AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms

Lain

Overview

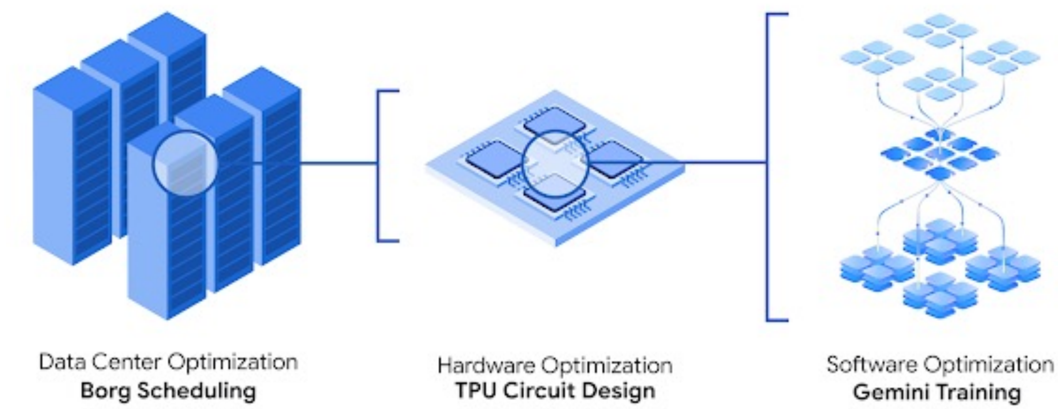
- LLM-guided evolution for code optimization and algorithm design
- Previous work: FunSearch (2023)
- AlphaEvolve (2025)
  - Inspired by MAP-Elites algorithm and island-based population models
  - Distributed evaluation (across multiple fitness metrics)
  - Evolve entire file; search across codebase
  - No explicit evolutionary operations!
    - LLMs serve as evolutionary operators (no need to manually design operators, but lacks interpretability)
- Results

```diff
  def __init__(self, mode, init_rng, config, hypers):
    self.hypers = hypers
    super().__init__(mode=mode, init_rng=init_rng, config=config)

  def _get_optimizer(self) -> optax.GradientTransformation:
    """Returns optimizer."""
+   b1 = 0.9
+   b2 = 0.999
    return optax.adamw(
-       self.hypers.learning_rate, weight_decay=self.hypers.weight_decay
+       self.hypers.learning_rate, weight_decay=self.hypers.weight_decay, b1=b1,
+   b2=b2

    )

  def _get_init_fn(self) -> jax.nn.initializers.Initializer:
    """Returns initializer function."""
    scale = self.hypers.init_scale
    # Initialize with a smaller scale to encourage finding low-rank solutions
-   return initializers.normal(0 + 1j * 0, scale * 0.1, jnp.complex64)
+   return initializers.normal(0 + 1j * 0, scale * 0.2, jnp.complex64)


- def _linear_schedule(self, global_step, start: float = 0.0, end: float = 1.0):
+ def _linear_schedule(self, global_step, start: float = 0.0, end: float = 0.0):

    frac = 1 - global_step / self.config.training_steps
    return (start - end) * frac + end

  @functools.partial(jax.jit, static_argnums=0)
  def _update_func(
      self,
      decomposition: tuple[jnp.ndarray, jnp.ndarray, jnp.ndarray],
      opt_state: optax.OptState,
      global_step: jnp.ndarray,
      rng: jnp.ndarray,
  ) -> tuple[
      tuple[jnp.ndarray, jnp.ndarray, jnp.ndarray],
      optax.OptState,
      jnp.ndarray,
  ]:
    """A single step of decomposition parameter updates."""
    # Compute loss and gradients.
    loss, grads = jax.value_and_grad(
        lambda decomposition, global_step, rng: jnp.mean(
            self._loss_fn(decomposition, global_step, rng)
        )
    )(decomposition, global_step, rng)
    # When optimizing real-valued functions of complex variables, we must take
    # the conjugate of the gradient.
    grads = jax.tree_util.tree_map(lambda x: x.conj(), grads)
    # Gradient updates.
    updates, opt_state = self.opt.update(grads, opt_state, decomposition)
    decomposition = optax.apply_updates(decomposition, updates)

    # Add a small amount of gradient noise to help with exploration
    rng, g_noise_rng = jax.random.split(rng)
```

Iteration 15

Why do we care?

- Evolution can drive algorithmic and scientific discoveries
- **Issues:**
  - Requires hard-coding genetic operators
  - Evaluation is expensive
- It is difficult to scale evolutionary approaches!
- **AlphaEvolve:** LLM-guided evolutionary search that scales beyond toy functions with multi-objective constraints



FunSearch (2023)

- Evolves single Python function (≤ 20 LOC)
- Single fitness metric; millions of LLM calls
- Evaluation: ~20 min on one CPU per candidate

FunSearch

Evaluation

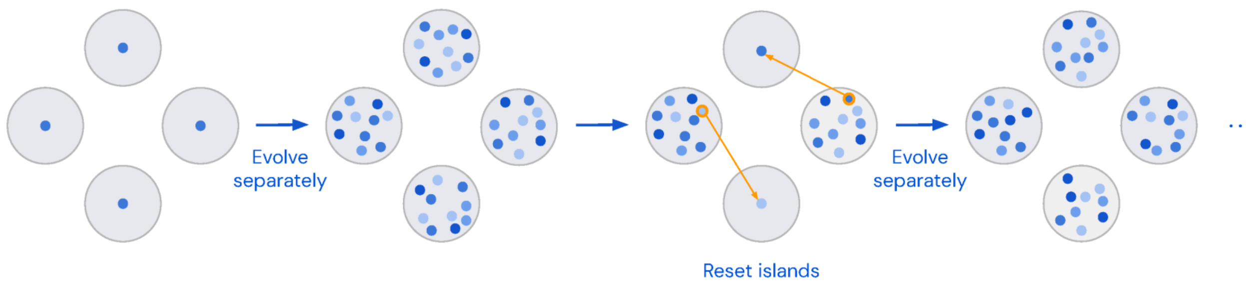
FunSearch [83]	*AlphaEvolve*
evolves single function	evolves entire code file
evolves up to 10-20 lines of code	evolves up to hundreds of lines of code
evolves code in Python	evolves any language
needs fast evaluation (≤ 20min on 1 CPU)	can evaluate for hours, in parallel, on accelerators
millions of LLM samples used	thousands of LLM samples suffice
small LLMs used; no benefit from larger	benefits from SOTA LLMs
minimal context (only previous solutions)	rich context and feedback in prompts
optimizes single metric	can simultaneously optimize multiple metrics

Programs database

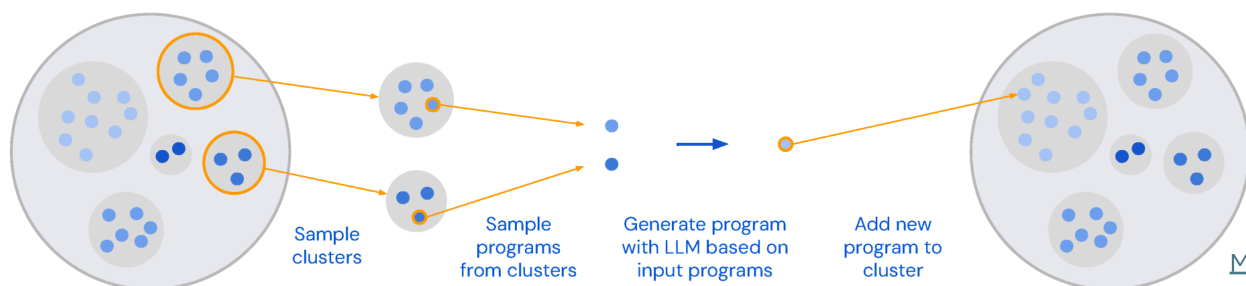
[Mathematical discoveries from program search with large language models (Romera-Parades et al. 2023)]

LLM-Guided Evolution

- Islands of programs evolved separately:

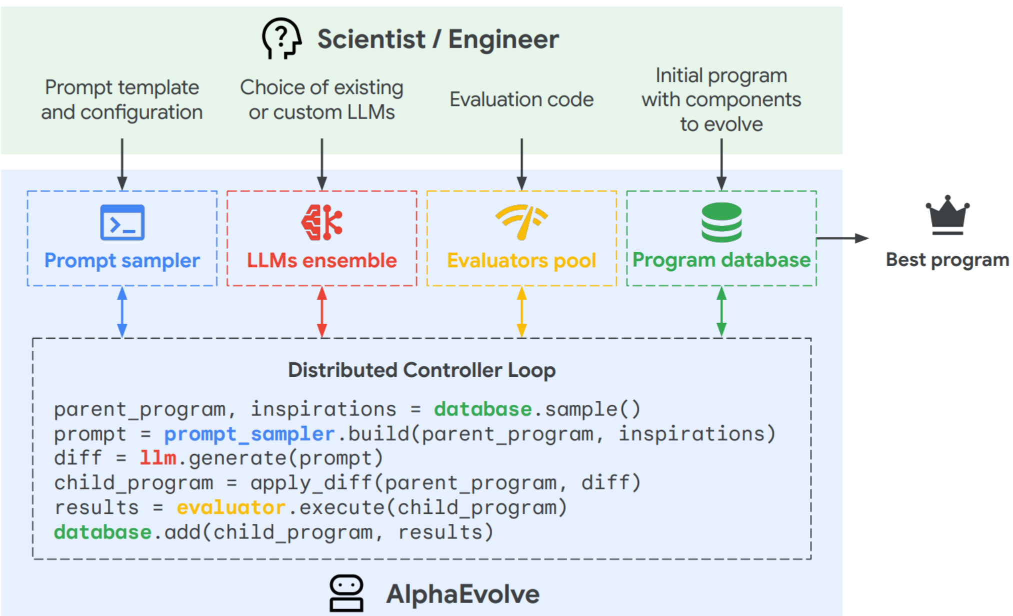


- Generating new offspring:



[Mathematical discoveries from program search with large language models (Romera-Parades et al. 2023)]

AlphaEvolve



- Evolve entire files in *any* language
- **Database:** Pool of candidate programs
  - Every candidate is a *full source file* with versioned metadata and vector of fitness scores
- LLM **samplers** = mutation/crossover engine
  - Sampler prompt = top-k elites + diff context + task spec
- **Evaluators:** distributed evaluation of candidates
  - Supports *a vector of fitness metrics*: ⟨performance, resources, provability…⟩

Evolutionary Operations

- No *explicit* evolutionary operations – LLMs drive evolution
- **Sampling step:** pick **2-3 parent files** (often from different islands) + diagnostic feedback (why they're good/bad)
  - Prompts include top programs for **each** metric
- **LLM output:**
  - *Patch*
  - *Full file* rewrite
- **Critic pass:** fast LLM or regex rules refuse certain patches

LLM Input and Output

Initial solution

```
# EVOLVE-BLOCK START
"""Image classification experiment in jaxline."""

import jax
...
# EVOLVE-BLOCK-END

...

# EVOLVE-BLOCK-START
class ConvNet(hk.Module):
  def __init__(self, num_classes): ...
  def __call__(self, inputs, is_training): ...


def sweep():
  return hyper.zipit([...])
# EVOLVE-BLOCK-END
...

def evaluate(eval_inputs) -> dict[str, float]:
  ...
  return metrics
```

Evaluation function

Task description

```
Act as an expert software developer. Your task is to iteratively
improve the provided codebase. [...]
```

Elite programs sampled from different clusters + fitness scores

```
- Prior programs

Previously we found that the following programs performed well
on the task at hand:

top_1_acc: 0.796; neg_eval_log_loss: 0.230; average_score: 0.513

"""Image classification experiment in jaxline."""
[...]
class ConvNet(hk.Module):
  """Network."""

  def __init__(self, num_channels=32, num_output_classess=10):
    super().__init__()
    self._conv1 = hk.Conv2D(num_channels, kernel_shape=3)
    self._conv2 = hk.Conv2D(num_channels * 2, kernel_shape=3)
    self._conv3 = hk.Conv2D(num_channels * 4, kernel_shape=3)
    self._logits_module = hk.Linear(num_output_classes)
[...]
```

```
- Current program

Here is the current program we are trying to improve (you will
need to propose a modification to it below).

top_1_acc: 0.862; neg_eval_log_loss: 0.387; average_score: 0.624

"""Image classification experiment in jaxline."""
[...]
class ConvNet(hk.Module):
  """Network."""

  def __init__(self, num_channels=32, num_output_classes=10):
    super().__init__()
    self._conv1 = hk.Conv2D(num_channels, kernel_shape=3)
    self._block1 = ResNetBlock(num_channels)
    self._block2 = ResNetBlock(num_channels * 2, stride=2)
    self._block3 = ResNetBlock(num_channels * 4, stride=2)
    self._logits_module = hk.Linear(num_output_classes)
[...]
```

Prompt to guide evolution

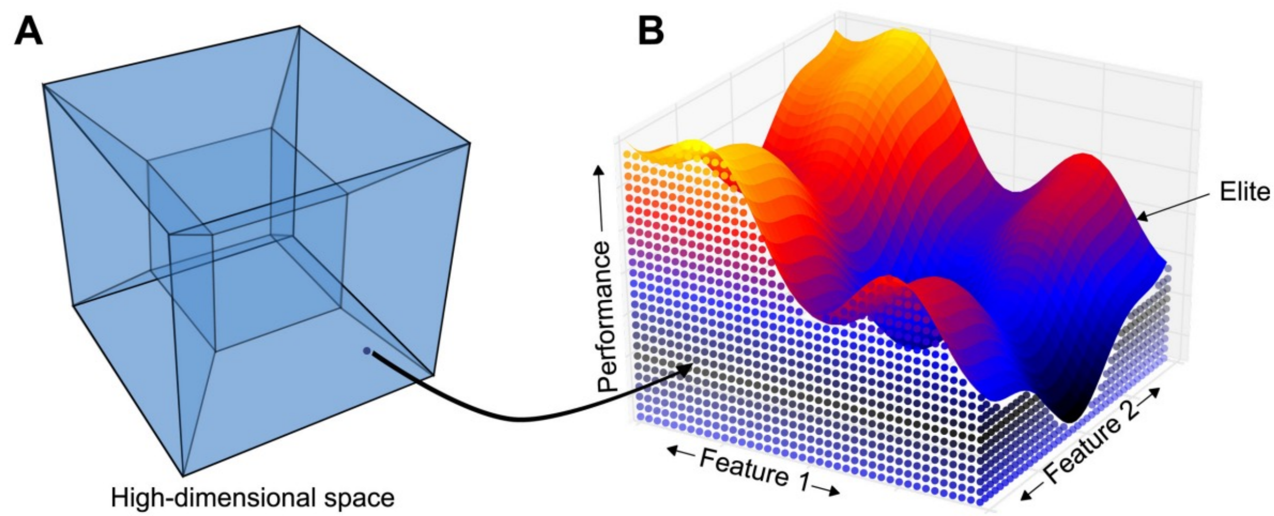
```
SEARCH/REPLACE block rules:
[...]

Make sure that the changes you propose are consistent with each
other. For example, if you refer to a new config variable
somewhere, you should also propose a change to add that
variable.

Example:
[...]

Task
Suggest a new idea to improve the code that is inspired by your
expert knowledge of optimization and machine learning.

Describe each change with a SEARCH/REPLACE block.
```

LLM output:

```
The current model uses a simple ResNet architecture with only
three ResNet blocks. We can improve its performance by
increasing the model capacity and adding regularization. This
will allow the model to learn more complex features and
generalize better to unseen data. We also add weight decay to
the optimizer to further regularize the model and prevent
overfitting. AdamW is generally a better choice than Adam,
especially with weight decay.


<<<<<<< SEARCH
    self._block1 = ResNetBlock(num_channels)
    self._block2 = ResNetBlock(num_channels * 2, stride=2)
    self._block3 = ResNetBlock(num_channels * 4, stride=2)
=======
    self._block1 = ResNetBlock(num_channels)
    self._block2 = ResNetBlock(num_channels, stride=1)
    self._block3 = ResNetBlock(num_channels * 2, stride=2)
    self._block4 = ResNetBlock(num_channels * 2, stride=1)
    self._block5 = ResNetBlock(num_channels * 4, stride=2)
    self._block6 = ResNetBlock(num_channels * 4, stride=1)
>>>>>>> REPLACE
<<<<<<< SEARCH
  def optimizer(self, learning_rate):
    return optax.adam(learning_rate)
=======
  def optimizer(self, learning_rate):
    return optax.adamw(learning_rate, weight_decay=1e-4)
>>>>>>> REPLACE
```

Exploration vs. Exploitation

- **Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) *within* each island**
  - MAP-Elites: illuminates the fitness potential of each area of the feature space (vector of fitness of metrics)



[Illuminating search spaces by mapping elites (Mouret and Clune 2015)]

Multi-Objective Distributed Evaluation

- **Selection rule within an island:**
  - Keep *elites* per MAP cell
  - Maintain a Pareto front on the primary/secondary objectives
  - When wiping islands, compute a scalar “island score” = weighted sum of top-k elites; wipe the worst

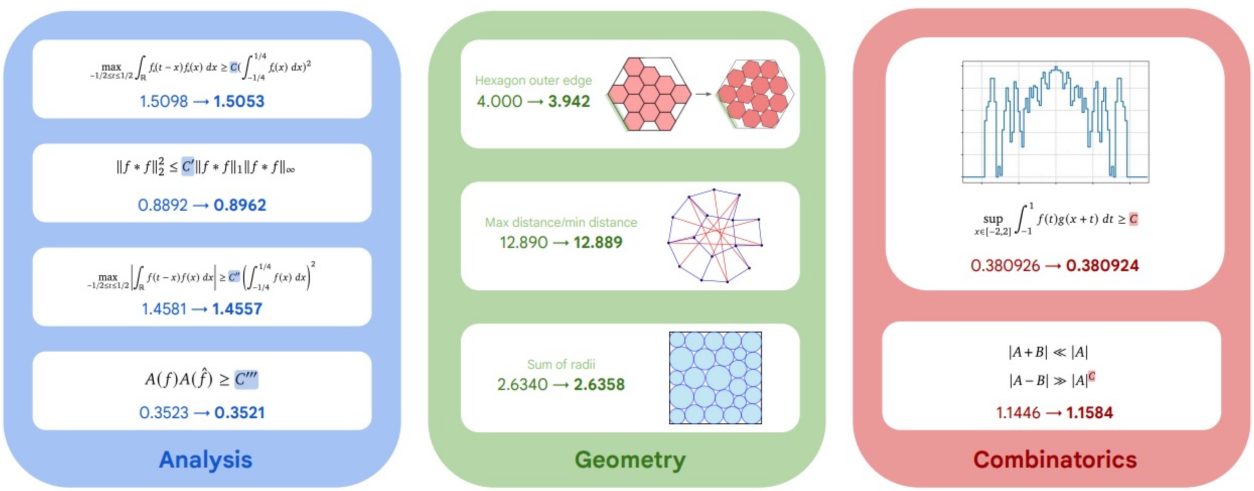
- Evaluations are performed *asynchronously*: sandbox executes the code on real hardware, returns fitness vector
  - ~1000 TPUs/GPUs in parallel

```diff
  def __init__(self, mode, init_rng, config, hypers):
    self.hypers = hypers
    super().__init__(mode=mode, init_rng=init_rng, config=config)

  def _get_optimizer(self) -> optax.GradientTransformation:
    """Returns optimizer."""
+   b1 = 0.9
+   b2 = 0.999
    return optax.adamw(
-       self.hypers.learning_rate, weight_decay=self.hypers.weight_decay
+       self.hypers.learning_rate, weight_decay=self.hypers.weight_decay, b1=b1,
+  b2=b2

    )

  def _get_init_fn(self) -> jax.nn.initializers.Initializer:
    """Returns initializer function."""
    scale = self.hypers.init_scale
    # Initialize with a smaller scale to encourage finding low-rank solutions
-   return initializers.normal(0 + 1j * 0, scale * 0.1, jnp.complex64)
+   return initializers.normal(0 + 1j * 0, scale * 0.2, jnp.complex64)


- def _linear_schedule(self, global_step, start: float = 0.0, end: float = 1.0):
+ def _linear_schedule(self, global_step, start: float = 0.0, end: float = 0.0):

    frac = 1 - global_step / self.config.training_steps
    return (start - end) * frac + end

  @functools.partial(jax.jit, static_argnums=0)
  def _update_func(
      self,
      decomposition: tuple[jnp.ndarray, jnp.ndarray, jnp.ndarray],
      opt_state: optax.OptState,
      global_step: jnp.ndarray,
      rng: jnp.ndarray,
  ) -> tuple[
      tuple[jnp.ndarray, jnp.ndarray, jnp.ndarray],
      optax.OptState,
      jnp.ndarray,
  ]:
    """A single step of decomposition parameter updates."""
    # Compute loss and gradients.
    loss, grads = jax.value_and_grad(
        lambda decomposition, global_step, rng: jnp.mean(
            self._loss_fn(decomposition, global_step, rng)
        )
    )(decomposition, global_step, rng)
    # When optimizing real-valued functions of complex variables, we must take
    # the conjugate of the gradient.
    grads = jax.tree_util.tree_map(lambda x: x.conj(), grads)
    # Gradient updates.
    updates, opt_state = self.opt.update(grads, opt_state, decomposition)
    decomposition = optax.apply_updates(decomposition, updates)

    # Add a small amount of gradient noise to help with exploration
    rng, g_noise_rng = jax.random.split(rng)
```

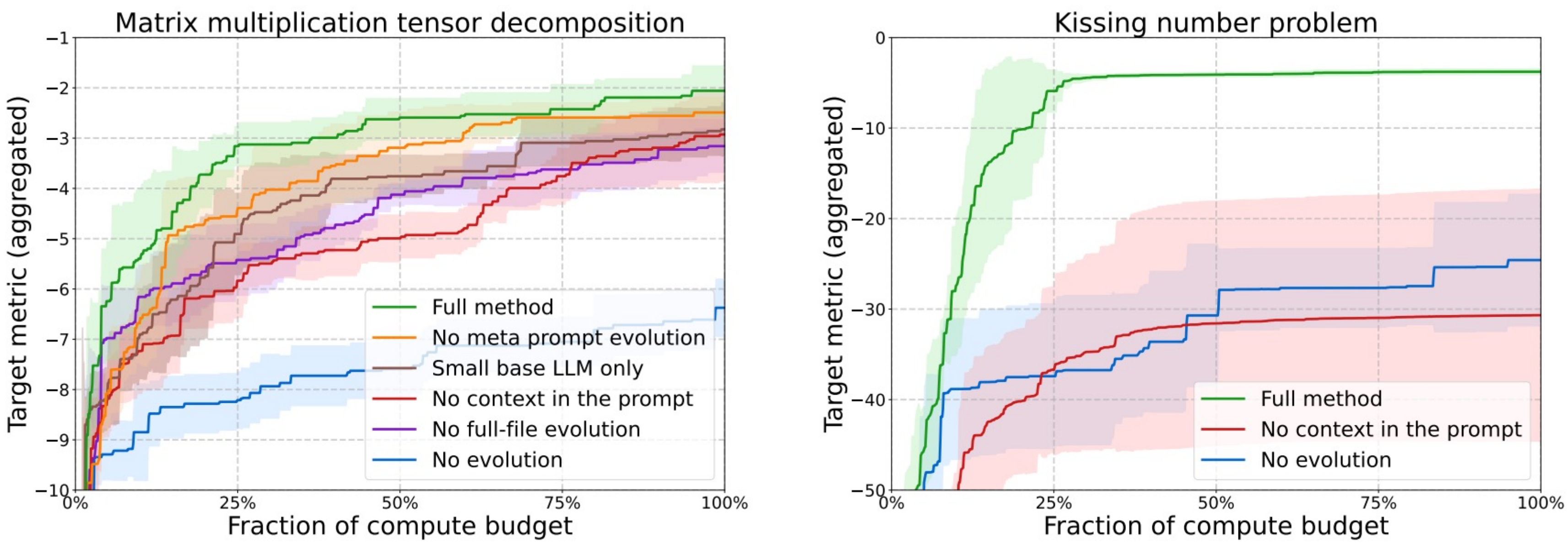
Iteration 15

Results

- Rank-48 algorithm for 4 × 4 complex MM (first < 49 in 56 yrs)
- 23% speed-up for Gemini tiling kernel; 1% end-to-end training gain
- 32% FlashAttention kernel speed-up in XLA IR
- Applied to 50+ math problems; AlphaEvolve matched optimal constructions in 75% of them and surpassed SOTA in 25%:



Ablations


Matrix multiplication tensor decomposition
Target metric (aggregated)
Fraction of compute budget
Full method
No meta prompt evolution
Small base LLM only
No context in the prompt
No full-file evolution
No evolution
−1
−2
−3
−4
−5
−6
−7
−8
−9
−10
0%
25%
50%
75%
100%
Kissing number problem
Target metric (aggregated)
Fraction of compute budget
Full method
No context in the prompt
No evolution
0
−10
−20
−30
−40
−50
0%
25%
50%
75%
100%

Takeaways

- **LLM-guided evolution** removes manual operator design
  - Any LLM can be used to drive evolution, performance scales with better underlying LLM
- First evolutionary agent for multi-objective optimization + asynchronous evaluation of entire code files in any language
- Promising template for any domain with *executable* evaluators
- **Limitations:** LLM-driven evolution isn't interpretable; compute intensive