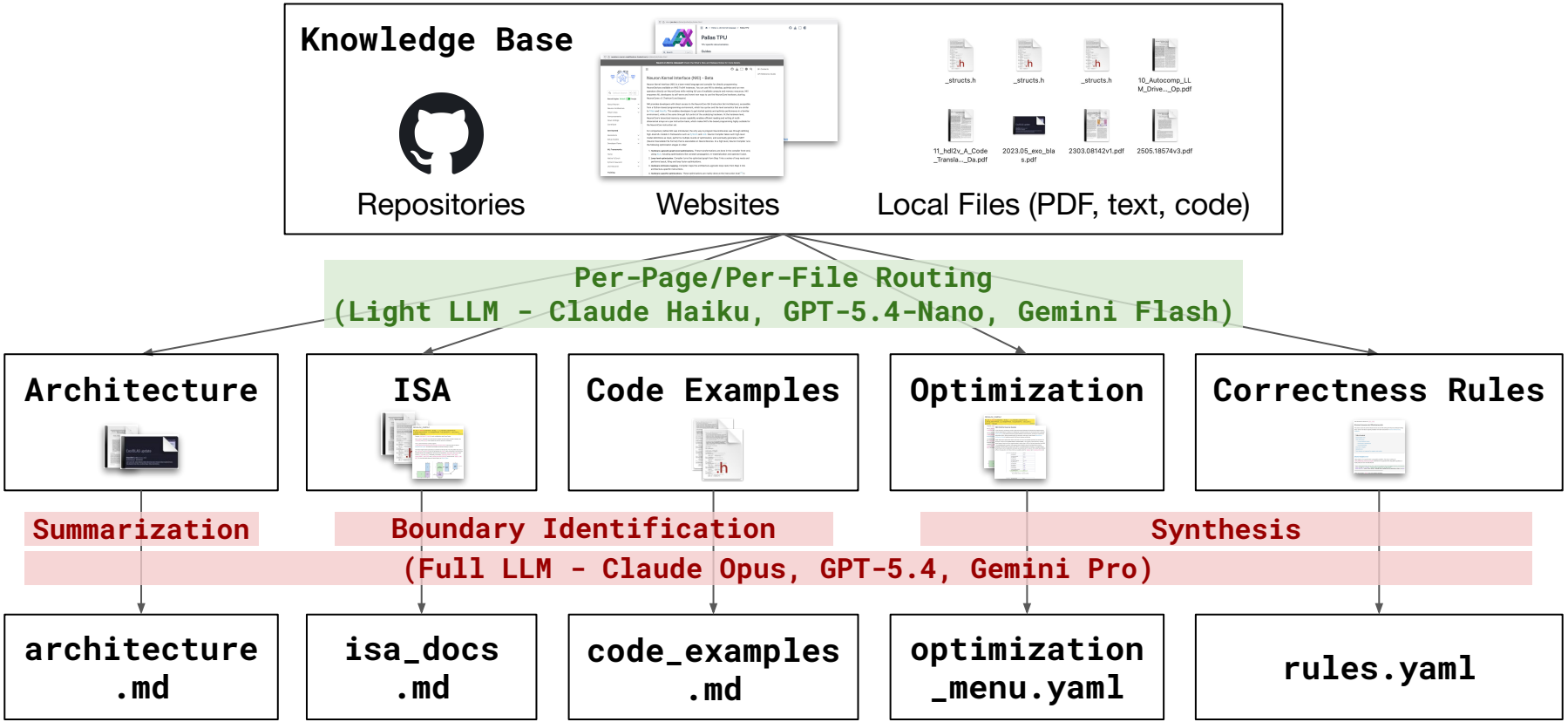
Autocomp repo organization

autocomp/

- autocomp/
  - agents/
  - backend/
  - hw_config/
  - search/
  - common/
- **sols/** - The actual code to be optimized (used by search/ and backend/)
- **tests/** - Test harnesses specific to individual sols (used by backend/)

Agents

Autocomp Agent Builder



Autocomp Agent Builder

- Point it at your docs, code, PDFs, or API webpages, and it produces a set of human-editable config files that define a fully functional Autocomp agent
- **autocomp/autocomp/agent_builder/built_agent.py**
  - `_get_problem_context()` identifies ISA instructions and code examples relevant to a given workload
  - `_get_propose_new_menu_prompt()` generates workload-specific optimization strategies

isa_docs.md

```
## Tensor Operations
### nc.matmul()
This instruction does a matmul…
## Memory Operations
### nisa.dma_copy()
Copies data from HBM to and from SBUF…
```

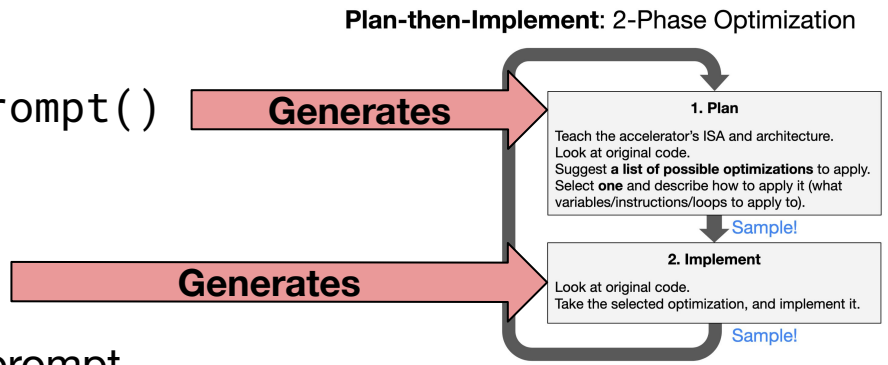
code_examples.md

```
## Matrix multiplication example
def tiled_matmul():
    ...
```

BuiltLLMAgent (extends LLMAgent)

autocomp/autocomp/agent_builder/built_agent.py

- 2 main entry points for search:
  - `_get_propose_optimizations_prompt()`
    - Generates planning prompt
  - `_get_implement_code_prompt()`
    - Generates code implementation prompt



- Pulls metadata for prompts from built agent directory
  - For Trainium: see `agent_builder/.built/trn-nki1/`

Trainium Planning Prompt

`_get_propose_optimizations_prompt()`

Collects data sources and constructs string prompt

1. Trainium architecture overview
2. NKI ISA (filtered by agent)
3. Code examples (filtered by agent)
4. Original code
5. Original code latency
6. Menu opts - defined in `get_opt_menu_options()`, with dropout applied
7. Instruction
8. Rules (correctness guide)

Trainium Implementation Prompt

`_get_implement_code_prompt()`

Similar to plan generation; slightly simpler

1. Trainium architecture overview
2. NKI ISA
3. Original code
4. The plan generated by phase 1
5. Instruction
6. Rules

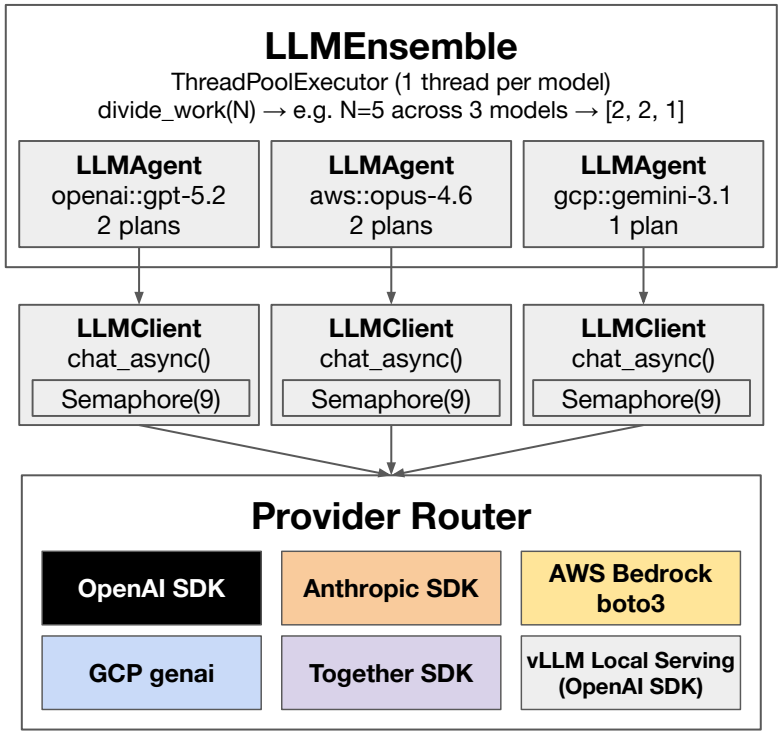
Who calls the LLM?

autocomp/autocomp/agents/llm_agent.py
autocomp/autocomp/agents/llm_ensemble.py

- Collect the prompts for a given phase and sends them to LLMClient

autocomp/common/llm_utils.py

- LLMClient reformats API calls and routes requests to local/remote providers
- You choose your own provider
- **Fully parallel** to minimize time spent waiting for API calls

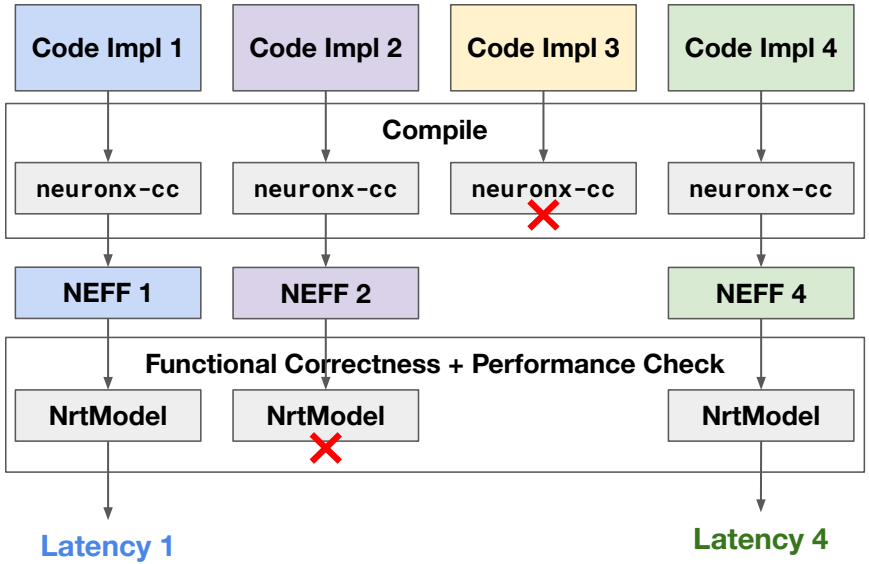


Evaluation Backend

Trainium Evaluation: Decoupled Compile/Execute

autocomp/autocomp/backend/trn/trn_eval.py

- Decouple compilation from execution to maximize evaluation throughput
- Use all CPU cores to run `neuronx-cc` compiler in parallel for different samples
- Run correctness + performance tests sequentially to prevent contention



Search

Running Search

autocomp/autocomp/search/search.py

bash autocomp | search.py | 0_conv1d_ref.py U | 0_conv1d_test.py

autocomp > autocomp > search > search.py > main

```
789  def main():
790      # Select evaluation backend, LLM agent, and hardware config
791      backend_name = "trn"  # Options: "gemmini", "trn", "tpu", "kernelbench", "gpumode"
792      agent_name = "built:trn-nki1"  # Options: "gemmini", "trn", "cuda", "built:<name>", or a path to a built agent (for TPU v6e, use built:tpu-v6e)
793      simulator = None # "firesim" or "spike" if backend_name == "gemmini"; "gpumode-local" or "gpumode-cli" if backend_name == "gpumode"
794      # Hardware configuration
795      hw_config = TrnHardwareConfig("trn1.2xlarge")
796      # Examples:
797      # hw_config = TrnHardwareConfig("trn1.2xlarge")
798      # hw_config = GemminiHardwareConfig(pe_dim=16, spad_size_kb=256, acc_size_kb=64)
799      # hw_config = CudaHardwareConfig("NVIDIA L40S", "2.5.0", "12.4")
800      # hw_config = TpuHardwareConfig("v6e-1")
801
802      # Models are specified as "provider::model"
803      # Valid providers are "openai", "anthropic", "together", "aws", "gcp", "vllm"
804      # If no provider is specified, the provider is inferred from the model name
805      models = ["aws::us.anthropic.claude-opus-4-5-20251101-v1:0", "aws::zai.glm-4.7", "aws::deepseek.v3.2", "aws::moonshotai.kimi-k2.5"] # Models for planning
806      code_models = None # Models for code implementation (None means use same as planning models)
807      metric = "latency"
808      search_strategy = "beam"
809      iterations = 3
810      prob_type = "trn-tutorial" # see README.md or sols directory for available problems
811      prob_id = 1
```

Install Autocomp package and dependencies

```
> source /opt/aws_neuronx_venv_pytorch_latest/bin/activate

> cd ~/environment/autocomp

> pip install -e .
```

If you weren't here earlier, access the env at <link> and:
cd ~/environment
git clone https://github.com/ucb-bar/autocomp.git
cd autocomp
git checkout asplos

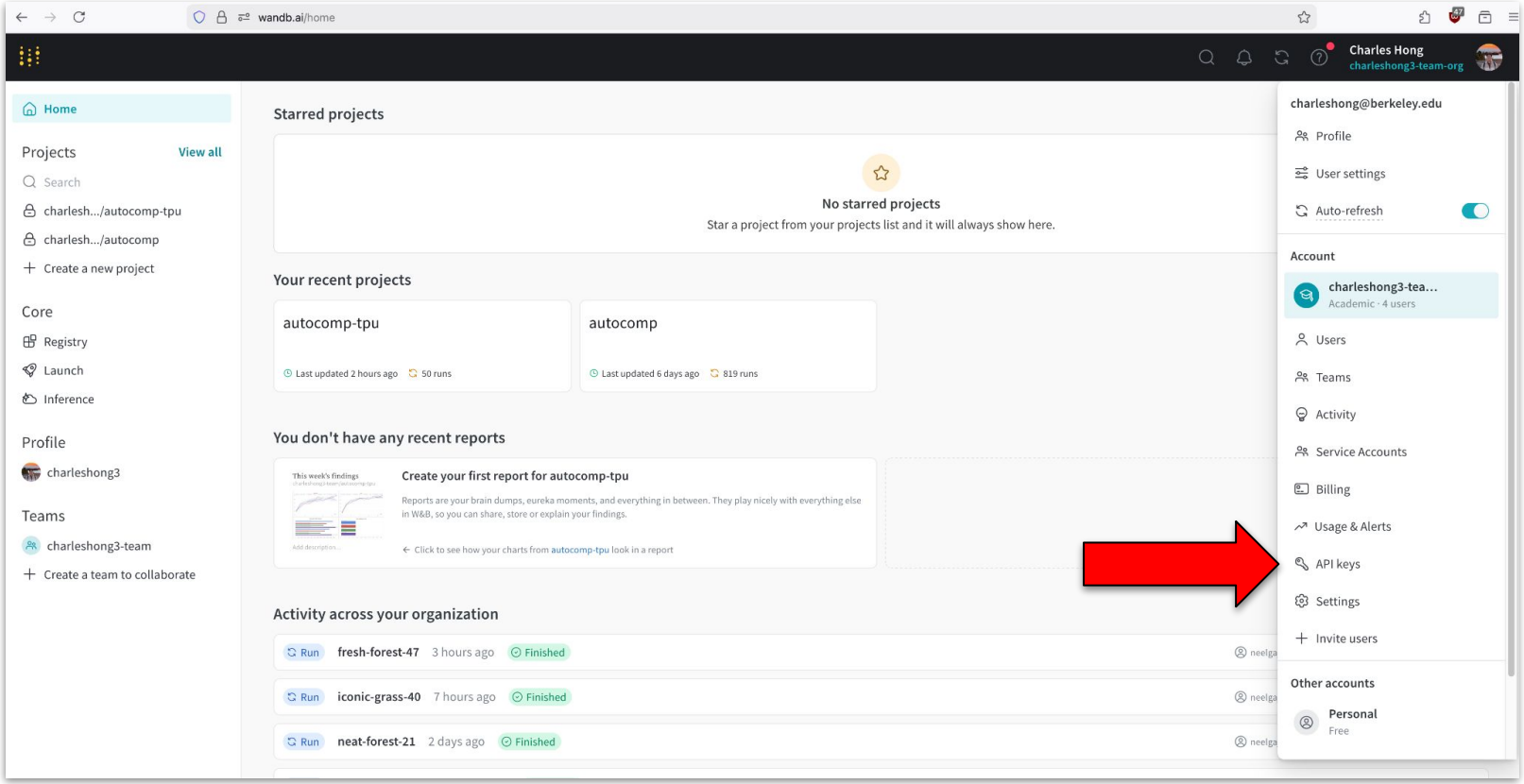
Run search

```
> python autocomp/search/search.py
```

Enter your W&B API key if you have an account (or make an account, it's quick)

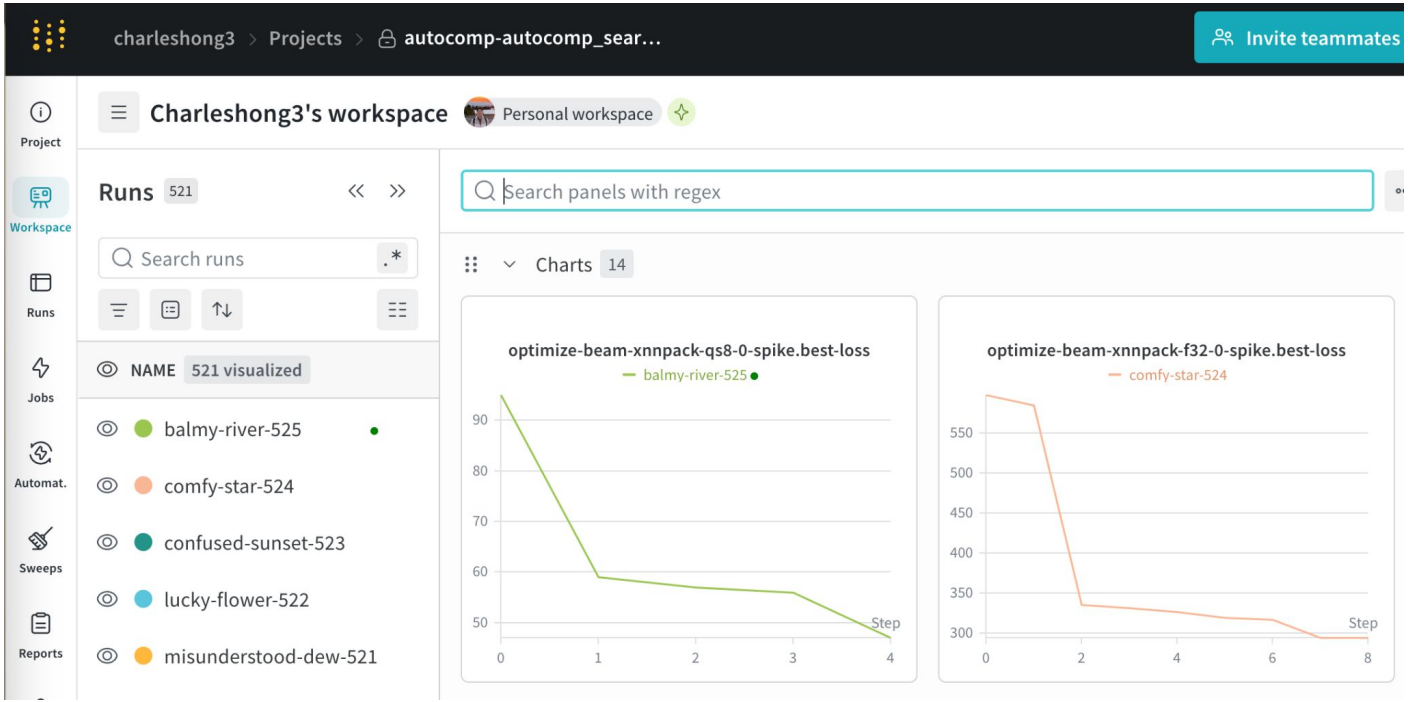
```
ubuntu:~/environment/autocomp:$ python autocomp/search/search.py
[2026-03-18 06:06:04,913 INFO llm_utils.py:62 <module>] Keys unavailable: OPENAI_API_KEY, ANTHROPIC_API_KEY, TOGETHER_API_KEY, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, GOOGL
[2026-03-18 06:06:04,975 INFO search.py:936 main] Output directory: output/trn_trn-tutorial_1_beam_iters8_Trainium_trn1.2xlarge_us-4-5-20251101-v1:0_aws::zai.glm-4.7_aws::dee
b4_score1_ms1_fgisa1_ex0.25
[2026-03-18 06:06:05,485 INFO trn_eval.py:467 _try_combined_evaluation] Phase 1: compiling ref + 1 implementations in parallel
[2026-03-18 06:06:21,324 INFO trn_eval.py:266 _compile_impls_parallel] Compile ref: OK
[2026-03-18 06:07:21,992 INFO trn_eval.py:266 _compile_impls_parallel] Compile 0: OK
[2026-03-18 06:07:21,992 INFO trn_eval.py:474 _try_combined_evaluation] Phase 1 complete: ref=OK, 1/1 implementation NEFFs saved
[2026-03-18 06:07:21,992 INFO trn_eval.py:482 _try_combined_evaluation] Phase 2: correctness + benchmark via libnrt
[2026-03-18 06:07:34,106 INFO trn_eval.py:513 _try_combined_evaluation] Code 0 latency: 2.425
[2026-03-18 06:07:34,107 INFO search.py:236 __init__] Initial code scores:
[2026-03-18 06:07:34,107 INFO search.py:238 __init__] 2.425
wandb: (1) Create a W&B account
wandb: (2) Use an existing W&B account
wandb: (3) Don't visualize my results
wandb: Enter your choice: 2
wandb: You chose 'Use an existing W&B account'
wandb: Logging into https://api.wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.me/wandb-server)
wandb: Create a new API key at: https://wandb.ai/authorize?ref=models
wandb: Store your API key securely and do not share it.
wandb: Paste your API key and hit enter:
wandb: No netrc file found, creating one.
wandb: Appending key for api.wandb.ai to your netrc file: /home/ubuntu/.netrc
wandb: Currently logged in as: charleshong3 to https://api.wandb.ai. Use `wandb login --relogin` to force relogin
wandb: Tracking run with wandb version 0.25.1
wandb: Run data is saved locally in /home/ubuntu/environment/autocomp/wandb/run-20260318_060804-5xhqy8hr
wandb: Run `wandb offline` to turn off syncing.
wandb: Syncing run super-pond-51
wandb: ⭐ View project at https://wandb.ai/charleshong3-team/autocomp-tpu
```

W&B API Key

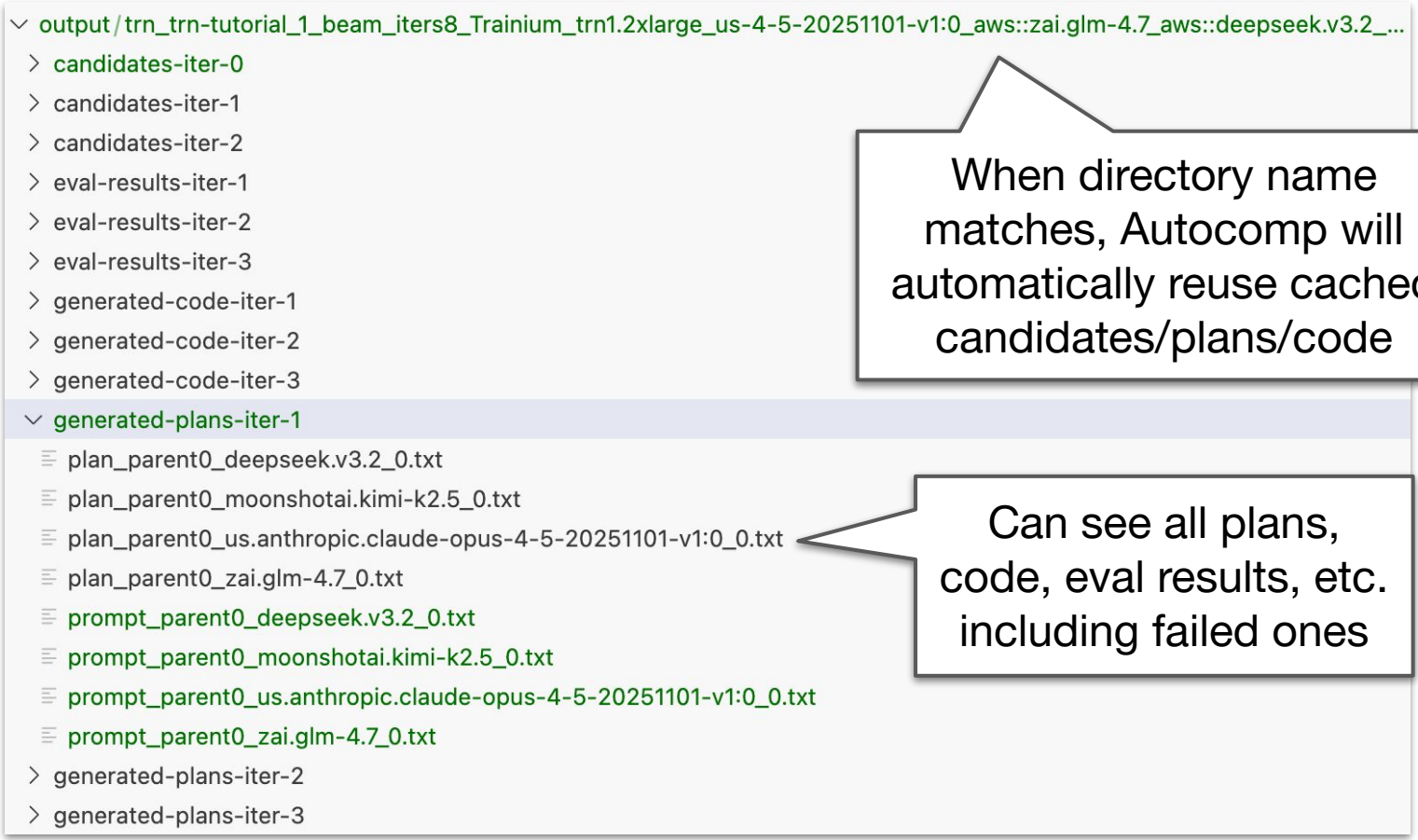

wandb.ai/home
Charles Hong
charleshong3-team-org
Home
Projects
View all
Search
charlesh.../autocomp-tpu
charlesh.../autocomp
Create a new project
Core
Registry
Launch
Inference
Profile
charleshong3
Teams
charleshong3-team
Create a team to collaborate
Starred projects
No starred projects
Star a project from your projects list and it will always show here.
Your recent projects
autocomp-tpu
Last updated 2 hours ago
50 runs
autocomp
Last updated 6 days ago
819 runs
You don't have any recent reports
Create your first report for autocomp-tpu
Reports are your brain dumps, eureka moments, and everything in between. They play nicely with everything else in W&B, so you can share, store or explain your findings.
Click to see how your charts from autocomp-tpu look in a report
Activity across your organization
Run fresh-forest-47 3 hours ago Finished
Run iconic-grass-40 7 hours ago Finished
Run neat-forest-21 2 days ago Finished
charleshong@berkeley.edu
Profile
User settings
Auto-refresh
Account
charleshong3-tea...
Academic · 4 users
Users
Teams
Activity
Service Accounts
Billing
Usage & Alerts
API keys
Settings
Invite users
Other accounts
Personal
Free

wandb

Lets you monitor status of ongoing runs and keep track of past runs



Autocomp artifacts (located at `<cwd>/output`)



CodeCandidate object (`output/candidates-iter-<i>`)

Stores full history of each candidate in beam, including:

- Ancestor candidates
- Latency
- Plan
- Code
- Models used to generate plan/code
- stdout/stderr (if returned by evaluation backend)

```
            # Load gamma and beta per-chunk
            i_p_param = nl.arange(1)[:, None]
            gamma_chunk = nl.load(gamma_reshaped[i_p_param, col_start + i_f_chunk])
            beta_chunk = nl.load(beta_reshaped[i_p_param, col_start + i_f_chunk])

            # Normalize: (x - mean) * inv_std
            centered = nl.subtract(input_chunk, mean_tile, mask=combined_mask)
            normalized = nl.multiply(centered, inv_std, mask=combined_mask)

            # Apply affine transformation: gamma * normalized + beta
            scaled = nl.multiply(normalized, gamma_chunk, mask=combined_mask)
            output_chunk = nl.add(scaled, beta_chunk, mask=combined_mask)

            # Store result
            nl.store(output_tensor[row_offset + i_p, col_start + i_f_chunk],
                     value=output_chunk, mask=combined_mask)

    return output_tensor
''',
score=1.446,
hw_feedback=[],
plan_gen_model='moonshotai.kimi-k2.5',
code_gen_model='us.anthropic.claude-opus-4-5-20251101-v1:0',
stdout='Latency: 1.446 ms (P99)\n',
stderr=''),
plan='''## Analysis of the Code

Looking at the current implementation, I can identify several inefficiencies:

1. **Redundant gamma/beta loads**: Inside the normalization loop, `gamma_chunk` and `beta_chunk` are loaded fr

2. **Multiple passes over data**: The code loads input data, stores it to a buffer, then reads it back for nor

3. **Unrolling structure is awkward**: The manual unrolling with `for unroll_idx in range(2)` creates a Python

4. **Gamma and beta are loaded per-chunk inside the row loop**: These parameters are the same for all rows, so

## Selected Optimization: #6 - Increase reuse by keeping data in SBUF across outer loop iterations

**Plan:**

The key insight is that `gamma_vector` and `beta_vector` are 1D parameters of shape `[num_cols]` that are **co
```

trn-tutorial problem 1

- Layernorm
- Ran as a quick test for 3 iterations

bash autocomp | 1_layernorm_ref.py | search.py M

autocomp > sols > trn-tutorial > 1_layernorm_ref.py > ...

```python
@nki.jit
def test(input_tensor, epsilon, gamma_vector, beta_vector):
  """Computes LayerNorm.
  """
  output_tensor = nl.ndarray(input_tensor.shape, dtype=input_tensor.dtype,
                             buffer=nl.shared_hbm)

  # Ensure that the shapes of tensors match
  assert input_tensor.shape[1] == gamma_vector.shape[0] == beta_vector.shape[0]

  # Generate tile indices for loading/storing data
  i_p_io = nl.arange(nl.tile_size.pmax)[:, None]
  i_f_io = nl.arange(input_tensor.shape[1])[None, :]
  i_p_param = nl.arange(1)[:, None]

  # Number of rows in the input tensor
  num_rows = input_tensor.shape[0]

  # Load gamma and beta, which will be reused across rows/tiles of input_tensor
  gamma_sb = nl.load(gamma_vector.reshape((1, gamma_vector.shape[0]))[i_p_param, i_f_io])
  beta_sb = nl.load(beta_vector.reshape((1, beta_vector.shape[0]))[i_p_param, i_f_io])

  # Broadcast the gamma and beta to match the dimensions of the tiles
  gamma_sb_bcast = gamma_sb.broadcast_to((nl.tile_size.pmax, gamma_vector.shape[0]))
  beta_sb_bcast = beta_sb.broadcast_to((nl.tile_size.pmax, beta_vector.shape[0]))

  # Tile partition dimension of the input tensor by nl.tile_size.pmax
  for i in nl.affine_range(math.ceil(input_tensor.shape[0]/nl.tile_size.pmax)):
    # Load input tile
    input_sb = nl.load(input_tensor[i * nl.tile_size.pmax + i_p_io, i_f_io],
                       mask=(i * nl.tile_size.pmax + i_p_io < num_rows))

    # Compute mean and variance
    mean = nl.mean(input_sb, axis=1)
    # Trick to calculate var with mean: mean(x^2) - mean(x)^2
    var = nl.mean(nl.square(input_sb), axis=1) - mean * mean

    # Normalize the input by shifting with the mean
    # and scaling with rsqrt of variance and epsilon
    shift_scale_tensor = (input_sb - mean) * nl.rsqrt(var + epsilon)

    # Scale the normalized tile using gamma and add beta
    output_sb = shift_scale_tensor * gamma_sb_bcast + beta_sb_bcast

    nl.store(output_tensor[i * nl.tile_size.pmax + i_p_io, i_f_io], value=output_sb,
             mask=(i * nl.tile_size.pmax + i_p_io < num_rows))

```

Defining Software Workloads

Generally, defined by agent/backend pair

`autocomp/sols/` - Unoptimized kernel code

- Started with `sols/trn-tutorial/1_layernorm_ref.py`
  - Automatically imported based on `prob_type` and `prob_id` in search.py

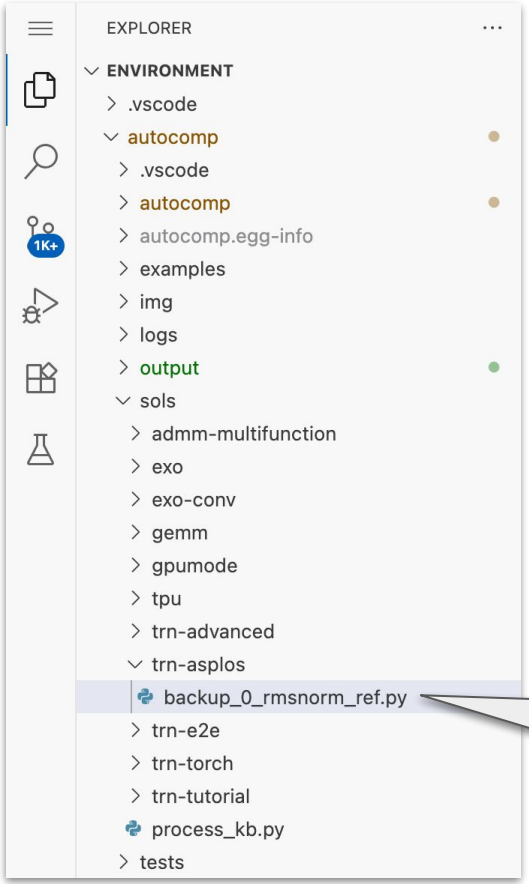
`autocomp/tests/` - Test harnesses for kernels

- Started with `sols/trn-tutorial/1_layernorm_test.py`

autocomp/autocomp/backend/trn/trn_eval.py

- Inlines sol into test to create runnable file

Add a new workload!



- If you have it, add the RMSNorm code generated by LLMLift as a new workload
- If you missed the LLMLift session, find the backup of the code and move it to the right place

Create:
`0_rmsnorm_ref.py`
Or rename:
`backup_0_rmsnorm_ref.py`

Optimize the new workload

bash autocomp | search.py M | 0_conv1d_ref.py U | 0_conv1d_test.py

autocomp > autocomp > search > search.py > main

```
789  def main():
790      # Select evaluation backend, LLM agent, and hardware config
791      backend_name = "trn"  # Options: "gemmini", "trn", "tpu", "kernelbench", "gpumode"
792      agent_name = "built:trn-nki1"  # Options: "gemmini", "trn", "cuda", "built:<name>", or a path to a built agent (for TPU v6e, use built:tpu-v6e)
793      simulator = None # "firesim" or "spike" if backend_name == "gemmini"; "gpumode-local" or "gpumode-cli" if backend_name == "gpumode"
794      # Hardware configuration
795      hw_config = TrnHardwareConfig("trn1.2xlarge")
796      # Examples:
797      # hw_config = TrnHardwareConfig("trn1.2xlarge")
798      # hw_config = GemminiHardwareConfig(pe_dim=16, spad_size_kb=256, acc_size_kb=64)
799      # hw_config = CudaHardwareConfig("NVIDIA L40S", "2.5.0", "12.4")
800      # hw_config = TpuHardwareConfig("v6e-1")
801
802      # Models are specified as "provider::mod
803      # Valid providers are "openai", "anthrop
804      # If no provider is specified, the provi
805      models = ["aws::us.anthropic.claude-opus                                                    shotai.kimi-k2.5"] # Models for planning
806      code_models = None # Models for code imp
807      metric = "latency"
808      search_strategy = "beam"
809      iterations = 5
810      prob_type = "trn-asplos" # see README.mo
811      prob_id = 0
```

Set:

```
iterations = 5
prob_type = “trn-asplos”
prob_id = 0
```

Run search

```
> python autocomp/search/search.py
```

Feel free to run other problems defined under `trn-tutorial`, `trn-advanced`

Experiment with iterations, beam size, dropout %, other parameters

What else can Autocomp do?

- Existing support for **Google TPU, NVIDIA GPUs, Berkeley Gemmini**
- Supports adding different types of **hardware feedback** (other than just latency) in the planning prompt
- Supports adding a **translation phase** (e.g. PyTorch to CUDA or NKI) before starting optimization
- Agent builder - pass in documentation for **your hardware target**, get an auto-built Autocomp agent!

Coming soon to Autocomp:

- Enhanced trace visualizer and Berkeley Saturn-RVV support
- Whatever feature you would like to contribute (e.g. deterministic search?)

Conclusion

We covered:

- How to build an agent
- How to build an evaluation backend
- How to set search parameters and run search
- How to look at search results using local artifacts and W&B
- How to add and run new workloads

Thank you for attending!

Acknowledgements

- Thank you fellow organizers and volunteers!
- Special thanks to AWS for the incredible support

Some Remaining Research Problems

- **More workloads, more hardware backends**
  - Different hardware requires different considerations. How to distill documentation into prompts, how to speed up evaluation, what abstraction level to optimize at (e.g. CUDA vs CuTe DSL vs cuTile vs Triton), how to profile/understand bottlenecks.
- **Beam search vs evolutionary search vs other search algorithms**
  - OpenEvolve results not strong compared to Autocomp - may need to do more hyperparameter tuning.
- **How exactly does code size affect code optimization performance?**
  - And assuming bigger code is harder to optimize, how do we decompose the problem into tractable subcomponents?
- **How can we combine inference-time search and prompt optimization/context engineering?**
  - Naively combining them creates an exponentially growing search space.
  - AlphaEvolve does some prompt evolution, but with limited effect. Needs more principled study.
- **How can we use LLM search to improve compilers and libraries?**
  - Evaluation becomes more expensive/ambiguous when kernels are need to be parameterizable for different shapes, datatypes, etc.
  - Do compilers/libraries still make sense, or should we just decompose every workload into AI-generated, hyper-specialized kernels?