A Results

In Table 4, we show a summary of the results of AlgoTuner for each of the four frontier models tests. In Table 5, we detail the per-task timings for every model and task.

Table 4: AlgoTuner speedup when using each LM, with a budget of \$0.5 for each task. Speedup percentage is calculated as the percentage of tasks for which AlgoTuner gets at least a 1.1× speedup.

	o4-mini-high	R1	Claude 3.7 Sonnet	Gemini 2.5 Pro
Pct. of Tasks Sped Up	53.3%	42.5%	34.2%	34.2%

Table 5: Per task speedup for AlgoTuner, using four frontier LMs. Speedup is calculated as the ratio between the reference solve function's time and the LM-generated solve function's time.

Task	o4-mini-high	R1	Claude 3.7 Sonnet	Gemini 2.5 Pro
`affine_transform_2d`	1.00	1.00	1.00	1.00
`articulation_points`	4.75	1.00	1.45	1.00
`base64_encoding`	1.01	1.00	1.00	1.00
`btsp`	74.47	4.02	3.23	6.74
`capacitated_facility_location`	8.16	8.19	1.00	1.00
`chacha_encryption`	1.00	1.00	1.00	1.00
`channel_capacity`	1.00	1.73	1.71	1.57
`chebyshev_center`	1.78	2.21	1.23	1.13
`cholesky_factorization`	2.17	1.00	1.00	1.00
`clustering_outliers`	1.11	1.00	1.06	1.18
`communicability`	217.10	162.59	183.45	166.40
`convex_hull`	2.18	1.00	1.00	1.00
`convex_quadratic_check`	96.58	2.52	6.05	6.62
`convolve2d_full_fill`	272.12	229.80	227.84	229.30
`convolve_1d`	1.00	1.00	1.00	1.00
`correlate_1d`	1.62	1.00	1.54	1.00
`count_connected_components`	6.86	1.00	3.92	3.44
`count_riemann_zeta_zeros`	1.00	1.00	1.02	1.00
`crew_pairing`	1.00	1.00	1.00	1.00
`cumulative_simpson_multid`	1.00	1.00	1.00	1.00
`cyclic_independent_set`	1.00	1.00	1.00	1.00
`determinant_matrix_exponential`	416.78	1.00	277.06	395.17
`dijkstra_from_indices`	1.64	1.00	1.00	1.00
`discrete_log`	1.28	1.00	1.00	1.00
`dynamic_assortment_planning`	2.45	1.08	1.00	28.82
`earth_movers_distance`	1.00	1.17	1.09	1.00
`edge_expansion`	25.22	1.00	1.00	1.00
`efficiency`	71.63	2.76	1.07	1.11
`eigenvalues_complex`	1.47	1.52	1.47	1.45
`eigenvalues_real`	2.43	2.44	2.39	2.41
`eigenvectors_complex`	1.03	1.00	1.00	1.02
`eigenvectors_real`	1.03	1.06	1.01	1.02
`elementwise_integration`	1.01	1.00	1.00	1.00
`feedback_controller_design`	119.25	1.00	1.00	1.00
`fft_real_scipy_fftpack`	2.98	2.66	1.00	1.00
`generalized_eigenvalues_complex`	3.84	2.04	2.04	2.02
`generalized_eigenvalues_real`	2.55	2.98	2.52	1.00
`generalized_eigenvectors_complex`	2.84	1.06	1.06	1.05
`generalized_eigenvectors_real`	1.72	1.71	1.52	1.15
`graph_coloring`	27.19	1.18	1.00	1.08
`graph_isomorphism`	56.73	19.68	24.62	1.98
`graph_laplacian`	1.48	1.00	1.08	1.57

Continued on next page

Table 5 – continued from previous page

Task	o4-mini-high	R1	Claude 3.7 Sonnet	Gemini 2.5 Pro
gzip_compression	1.00	1.00	1.00	1.00
job_shop_scheduling	1.42	1.75	1.40	1.48
kalman_filter	71.08	2.96	1.00	1.00
kcenters	1.00	1.00	1.00	1.00
kd_tree	1.23	1.20	1.00	1.01
kernel_density_estimation	1.00	1.00	1.00	1.00
kmeans	32.30	1.00	1.00	1.02
ks_test_2samp	1.00	1.03	1.03	1.00
l0_pruning	3.98	3.15	1.00	2.68
l1_pruning	2.94	2.84	1.27	1.40
lasso	1.00	1.00	1.00	1.00
linear_system_solver	1.10	1.12	1.09	1.09
lp_box	2.26	14.19	1.53	2.25
lp_centering	1.00	1.00	1.00	1.00
lqr	1.05	1.00	1.10	1.00
lu_factorization	1.00	1.00	1.00	1.00
markowitz	1.00	1.00	1.00	1.01
matrix_exponential	1.01	1.00	3.79	1.00
matrix_multiplication	1.08	1.06	1.09	1.08
matrix_sqrt	1.02	1.00	1.04	1.00
max_clique	21.15	1.26	3.80	2.66
max_common_subgraph	20.94	1.00	1.56	1.47
max_flow_min_cost	3.04	20.46	2.09	1.00
max_independent_set	1.00	1.63	1.00	1.51
max_weighted_independent_set	71.37	9.55	1.04	1.00
min_weight_assignment	1.42	1.25	1.00	1.00
minimum_spanning_tree	30.88	3.94	33.52	12.50
minimum_volume_ellipsoid	1.00	1.00	1.00	1.00
multi_dim_knapsack	1.52	1.48	1.37	1.00
nmf	1.00	1.00	1.00	1.00
ode_fitzhughnagumo	1.00	1.00	1.00	1.00
ode_hires	10.72	5.58	1.00	5.49
ode_lorenz96_nonchaotic	1.00	1.00	1.00	1.00
ode_nbodyproblem	1.00	1.00	1.00	1.00
ode_stiff_robertson	18.79	1.00	1.66	4.89
ode_stiff_vanderpol	1.00	1.00	1.00	1.00
odr	1.01	1.00	1.00	1.00
pagerank	1.00	1.00	1.00	1.57
pca	3.77	1.00	3.60	1.54
pde_burgers1d	1.00	1.00	1.00	1.00
pde_heat1d	1.00	1.00	1.00	1.00
polynomial_mixed	1.02	1.01	1.00	1.01
polynomial_real	23.60	142.29	1.00	1.00
portfolio_optimization_cvar	1.00	10.72	8.25	1.15
procrustes	1.23	1.04	1.03	1.04
psd_cone_projection	10.16	8.78	9.30	7.34
qp	1.75	1.76	1.01	1.00
qr_factorization	1.98	1.17	1.16	1.00
quantile_regression	1.27	1.00	1.00	1.03
queens_with_obstacles	1.42	2.40	1.00	1.38
queuing	76.82	6.72	1.00	1.00
qz_factorization	1.00	1.01	1.01	1.01
randomized_svd	5.66	3.84	1.26	1.98
rbf_interpolation	1.00	1.00	1.00	1.00
rectanglepacking	1.67	7.54	1.00	1.55
robust_linear_program	1.00	1.07	458.28	1.00
rotate_2d	1.00	1.00	1.00	1.00
set_cover_conflicts	31.03	4.97	4.49	4.92
set_cover	1.84	1.35	1.44	1.34
sha256_hashing	1.00	1.00	1.00	1.00
shift_2d	1.00	1.00	1.00	1.00

Continued on next page

Table 5 – continued from previous page

Task	o4-mini-high	R1	Claude 3.7 Sonnet	Gemini 2.5 Pro
shortest_path_dijkstra	2.31	2.50	53.17	2.46
sparse_eigenvectors_complex	1.00	1.00	1.00	1.00
spectral_clustering	1.00	68.34	1.00	1.00
stable_matching	1.05	1.00	1.00	1.00
toeplitz_solver	1.00	1.00	1.00	1.00
tsp	1.00	1.27	1.00	1.09
two_eigenvalues_around_0	1.41	1.65	1.01	1.58
unit_simplex_projection	1.92	1.05	1.05	1.05
vector_quantization	1.00	1.00	1.00	1.00
vectorized_newton	1.00	1.00	1.00	1.00
vehicle_routing	1.00	1.17	1.00	1.00
vehicle_routing_circuit	1.00	1.47	1.01	1.00
vertex_cover	12.04	1.07	12.52	1.23
vertex_cover_cpsat	14.36	1.44	1.00	1.19
wasserstein_dist	8.15	7.52	1.00	6.78
water_filling	418.00	346.88	109.62	81.21
zoom_2d	1.00	1.00	1.00	1.00

C AlgoTuner Agent Setup

Initial Prompt. The LM receives an initial message, consisting of general instructions on how to use the system (see §C.1), Numba (Lam et al., 2015), Dask (Rocklin, 2015), and Cython (Behnel et al., 2011) (for a full list see Appendix E). Additionally, the LM is given the task's description, which includes input and output descriptions and examples, as well as the task's `solve` and `is_solution` functions. In essence, everything apart from the problem generating function is shown to the LM.

Using the Interface. The goal of the LM is to write a `Solver` class with a `solve()` function that takes problem instances and produces a correct output. To do this, the LM sends messages that must consist of exactly one thought and one command (Yao et al., 2023). Responses given by the system always start with a budget status, for example: `You have so far sent 3 messages, and used up $0.08. You have $0.42 remaining`. We use the LiteLLM (BerriAI, 2025) API to access all models used. Each model is limited to a budget of $0.50 per task, and is continuously prompted until its budget runs out. The budget includes both input and output tokens. Where applicable, we set the temperature to 0 and the `top_p` parameter to 0.95.

Commands. We detail the available system commands in Table 6. Following Yang et al. (2024) after an edit command is used the modified code is ran through a linter. If the linter raises errors, the code is reverted, and the linter errors are sent back to the LM. When there are no linter errors, the code is evaluated on 100 training samples, with results sent back to the LM. When there are runtime errors, those are also sent back to the LM. When there are no runtime errors, the performance score, along with average evaluation time is reported back to the LM. If the performance score reached is better than any score previously reached, the code state is saved.

Table 6: Available interface commands.

Command	Description
edit	Replace a range of lines in a file with new content. Can create new files.
delete	Remove a range of lines from a file.
ls	List all files in the current working directory.
view_file	Display 100 lines of a file from a specified start line.
revert	Revert all files to the best-performing version of the code.
reference	Get the reference solve's solution for a given input
eval	Evaluate the current solve function on the 100 training instances and report results.
eval_input	Run the solver on a given input and compare with the oracle.
profile	Profile the performance of the solve method on a given input.
profile_lines	Profile specified lines on a given input.

Message History. To manage conversational context within token limits, we truncate the messages send to the LM in the following manner: The initial system prompt and the full content of the most recent five user and five assistant turns are always sent, following (Yang et al., 2024). Messages older than these are truncated to the first 100 characters. If the total token count still exceeds the model's limit, these older, content-truncated messages are progressively dropped starting from the oldest and are replaced by a single placeholder message indicating the truncation is inserted after the system prompt to signal the discontinuity.

C.1 Initial Prompt

We show the initial prompt given to the language model. The prompt we use is loosely modeled after the one used in SWE-Agent (Yang et al., 2024). For each task, we add a task description and the reference solver implementation (see below).

```
SETTING:
You're an autonomous programmer tasked with solving a specific problem. You are to use the commands defined below to accomplish this task. Every message you send incurs a cost--you will be informed of your usage and remaining budget by the system.
You will be evaluated based on the best-performing piece of code you produce, even if the final code doesn't work or compile (as long as it worked at some point and achieved a score, you will be eligible).
Apart from the default Python packages, you have access to the following additional packages:
 - cryptography
 - cvxpy
 - cython
 - dask
 - diffrax
 - ecos
 - faiss-cpu
 - hdbscan
 - highspy
 - jax
 - networkx
 - numba
 - numpy
 - ortools
 - pandas
 - pot
 - pulp
 - pyomo
 - python-sat
 - scikit-learn
 - scipy
 - sympy
 - torch

YOUR TASK:
Your objective is to define a class named `Solver` in `solver.py` with a method:
```
class Solver:
    def solve(self, problem, **kwargs) -> Any:
        """Your implementation goes here."""
        ...
```

IMPORTANT: Compilation time of your init function will not count towards your function's runtime.

This `solve` function will be the entrypoint called by the evaluation harness. Strive to align your class and method implementation as closely as possible with the desired performance criteria.
For each instance, your function can run for at most 10x the baseline runtime for that instance. Strive to have your implementation run as fast as possible, while returning the same output as the baseline
```

````
    function (for the same given input). Be creative and optimize your
    approach!

Your messages should include a short thought about what you should do,
    followed by a _SINGLE_ command. The command must be enclosed within ```
     and ```, like so:
<Reasoning behind executing the command>
```
<command>
```

IMPORTANT: Each set of triple backticks (```) must always be on their own
    line, without any other words or anything else on that line.

Here are the commands available to you. Ensure you include one and only one
     of the following commands in each of your responses:
- `edit`: Replace a range of lines with new content in a file. This is how
    you can create files: if the file does not exist, it will be created.
    Here is an example:
  ```
  edit
  file: <file_name>
  lines: <start_line>-<end_line>
  ---
  <new_content>
  ---
  ```

  The command will:
  1. Delete the lines from <start_line> to <end_line> (inclusive)
  2. Insert <new_content> starting at <start_line>
  3. If both <start_line> and <end_line> are 0, <new_content> will be
    prepended to the file

  Example:
  edit
  file: solver.py
  lines: 5-7
  ---
  def improved_function():
      print("Optimized solution")
  ---
- `ls`: List all files in the current working directory.
- `view_file <file_name> [start_line]`: Display 100 lines of `<file_name>`
    starting from `start_line` (defaults to line 1).
- `revert`: Revert the code to the best-performing version thus far.
- `baseline <string>`: Query the baseline solver with a problem and receive
     its solution. If the problem's input is a list, this command would
    look like:
  ```
  baseline [1,2,3,4]
  ```
- `eval_input <string>`: Run your current solver implementation on the
    given input. This is the only command that shows stdout from your
    solver along with both solutions. Example:
  ```
  eval_input [1,2,3,4]
  ```
- `eval`: Run evaluation on the current solution and report the results.
````

- `delete`: Delete a range of lines from a file using the format:
```
delete
file: <file_name>
lines: <start_line>-<end_line>

The command will delete the lines from <start_line> to <end_line> (inclusive)

Example:
delete
file: solver.py
lines: 5-10
```
- `profile <filename.py> <input>`: Profile your currently loaded solve method's performance on a given input. Shows the 25 most time-consuming lines. Requires specifying a python file (e.g., `solver.py`) for validation, though profiling runs on the current in-memory code.
  Example:
```
profile solver.py [1, 2, 3]
```

- `profile_lines <filename.py> <line_number1, line_number2, ...> <input>`: Profiles the chosen lines of the currently loaded code on the given input. Requires specifying a python file for validation.
  Example:
```
profile_lines solver.py 1,2,3 [1, 2, 3]
```

TIPS:
After each edit, a linter will automatically run to ensure code quality. If there are critical linter errors, your changes will not be applied, and you will receive the linter's error message. Typically, linter errors arise from issues like improper indentation--ensure your edits maintain proper code formatting.
Cython Compilation: Edits creating or modifying Cython (`.pyx`) files will automatically trigger a compilation attempt (requires a `setup.py`). You will be notified if compilation succeeds or fails. If it fails, the edit to the `.pyx` file will be automatically reverted.
You have 8 CPU cores at your disposal, you can use them for parallelization.

If the code runs successfully without errors, the in-memory 'last known good code' will be updated to the new version. Following successful edits, you will receive a summary of your `solve` function's performance compared to the baseline.
If you get stuck, try reverting your code and restarting your train of thought.
Do not put an if __name__ == "__main__": block in your code, as it will not be ran (only the solve function will).
Keep trying to better your code until you run out of money. Do not stop beforehand!

GOALS:
Your primary objective is to optimize the `solve` function to run as as fast as possible, while returning the optimal solution.

```
You will receive better scores the quicker your solution runs, and you will
     be penalized for exceeding the time limit or returning non-optimal
    solutions.

Below you find the description of the task you will have to solve. Read it
    carefully and understand what the problem is and what your solver
    should do.
```

```
<task/description.txt>
```

```
Here is the baseline which you will be graded against. Your task is to
    write a function that produces the same output, in less time.
```

```
<task.solve>
```

```
This function will be used to check if your solution is valid for a given
    problem. If it returns False, it means the solution is invalid:
```

```
<task.is_solution>
```

C.2 AlgoTuner Interface Feedback

In this section, we show the type and style of feedback given back to the LM (within the running of AlgoTuner) in three different scenarios.

C.2.1 Successful Edit and Evaluation

LM Message

````
I'll create our solver function in solver.py which transforms the generalized
    eigenvalue problem into a standard one using Cholesky decomposition and
    then solves it using NumPy's eigenvalue routines.
```
edit
file: solver.py
lines: 1-100
---
import numpy as np
from typing import List, Tuple

class Solver:
    @staticmethod
    def solve(problem: Tuple[List[List[float]], List[List[float]]]) -> Tuple[
    List[float], List[List[float]]]:
        """
        Solve the generalized eigenvalue problem A x = lambda B x.

        Given:
          - A: a symmetric matrix,
          - B: a symmetric positive definite matrix,
        this function computes the eigenpairs (eigenvalues and eigenvectors)
    where the eigenvectors are
        B-normalized and mutually B-orthogonal.

        The eigenvalues are returned in descending order along with their
    corresponding eigenvectors.

        :param problem: Tuple (A, B), where A and B are given as list-of-lists.
        :return: tuple (eigenvalues_list, eigenvectors_list) where:
                 - eigenvalues_list is a list of eigenvalues (sorted in
    descending order),
                 - eigenvectors_list is a list of eigenvectors (each a list of
    floats) corresponding to those eigenvalues.
        """
        A, B = problem
        A = np.array(A, dtype=float)
        B = np.array(B, dtype=float)

        # Use Cholesky decomposition for numerical stability when dealing with
    B
        L = np.linalg.cholesky(B)
        Linv = np.linalg.inv(L)
        Atilde = Linv @ A @ Linv.T

        # Compute eigenvalues and eigenvectors
        eigenvalues, eigenvectors = np.linalg.eigh(Atilde)

        # Transform eigenvectors back to the original space
        eigenvectors = Linv.T @ eigenvectors

        # Normalize eigenvectors with respect to the B-inner product
        for i in range(eigenvectors.shape[1]):
            v = eigenvectors[:, i]
````
````

```
            norm = np.sqrt(np.dot(v, B @ v))
            if norm > 0:
                eigenvectors[:, i] = v / norm

        # Reverse to get descending order
        eigenvalues = eigenvalues[::-1]
        eigenvectors = eigenvectors[:, ::-1]

        # Convert to lists
        eigenvalues_list = eigenvalues.tolist()
        eigenvectors_list = [eigenvectors[:, i].tolist() for i in range(
    eigenvectors.shape[1])]

        return eigenvalues_list, eigenvectors_list
---
```
```

**System Message**

```
You have sent 2 messages and have used up $0.0130. You have $0.4870 remaining.

Edit successful for solver.py.

Contents of solver.py (lines 1-50 out of 53)
(| = existing code, > = modified code)

> 01: import numpy as np
> 02: from typing import List, Tuple
> 03:
> 04: class Solver:
> 05:   @staticmethod
> 06:   def solve(problem: Tuple[List[List[float]], List[List[float]]]) ->
    Tuple[List[float], List[List[float]]]:
> 07:   """
> 08:   Solve the generalized eigenvalue problem A x = lambda B x.
> 09:
> 10:   Given:
> 11:   - A: a symmetric matrix,
> 12:   - B: a symmetric positive definite matrix,
> 13:   this function computes the eigenpairs (eigenvalues and
    eigenvectors) where the eigenvectors are
> 14:   B-normalized and mutually B-orthogonal.
> 15:
> 16:   The eigenvalues are returned in descending order along with their
     corresponding eigenvectors.
> 17:
> 18:   :param problem: Tuple (A, B), where A and B are given as list-of-
    lists.
> 19:   :return: tuple (eigenvalues_list, eigenvectors_list) where:
> 20:   - eigenvalues_list is a list of eigenvalues (sorted in
    descending order),
> 21:   - eigenvectors_list is a list of eigenvectors (each a
    list of floats) corresponding to those eigenvalues.
> 22:   """
> 23:   A, B = problem
> 24:   A = np.array(A, dtype=float)
> 25:   B = np.array(B, dtype=float)
> 26:
> 27:   # Use Cholesky decomposition for numerical stability when dealing
     with B
```
```

```
> 28:         L = np.linalg.cholesky(B)
> 29:         Linv = np.linalg.inv(L)
> 30:         Atilde = Linv @ A @ Linv.T
> 31:
> 32:         # Compute eigenvalues and eigenvectors
> 33:         eigenvalues, eigenvectors = np.linalg.eigh(Atilde)
> 34:
> 35:         # Transform eigenvectors back to the original space
> 36:         eigenvectors = Linv.T @ eigenvectors
> 37:
> 38:         # Normalize eigenvectors with respect to the B-inner product
> 39:         for i in range(eigenvectors.shape[1]):
> 40:             v = eigenvectors[:, i]
> 41:             norm = np.sqrt(np.dot(v, B @ v))
> 42:             if norm > 0:
> 43:                 eigenvectors[:, i] = v / norm
> 44:
> 45:         # Reverse to get descending order
> 46:         eigenvalues = eigenvalues[::-1]
> 47:         eigenvectors = eigenvectors[:, ::-1]
> 48:
> 49:         # Convert to lists
> 50:         eigenvalues_list = eigenvalues.tolist()
...

Speedup: 1.00x
  (Speedup = Baseline Time / Your Time; Higher is better)

  Valid Solutions: 100%
  Invalid Solutions: 0%
  Timeouts: 0%


Snapshot saved (Best speedup reached, state saved!)
```

C.2.2 Linter Errors

LM Message

```
I'll create the solver.py file with our numba-accelerated implementation using
    the proper prepend command.

edit
file: solver.py
lines: 0-0
---
import numpy as np
from numba import njit

# Global up and down factors. These might be overridden externally.
up = 1
down = 1

@njit(cache=True)
def _upfirdn1d_numba(h, x, up, down):
    Lx = x.shape[0]
    Lh = h.shape[0]
    # The length of the convolved signal after upsampling.
    Lfull = (Lx - 1) * up + Lh
    # Compute the length of the output after downsampling.
```

```
    Lout = (Lfull - 1) // down + 1
    y = np.zeros(Lout, dtype=x.dtype)
    for n in range(Lout):
        pos = n * down
        s = 0.0
        kmin = 0
        if pos - (Lx - 1) * up > 0:
            kmin = pos - (Lx - 1) * up
        kmax = pos
        if kmax > Lh - 1:
            kmax = Lh - 1
        for k in range(kmin, kmax + 1):
            r = pos - k
            if r % up == 0:
                m = r // up
                if m < Lx:
                    s += x[m] * h[k]
        y[n] = s
    return y

def solve(problem: list) -> list:
    """
    Compute the upfirdn operation for each filter-signal pair in the problem
     list.

    Each tuple (h, x) is processed by upsampling x by factor 'up', filtering
     with h, and then downsampling by factor 'down'.

    :param problem: A list of tuples (h, x) where h and x are 1D arrays.
    :return: A list of 1D arrays representing the upfirdn results.
    """
    results = []
    for h, x in problem:
        h_arr = np.ascontiguousarray(np.array(h, dtype=np.float64))
        x_arr = np.ascontiguousarray(np.array(x, dtype=np.float64))
        res = _upfirdn1d_numba(h_arr, x_arr, up, down)
        results.append(res)
    return results
---
```

System Message

```
You have sent 2 messages and have used up $0.0304. You have $0.0696 remaining.

Edit failed (and thus not applied) for solver.py: Line 9: Redefining name 'up'
    from outer scope (line 5) (redefined-outer-name)
Line 9: Redefining name 'down' from outer scope (line 6) (redefined-outer-name)

PROPOSED CHANGES - This is what you tried to apply (lines 1-50 out of 50):
(| = existing code, > = modified code)

> 01: import numpy as np
> 02: from numba import njit
> 03:
> 04: # Global up and down factors. These might be overridden externally.
> 05: up = 1
> 06: down = 1
> 07:
> 08: @njit(cache=True)
> 09: def _upfirdn1d_numba(h, x, up, down):
```

```
> 10:     Lx = x.shape[0]
> 11:     Lh = h.shape[0]
> 12:     # The length of the convolved signal after upsampling.
> 13:     Lfull = (Lx - 1) * up + Lh
> 14:     # Compute the length of the output after downsampling.
> 15:     Lout = (Lfull - 1) // down + 1
> 16:     y = np.zeros(Lout, dtype=x.dtype)
> 17:     for n in range(Lout):
> 18:         pos = n * down
> 19:         s = 0.0
> 20:         kmin = 0
> 21:         if pos - (Lx - 1) * up > 0:
> 22:             kmin = pos - (Lx - 1) * up
> 23:         kmax = pos
> 24:         if kmax > Lh - 1:
> 25:             kmax = Lh - 1
> 26:         for k in range(kmin, kmax + 1):
> 27:             r = pos - k
> 28:             if r % up == 0:
> 29:                 m = r // up
> 30:                 if m < Lx:
> 31:                     s += x[m] * h[k]
> 32:         y[n] = s
> 33:     return y
> 34:
> 35: def solve(problem: list) -> list:
> 36:     """
> 37:     Compute the upfirdn operation for each filter-signal pair in the
      problem list.
> 38:
> 39:     Each tuple (h, x) is processed by upsampling x by factor 'up',
      filtering with h, and then downsampling by factor 'down'.
> 40:
> 41:     :param problem: A list of tuples (h, x) where h and x are 1D arrays.
> 42:     :return: A list of 1D arrays representing the upfirdn results.
> 43:     """
> 44:     results = []
> 45:     for h, x in problem:
> 46:         h_arr = np.ascontiguousarray(np.array(h, dtype=np.float64))
> 47:         x_arr = np.ascontiguousarray(np.array(x, dtype=np.float64))
> 48:         res = _upfirdn1d_numba(h_arr, x_arr, up, down)
> 49:         results.append(res)
> 50:     return results


Contents of current file:
File solver.py is empty.
```

C.2.3 Runtime Errors

LM Message

````
```
eval_input [[1.2, -0.5], [0.3, 2.1]
```
````

System Message

```
You have sent 0 messages and have used up $0.2630. You have $0.2370 remaining.

Starting evaluation...

numpy.linalg.LinAlgError: 0-dimensional array given. Array must be at least two-
    dimensional in function 'solve' at line 22 in solver.py

Code Context:
   12:         """
   13:         Solve the eigenvector problem for the given non-symmetric matrix.

   14:         Compute eigenvalues and eigenvectors using np.linalg.eig.
   15:         Sort the eigenpairs in descending order by the real part (and
    then imaginary part) of the eigenvalues.
   16:         Return the eigenvectors (each normalized to unit norm) as a list
     of lists of complex numbers.
   17:
   18:         :param problem: A non-symmetric square matrix.
   19:         :return: A list of normalized eigenvectors sorted in descending
    order.
   20:         """
   21:         A = problem
 ! 22:         eigenvalues, eigenvectors = np.linalg.eig(A)
   23:         # Zip eigenvalues with corresponding eigenvectors (columns of
    eigenvectors matrix)
   24:         pairs = list(zip(eigenvalues, eigenvectors.T))
   25:         # Sort by descending order of eigenvalue real part, then
    imaginary part
   26:         pairs.sort(key=lambda pair: (-pair[0].real, -pair[0].imag))
   27:         sorted_evecs = []
   28:         for _, vec in pairs:
   29:             vec_arr = np.array(vec, dtype=complex)
   30:             norm = np.linalg.norm(vec_arr)
   31:             if norm > 1e-12:
   32:                 vec_arr = vec_arr / norm
```

E Python Packages

In Table 7 we show the Python packages used AlgoTune, as well as packages installed on the AlgoTuner agent interface.

Table 7: Python packages used in the AlgoTune benchmark, installed on the AlgoTune Agent interface, and their open-source licenses.

Package	AlgoTune (Benchmark)	AlgoTuner (Agent)	License
NumPy (Harris et al., 2020)	✓	✓	BSD 3-Clause
SciPy (Virtanen et al., 2020)	✓	✓	BSD 3-Clause
Pandas (McKinney, 2010)	✗	✓	BSD 3-Clause
Cython (Behnel et al., 2011)	✗	✓	Apache 2.0
Numba (Lam et al., 2015)	✗	✓	BSD 2-Clause
Dask (Rocklin, 2015)	✗	✓	BSD 3-Clause
PuLP (Mitchell et al., 2009)	✓	✓	MIT
OR-Tools (Google, 2020)	✓	✓	Apache 2.0
Pyomo (Hart et al., 2011)	✗	✓	BSD 3-Clause
HiGHS / HighSpy (Huangfu and Hall, 2018)	✗	✓	MIT
NetworkX (Hagberg et al., 2008)	✓	✓	BSD 3-Clause
python-sat (Biere et al., 2012)	✓	✓	MIT
JAX (Bradbury et al., 2018)	✗	✓	Apache 2.0
Diffrax (Kidger, 2021)	✓	✓	Apache 2.0
CVXPY (Agrawal et al., 2018; Diamond and Boyd, 2016)	✓	✓	Apache 2.0

F Performance Improvements in Python Repositories

In this section, we show a sample of performance improving pull requests to three Python repositories: NumPy, SciPy, and NetworkX. All of the below pull requests were submitted in the past two years, and greatly increase performance (as reported in the PR itself). For each PR, we highlight the most significant reported performance greatest improvement.

NumPy:

- **ENH: Add a fast-path for `ufunc.at` on aligned 1D arrays** (PR #22889): Up to 6.3x faster when no casting is needed on 1D aligned inputs (e.g. `bench_ufunc.At.time_sum_at` dropped from 54.0 ± 0.2 ms to 8.42 ± 0.02 ms). https://github.com/numpy/numpy/pull/22889
- **ENH: Vectorize quicksort for 16-bit and 64-bit dtype using AVX512** (PR #22315): Up to 15x speedup for 16-bit sorts and 9x speedup for 64-bit sorts on AVX-512-capable CPUs. https://github.com/numpy/numpy/pull/22315
- **ENH: Accelerate `unique` for integer dtypes via hash tables** (PR #26018): Roughly 2.7x speedup on 1 billion random integers (unique count in 7.815 s vs. 21.436 s for the previous implementation). https://github.com/numpy/numpy/pull/26018

SciPy:

- **ENH: Vectorize `stats.mannwhitneyu`** (PR #19749): Vectorizes the statistic calculation, achieving up to ∼21x speedup (1.38 s → 64.4 ms in certain cases). https://github.com/scipy/scipy/pull/19749
- **ENH: Vectorize `stats.rankdata`** (PR #19776): Vectorizes `rankdata` along an axis, yielding up to $\sim$ 296x faster runtimes (2.58 ms → 8.7 μs for a $(100, 100)$ array). https://github.com/scipy/scipy/pull/19776
- **ENH: Fast-path for sparse Frobenius norm** (PR #14317): Directly accesses the data array to compute the norm, resulting in up to 5x speedup in some cases. https://github.com/scipy/scipy/pull/14317

NetworkX:

- **BUG: Fix `weakly_connected_components()` performance on graph views** (PR #7586): Moves the repeated `len(G)` call outside the loop, cutting runtime from ~ 15.4 s to 0.064 s per iteration, over $240\times$ faster. https://github.com/networkx/networkx/pull/7586
- **ENH: Speed up `harmonic_centrality`** (PR #7595): Implements graph reversal for node-subset queries, reducing computation on large wheel graphs from 95.9ms to 717μs, 134x faster. https://github.com/networkx/networkx/pull/7595
- **ENH: Speed up `common_neighbors` / `non_neighbors`** (PR #7244): Replaces generator-based neighbor lookups with direct `_adj` dict operations, achieving up to ∼600x speedup on star-center queries and around 11x on complete-graph common neighbors. https://github.com/networkx/networkx/pull/7244

G AlgoTune vs KernelBench

Concurrent work, KernelBench (Ouyang et al., 2025) is similar to AlgoTune: both are code optimization benchmarks. In this section, we summarize the main differences between them.

KernelBench is made up of 250 GPU kernels, where the goal is to write highly optimized low level code that speeds up their runtime, while still producing correct outputs. KernelBench is split into three levels based on kernel complexity. The first level contains simple funcitons like `softmax` or `tanh`, while the third level contains more complex kernels like `ShuffleNet` or `LSTM`.

This approach has two downsides: first, the runtimes of kernels in the benchmark is highly varied; level 1 kernels run in microseconds, while level 3 kernels run in milliseconds (see Table 8). 40.8% of the kernels in KernelBench run in under 0.1 milliseconds, while the rest take between 0.1 and 100 milliseconds to run. Kernels with low runtimes are harder to optimize, as the process overhead takes a significant part of the runtime. This makes the comparison between the improvement of different kernel runtimes somewhat complicated.

In contrast, AlgoTune's tasks have controllable runtimes, which results in the benchmark having more uniform runtimes (see I). Importantly, AlgoTune covers a broage range of functions in math, science, computer science, machine learning, and more (see §1 for a discussion).

Table 8: Number of kernels per time interval as reported by Ouyang (2025), for KernelBench (Ouyang et al., 2025), by level.

Time Interval	Level 1 (100 ops)	Level 2 (100 ops)	Level 3 (50 ops)	Pct of Total [%]
10–20 μs	21	0	0	8.4
20–50 μs	24	12	0	14.4
50–100 μs	4	37	4	18.0
0.1–1 ms	22	21	8	20.4
1–10 ms	23	20	20	25.2
10–100 ms	6	10	18	13.6

H Task Size Determination

Algorithm 1 shows the two phase search algorithm used to find the problem size parameter n for each Task in the benchmark.

Algorithm 1 FINDKFORTIME — choose n whose mean solve time best matches target time τ

Require: task T; target time τ; bounds $[k_{\min}, k_{\max}]$; examples m; seed s; initial samples n_{init}; refinement steps n_{ref}
Ensure: estimate $\hat{k}$ (or $\varnothing$ if none succeeds)

function PROBE(k)
    if $k \in$ cache **then**
        return cache[k]
    end if
    $(t_k, \text{stats}) \leftarrow \text{measure\_solve\_time}(T, k, \tau, m, s, \dots)$
    cache[k] $\leftarrow (t_k, \text{stats})$
    return (t_k, stats)
end function
cache $\leftarrow$ empty map
if $k_{\min} = k_{\max}$ **then**
    return PROBE($k_{\min}$)
end if
$\mathcal{K}_{\text{sample}} \leftarrow \left\{ \lfloor 10^x \rceil : x \in \text{linspace}(\log_{10} k_{\min}, \log_{10} k_{\max}, n_{\text{init}}) \right\} \cup \{k_{\min}, k_{\max}\}$
sort $\mathcal{K}_{\text{sample}}$ and deduplicate
$k_{\text{lastOK}} \leftarrow \varnothing,\ k_{\text{upper}} \leftarrow \varnothing$
for all $k \in \mathcal{K}_{\text{sample}}$ **do**
    $(t_k, \_) \leftarrow$ PROBE(k)
    if $t_k = \varnothing \ \vee \ t_k > \tau$ **then**
        if $k_{\text{lastOK}} \neq \varnothing$ **then**
            $k_{\text{upper}} \leftarrow k$; **break**
        end if
    else
        $k_{\text{lastOK}} \leftarrow k$
    end if
end for
$\hat{k} \leftarrow \arg \min_{(k, t_k) \in \text{cache},\, t_k \neq \varnothing} \left(|t_k - \tau|,\ k \right)$
if $k_{\text{lastOK}} \neq \varnothing$ **and** $k_{\text{upper}} \neq \varnothing$ **then**
    $lo \leftarrow k_{\text{lastOK}},\ hi \leftarrow k_{\text{upper}}$
    for $i = 1$ **to** n_{ref} **do**
        if $hi - lo \leq 1$ **then break**
        end if
        $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
        $(t_{mid}, \_) \leftarrow$ PROBE(mid)
        if $t_{mid} \neq \varnothing$ **and** $(|t_{mid} - \tau|, mid) < (|t_{\hat{k}} - \tau|, \hat{k})$ **then**
            $\hat{k} \leftarrow mid$
        end if
        if $t_{mid} = \varnothing \ \vee \ t_{mid} > \tau$ **then**
            $hi \leftarrow mid - 1$
        else
            $lo \leftarrow mid$
        end if
    end for
end if
return $\hat{k}$

I Task Timings

We report the size parameter n and per task timings in Table 9. The average time per task is calculated by running the reference solver on 100 development instances, and repeating this three times.

Table 9: Per task size parameter n values and average time for the reference solve function, across three timing runs.

Task	n	Average Time (ms)
`affine_transform_2d`	520	17.46 ± 0.14
`articulation_points`	841	98.41 ± 0.48
`base64_encoding`	2387	4.41 ± 0.01
`btsp`	8	0.82 ± 0.02
`capacitated_facility_location`	6	263.55 ± 1.69
`chacha_encryption`	3196	0.99 ± 0.00
`channel_capacity`	138	96.18 ± 0.25
`chebyshev_center`	242	101.06 ± 0.58
`cholesky_factorization`	668	8.77 ± 0.18
`clustering_outliers`	2359	92.37 ± 0.39
`communicability`	61	98.25 ± 1.79
`convex_hull`	191235	20.19 ± 0.33
`convex_quadratic_check`	605	100.75 ± 1.84
`convolve2d_full_fill`	6	147.48 ± 0.23
`convolve_1d`	16325	15.32 ± 0.09
`correlate_1d`	88	10.12 ± 0.04
`count_connected_components`	794	16.09 ± 0.10
`count_riemann_zeta_zeros`	15849	68.00 ± 0.18
`crew_pairing`	343	131.36 ± 1.00
`cumulative_simpson_multid`	67	10.01 ± 0.50
`cyclic_independent_set`	4	87.97 ± 0.17
`determinant_matrix_exponential`	662	10.04 ± 0.02
`dijkstra_from_indices`	1607	9.88 ± 0.02
`discrete_log`	25	19.72 ± 0.16
`dynamic_assortment_planning`	36	99.46 ± 0.94
`earth_movers_distance`	409	9.26 ± 0.05
`edge_expansion`	8507	67.99 ± 0.38
`efficiency`	500	101.01 ± 0.40
`eigenvalues_complex`	476	99.02 ± 0.21
`eigenvalues_real`	358	10.10 ± 0.02
`eigenvectors_complex`	467	99.26 ± 0.08
`eigenvectors_real`	445	19.36 ± 0.13
`elementwise_integration`	378	101.44 ± 0.04
`feedback_controller_design`	12	79.83 ± 0.33
`fft_real_scipy_fftpack`	631	8.14 ± 0.03
`generalized_eigenvalues_complex`	275	103.61 ± 0.05
`generalized_eigenvalues_real`	363	20.12 ± 0.08
`generalized_eigenvectors_complex`	272	102.09 ± 0.21
`generalized_eigenvectors_real`	236	9.92 ± 0.05
`graph_coloring_assign`	42	101.39 ± 0.20
`graph_isomorphism`	127	92.04 ± 0.12
`graph_laplacian`	4839	10.03 ± 0.12
`gzip_compression`	653	99.40 ± 0.01
`job_shop_scheduling`	17	99.40 ± 6.36
`kalman_filter`	23	97.41 ± 0.33
`kcenters`	33	9.63 ± 0.02
`kd_tree`	77	10.13 ± 0.05
`kernel_density_estimation`	631	199.79 ± 0.25
`kmeans`	278	96.28 ± 0.38
`ks_test_2samp`	50610	9.97 ± 0.06
`l0_pruning`	83829	9.95 ± 0.13
`l1_pruning`	96069	19.84 ± 0.22

Continued on next page

Table 9 – continued from previous page

Task	n	Average Time (ms)
lasso	87	10.14 ± 0.09
linear_system_solver	532	10.01 ± 0.04
lp_box	297	100.38 ± 0.59
lp_centering	304	105.24 ± 0.32
lqr	111	99.62 ± 1.41
lu_factorization	497	17.62 ± 0.19
markowitz	396	103.83 ± 0.25
matrix_exponential	552	98.32 ± 0.68
matrix_multiplication	298	11.27 ± 0.09
matrix_sqrt	281	100.40 ± 0.21
max_clique	17	88.86 ± 2.94
max_common_subgraph	4	24.21 ± 0.49
max_flow_min_cost	63	99.08 ± 1.20
max_independent_set	15	55.09 ± 1.68
max_weighted_independent_set	27	10.83 ± 0.82
min_weight_assignment	233	9.58 ± 0.01
minimum_spanning_tree	574	101.43 ± 0.62
minimum_volume_ellipsoid	27	115.07 ± 0.78
multi_dim_knapsack	215	192.66 ± 18.89
nmf	6	83.00 ± 0.07
ode_fitzhughnagumo	17	81.52 ± 0.16
ode_hires	706	120.77 ± 0.38
ode_lorenz96_nonchaotic	3	25.78 ± 0.23
ode_nbodyproblem	9	113.80 ± 0.73
ode_stiff_robertson	9999999	82.71 ± 0.15
ode_stiff_vanderpol	2	111.07 ± 0.55
odr	17637	45.08 ± 0.02
pagerank	7978	76.79 ± 0.54
pca	36	105.81 ± 0.59
pde_burgers1d	12	103.34 ± 0.52
pde_heat1d	9	98.58 ± 0.66
polynomial_mixed	415	103.33 ± 2.37
polynomial_real	396	99.12 ± 0.09
portfolio_optimization_cvar	15	97.71 ± 0.24
procrustes	307	19.76 ± 0.09
psd_cone_projection	349	100.60 ± 0.11
qp	278	97.59 ± 0.15
qr_factorization	500	18.42 ± 0.56
quantile_regression	187	98.70 ± 0.82
queens_with_obstacles	14	142.63 ± 3.12
queuing	420	4.83 ± 0.01
qz_factorization	271	99.88 ± 0.22
randomized_svd	310	10.23 ± 0.02
rbf_interpolation	85	68.17 ± 0.12
rectanglepacking	10	341.44 ± 150.17
robust_linear_program	12	99.10 ± 0.41
rotate_2d	506	18.41 ± 0.08
set_cover_conflicts	9	9.90 ± 1.05
set_cover	74	154.40 ± 11.25
sha256_hashing	1818	1.00 ± 0.00
shift_2d	555	18.72 ± 0.02
shortest_path_dijkstra	352	100.62 ± 0.27
sparse_eigenvectors_complex	1561	183.99 ± 0.35
spectral_clustering	27	133.05 ± 1.21
stable_matching	471	9.60 ± 0.03
toeplitz_solver	8588	101.29 ± 2.20
tsp	39	106.21 ± 26.47
two_eigenvalues_around_0	568	19.76 ± 0.12
unit_simplex_projection	127543	9.86 ± 0.10
vector_quantization	30	9.97 ± 0.00
vectorized_newton	40684	4.99 ± 0.02

Continued on next page

Table 9 – continued from previous page

Task	n	Average Time (ms)
`vehicle_routing`	14	91.82 $\pm$ 1.85
`vehicle_routing_circuit`	8	140.19 $\pm$ 2.36
`vertex_cover`	1	9.83 $\pm$ 1.12
`vertex_cover_cpsat`	17	79.95 $\pm$ 1.28
`wasserstein_dist`	15439	19.17 $\pm$ 0.11
`water_filling`	4075	98.59 $\pm$ 0.65
`zoom_2d`	480	26.06 $\pm$ 0.15