A APPROX DIALECT SPECIFICATION

This appendix provides a specification for each operation within the `approx` dialect.

A.1 ApproxMLIR Management Operations

These operations define logic for controlling approximations.

A.1.1 approx.knob

Syntax

```
approx.knob <{
    id = ...,
    params = "...",
    transform_types = "...",
}> (%arg0, ...) -> (type, ...) {
  // Region containing the exact code
}
```

Summary The `approx.knob` operation is the central interface with autotuner.

Data-Flow Analysis

Operands: The values that are live-in to the knob's region.

Results: The values that are live-out of the knob's region.

Attributes

id: A unique identifier for the knob.

params: (Array) This attribute is read and written by the autotuner, and is parsed by `approx-runtime`.

transform_types: (Array of Strings) Lists the types of transformations (e.g., "loop_perforate") that can be applied within this knob's region.

Region The region where an approximation can be applied.

A.1.2 approx.decision_tree

Syntax

```
approx.decision_tree (%runtime_input) <{
    runtime = "...",
    thresholds = [...],
    decisions = [...],
    transform_type = "..."
}> {
    // code in each branch
}
```

Summary Enables fine-grained, dynamic approximation strategies by emitting a decision tree implemented by control flow and `approx-runtime`.

Operands

runtime_input: The value, determined at runtime by the `runtime` function and the emitted branch counting logic, used to select a branch (i.e. which approximation to apply).

Attributes

runtime: (String) The name of the user-provided runtime function to be called.

thresholds: (Array of integers, derived from knob params) An array of values that define the boundaries for branch selection. This attribute is set by the compiler runtime, which parses the `params` attribute of the parent `approx.knob` operation.

decisions: (Array of integers, derived from knob params) An array of values, one for each branch, dictating the approximation to apply. This attribute is also set by the compiler by parsing the parent `approx.knob`'s `params`.

transform_type: (String, derived from knob params) Pass to `transform` operations within its branches.

Region Inherited from the parent knob operation, describing the region to apply approximation for each branch. When decision tree operation is lowered, the region will be cloned to all branches of the control-flow branches, along with emitted transform operation.

A.2 ApproxMLIR transform operation

A.2.1 approx.transform

Syntax

```
approx.transform <{
    transform_type = "...",
    transform_value = ...
}>
```

Summary Specify a concrete, low-level program transformation. Can be inserted within knob op, specifying static approximation on the knob, or inserted within decision tree's branches, specifying fine-grained dynamic approximation on the knob.

Attributes

`transform_type`: (String) Defines the type of approximation. Implemented examples include `"loop-perforate"`, `"function-substitute"`, and `"task-skipping"`.

`transform_value`: An integer value associated with the transformation. 0 means exact. The larger the value is, the more aggressively we approximate the parent region.

B WORKLOADS

B.1 lavaMD

LavaMD (Molecular Dynamics) is a computational simulation benchmark designed to model the physical movements of atoms and molecules. It calculate the potential energy and forces between all particles within a given system, which is typically divided into a 3D grid of boxes. Particles interact with other particles within their own box ("self-box") and with particles in adjacent boxes ("neighbor-box").

The workload is tuned with a grid size parameter of 8.

B.1.1 Input Dataset Generation

The input dataset consists of randomly generated particles. The number of particles is determined by the number of boxes in the 3D grid and a fixed number of particles per box.

The particle data are initialized randomly at the start. The position vector of each particle is initialized with random floating-point values in a specific range ($[0.1, 1.0]$). Each particle's scalar charge is also a random float in the same range.

B.1.2 Approximation Knobs

Self-Box Calculation Knob This approximation applies to the calculation of forces between particles within the same box. It uses loop perforation to dynamically skip calculations. The decision to approximate is based on the specific particle's charge. If the charge exceeds a pre-defined threshold, the computation is accelerated by skipping half of the particle-pair interactions. Otherwise, the full, exact calculation is performed for that particle.

Neighbor-Box Calculation Knob This approximation applies to the calculation of forces from particles in all adjacent boxes. It uses function substitution, dynamically replacing the exact function with one of approximate versions. The choice of which function to use is purely probabilistic. There is a 70% probability of substituting the exact function with a version that skips 1/8 of iterations, and a 30% probability of substituting it with a more aggressive version that skips 1/4 of iterations.

B.2 Knowledge Base (KB)

This workload simulates a vector search task. It compare a single query against a large corpus of documents. Both the query and the documents are represented as embeddings. The core of the workload involves parsing these embeddings from text and computing the cosine similarity between the query vector and every document vector to find the most relevant documents.

B.2.1 Input Dataset Generation

The input query is a question randomly selected from the Natural Questions (NQ) dataset, and the document corpus is the NQ dataset's documents. In this workload, each document's high-dimensional embedding is stored as a long string of text.

B.2.2 Approximation Knobs

Embedding Parsing Knob The approximation applies to the initial step of converting from documents to embeddings. It dynamically selects the parsing method: when the runtime state exceeds a threshold, the system switches from exact parsing to an approximate parser that truncates each value to lower precision

Similarity Computation Knob This approximation applies to the main computation loop that iterates over all documents in the corpus to find the best matches. It also uses function substitution. Based on a runtime state. In the exact case (if the state is below a threshold), it computes the similarity for 100% of the documents. In the approximate case (if the state is above that threshold), it substitutes an approximate version that probabilistically skipping 5% of the documents to accelerate the overall search process.

B.3 BM25

This workload implements the Okapi BM25 (Best Matching 25) algorithm, a ranking function used in information retrieval and search engines. It scores the relevance of documents based on the query terms they contain. The score is calculated using term frequency (TF), inverse document frequency (IDF), and document length. The process involves text preprocessing (lowercasing, tokenizing), calculating statistics for each query term, and scoring all documents.

B.3.1 Input Dataset Generation

The input query is a question randomly selected from the Natural Questions (NQ) dataset, and the document corpus

consists of the NQ dataset's documents, loaded from a text file where each line is a separate document.

B.3.2 Approximation Knobs

Text Preprocessing Knob This approximation applies during the document and query preprocessing stage. It uses function substitution. Based on a runtime state, the system may use a version that only lowercases the first character of each word and uses a simpler space-based word boundary check instead of the full alphanumeric test.

Corpus Subsetting Knob This approximation applies during the initial preprocessing stage. It uses function substitution. Based on a runtime state, the system may use a version that probabilistically skips documents (e.g., with 15%, 30%, or 50% probability), reducing the total number of documents that are considered for the search.

Term Scoring Knob This approximation applies during the main scoring loop, which is executed for each unique term in the query. It uses function substitution to replace the exact scoring function. Based on a runtime state (derived from the term's IDF), the system may skip the scoring calculation for a document with a 10% or 20% probability.

B.4 K-Means

This workload implements the K-Means algorithm. The algorithm iteratively refines a set of 'k' cluster centers (centroids). In each iteration, it performs two steps: first, it assigns every data point to its nearest centroid; second, it recalculates each centroid as the mean of all points assigned to it. This process repeats for a fixed number of iterations.

B.4.1 Input Dataset Generation

The input data set is generated synthetically. It consists of a specified number of high-dimensional data points. Each coordinate of each point is a random floating-point value within a fixed positive range ($[0.0, 100.0]$). The initial 'k' centroids are selected by choosing 'k' of these generated data points at random.

B.4.2 Approximation Knobs

Centroid Selection Knob This approximation applies to the "assignment" step, specifically the inner loop where a single point calculates its distance to all 'k' centroids to find the nearest one. This loop is a knob that can be perforated. Based on the current k-means iteration number, the loop may be accelerated by skipping distance calculations to some of the centroids, potentially assigning the point to a "near" but not "nearest" centroid.

Point Assignment Knob This approximation applies to the main "assignment" loop, which iterates over all data points in the dataset. This loop is also a knob that can be perforated. Based on the current k-means iteration number, the loop may skip processing some data points entirely. Those skipped points are not assigned to a new cluster and do not contribute to the recalculation of the centroids in that iteration.

B.5 PageRank

This workload implements the PageRank algorithm, a method for ranking the importance of nodes in a graph. The algorithm is iterative. In each iteration, it updates every node's rank by summing the rank contributions from all nodes that link to it (its in-neighbors). A node's contribution is its own rank divided by its total number of outgoing links (out-degree). A "damping factor" is also applied to simulate a random web surfer.

B.5.1 Input Dataset Generation

The input is a directed graph. The workload supports two generation modes. The primary mode generates a random graph with a specified number of nodes and an average in-degree. Edges are created by randomly selecting source nodes for each destination node. Alternatively, the "file" mode can load a graph from a text file containing an edge list.

B.5.2 Approximation Knobs

Node Rank Update Knob This approximation applies to the function that calculates a single node's new rank. It uses function substitution, and the decision is based on the current iteration number. The available approximate versions implement the summation logic using a hard-coded stride, skipping either half or two-thirds of the in-neighbors.

Neighbor Summation Knob This approximation applies during the iterative PageRank computation. It uses function substitution. Based on a runtime state, the system may use a version that probabilistically skips iterations (e.g., with 25% probability) or uses a strided iteration pattern reducing the total number of iterations performed and removing expensive thread synchronization barriers to allow asynchronous execution across worker threads.

B.6 LLM + RAG (bm25)

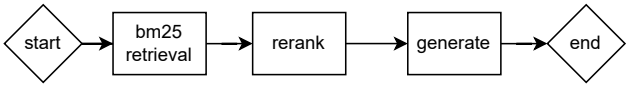

Figure 9. LLM + RAG (bm25) AI workflow

This workload is a Retrieval-Augmented Generation (RAG) system that answers user queries by first searching a document corpus. The system uses the BM25 kernel to retrieve an initial set of documents, which are then re-ranked. Based on a confidence score from this retrieval step, the system generates the final answer.

B.6.1 Approximation Knobs

Retrieval Knob The entire BM25 retrieval kernel are approximated the same as the bm25 kernel.

QA Model Selection Knob This is a dynamic approximation based on the retrieval score, to choose between fast and pro ML kernel.

B.7 LLM + RAG(kb)

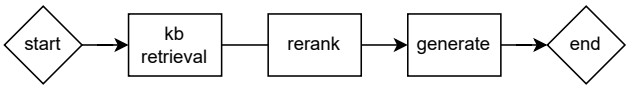

Figure 10. LLM + RAG (kb) AI workflow

This workload is a RAG system that uses vector embeddings for retrieval. A user's query is first embedded and then compared against a vector database using the "knowledge base" (KB) kernel to find semantically similar documents. After a re-ranking step, the system generates the final answer based on the retrieved context.

B.7.1 Approximation Knobs

Retrieval Knob The retrieval part is also the same as the kb kernel.

QA Model Selection Knob Based on the retrieval score, the ML kernel is approximated to use fast or pro ML kernel.

B.8 LLM + tool invocation

LLM + tool invocation answers queries by using external tools. A user's query is first analyzed by a "triage agent" which selects an appropriate tool (e.g., 'kb', 'bm25', 'lavaMD', 'kmeans') to execute. The raw output from the chosen tool is then post-processed by a "summary agent." Finally, this summarized result is passed to generate the final natural-language answer, as shown in Figure 11.

B.8.1 Approximation Knobs

This system has a deep hierarchy of approximation knobs at every stage of the pipeline.

Tool Kernel Knobs Each individual tool that can be called ('kb', 'bm25', 'lavaMD', etc.) has knobs within.

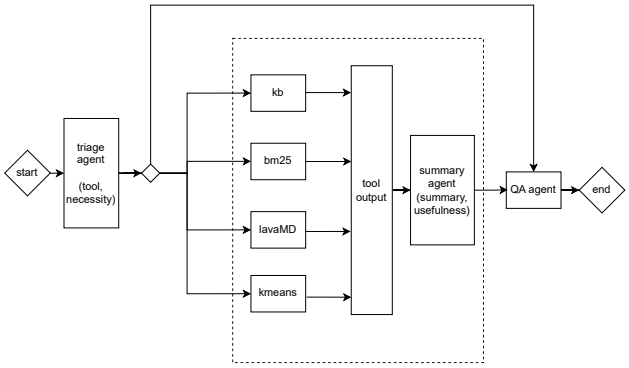

Figure 11. LLM + tool invocation AI workflow

Summary Agent Knob The summarization can be approximated using a simpler, faster summarization LLM.

QA Model Selection Knob The QA can also be approximated to using a faster LLM.

C REWRITE RULES

C.1 Function substitution

input MLIR

```
func.func @approx_base_1
(%arg0: i32, %arg1: i32) -> i32
attributes {llvm.linkage =
#llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c2 = arith.constant 2 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg2 = %c1 to %0 step %c2
iter_args(%arg3 = %c0_i32) -> (i32) {
  %2 = arith.index_cast %arg2 :
  index to i32
  %3 = arith.muli %2, %arg1 : i32
  %4 = arith.addi %arg3, %3 : i32
  scf.yield %4 : i32
}
return %1 : i32
}
func.func @approx_base_2
(%arg0: i32, %arg1: i32) -> i32
attributes {llvm.linkage =
#llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c2 = arith.constant 2 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg2 = %c1 to %0 step
%c2 iter_args(%arg3 = %c0_i32) -> (i32) {
  %2 = arith.index_cast %arg2 :
  index to i32
  %3 = arith.muli %2, %arg1 : i32
  %4 = arith.addi %arg3, %3 : i32
  scf.yield %4 : i32
```

```
}
return %1 : i32
}
func.func @base(%arg0: i32, %arg1: i32) -> i32
attributes {llvm.linkage =
#llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg2 = %c1 to %0 step %c1
iter_args(%arg3 = %c0_i32) -> (i32) {
  %2 = arith.index_cast %arg2 :
  index to i32
  %3 = arith.muli %2, %arg1 : i32
  %4 = arith.addi %arg3, %3 : i32
  scf.yield %4 : i32
}
return %1 : i32
}
```

output MLIR

```
func.func @__internal_base
(%arg0: i32, %arg1: i32)
-> i32 attributes
{llvm.linkage = #llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg2 = %c1 to
%0 step %c1 iter_args(%arg3 = %c0_i32)
-> (i32) {
  %2 = arith.index_cast %arg2 :
  index to i32
  %3 = arith.muli %2, %arg1 : i32
  %4 = arith.addi %arg3, %3 : i32
  scf.yield %4 : i32
}
return %1 : i32
}
func.func @base
(%arg0: i32, %arg1: i32) -> i32 attributes
{llvm.linkage = #llvm.linkage<external>} {
%7 = func.call @approx_base_2(%arg0, %arg1)
: (i32, i32) -> i32
scf.yield %7 : i32
return %7 : i32
}
```

C.2 Loop perforation

input MLIR

```
func.func @base(%arg0: i32, %arg1: i32) -> i32
attributes {llvm.linkage =
#llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg2 = %c1 to %0 step %c1
iter_args(%arg3 = %c0_i32) -> (i32) {
  %2 = arith.index_cast %arg2 :
  index to i32
  %3 = arith.muli %2, %arg1 : i32
  %4 = arith.addi %arg3, %3 : i32
  scf.yield %4 : i32
}
return %1 : i32
}
```

output MLIR

```
func.func @base
(%arg0: i32, %arg1: i32) -> i32
attributes {llvm.linkage =
#llvm.linkage<external>} {
  %4 = arith.index_cast %arg0 : i32 to index
  %5 = scf.for %arg2 = %c1 to %4 step %c1
  iter_args(%arg3 = %c0_i32) -> (i32) {
    %6 = arith.index_cast %arg2 :
    index to i32
    %7 = arith.muli %6, %arg1 : i32
    %8 = arith.addi %arg3, %7 : i32
    scf.yield %8 : i32
  }

return %5 : i32
}
```

C.3 Task skipping

input MLIR

```
func.func @v1_impl
(%arg0: i32, %arg1: memref<?xi32>,
%arg2: i32) attributes
{llvm.linkage = #llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg3 = %c1 to %0 step %c1
iter_args(%arg4 = %c0_i32) -> (i32) {
  %2 = arith.index_cast %arg3 : index to i32
  %3 = arith.muli %2, %arg2 : i32
  %4 = arith.addi %arg4, %3 : i32
  scf.yield %4 : i32
}
affine.store %1, %arg1[0] : memref<?xi32>
return
}
func.func @v2_impl(%arg0: i32,
%arg1: memref<?xi32>, %arg2: i32)
attributes {llvm.linkage =
#llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c2 = arith.constant 2 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg3 = %c1 to %0 step %c2
iter_args(%arg4 = %c0_i32) -> (i32) {
  %2 = arith.index_cast %arg3 : index to i32
  %3 = arith.muli %2, %arg2 : i32
  %4 = arith.addi %arg4, %3 : i32
  scf.yield %4 : i32
}
affine.store %1, %arg1[0] : memref<?xi32>
return
}
```

```
func.func @v3_impl(%arg0: i32,
%arg1: memref<?xi32>, %arg2: i32)
attributes
{llvm.linkage = #llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c4 = arith.constant 4 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg3 = %c1 to %0 step %c4
iter_args(%arg4 = %c0_i32) -> (i32) {
  %2 = arith.index_cast %arg3 : index to i32
  %3 = arith.muli %2, %arg2 : i32
  %4 = arith.addi %arg4, %3 : i32
  scf.yield %4 : i32
}
affine.store %1, %arg1[0] : memref<?xi32>
return
}
func.func @base(%arg0: i32,
%arg1: memref<?xi32>, %arg2: i32)
attributes
{llvm.linkage = #llvm.linkage<external>} {
%c20_i32 = arith.constant 20 : i32
%c10_i32 = arith.constant 10 : i32
%0 = arith.cmpi sle, %arg0, %c10_i32 : i32
scf.if %0 {
  func.call @v1_impl(%arg0, %arg1, %arg2) :
  (i32, memref<?xi32>, i32) -> ()
} else {
  %1 = arith.cmpi sle, %arg0, %c20_i32 : i32
  scf.if %1 {
    func.call @v2_impl(%arg0, %arg1, %arg2) :
    (i32, memref<?xi32>, i32) -> ()
  } else {
    func.call @v3_impl(%arg0, %arg1, %arg2) :
    (i32, memref<?xi32>, i32) -> ()
  }
}
return
}
```

output MLIR

```
func.func @v2_impl(%arg0: i32,
%arg1: memref<?xi32>, %arg2: i32)
attributes {llvm.linkage =
#llvm.linkage<external>} {
%c1 = arith.constant 1 : index
%c2 = arith.constant 2 : index
%c0_i32 = arith.constant 0 : i32
%0 = arith.index_cast %arg0 : i32 to index
%1 = scf.for %arg3 = %c1 to %0 step %c2
iter_args(%arg4 = %c0_i32) -> (i32) {
  %2 = arith.index_cast %arg3 : index to i32
  %3 = arith.muli %2, %arg2 : i32
  %4 = arith.addi %arg4, %3 : i32
  scf.yield %4 : i32
}
affine.store %1, %arg1[0] : memref<?xi32>
return
}
func.func @base(%arg0: i32, %arg1:
memref<?xi32>, %arg2: i32)
attributes {llvm.linkage =
#llvm.linkage<external>} {
  func.call @v2_impl(%arg0, %arg1, %arg2) :
  (i32, memref<?xi32>, i32) -> ()
    return
}
```

D QoS Metric Code Examples

This appendix shows the QoS metric definitions from our benchmarks, as examples of how users specify accuracy constraints in ApproxMLIR. Each benchmark follows the same pattern: (1) compile and run the *exact* configuration once to generate ground truth, (2) define a QoS metric that returns the accuracy of the result of the approximate program, (3) define an `evaluate_fn` that compiles and runs an approximate configuration and returns `(time_ms, accuracy)`, and (4) call `ar.tune()` with an `accuracy_threshold` serving as the QoS lower bound.

Specifically, `evaluate_fn` is a driver function that orchestrates the calling of compiled artifacts, including (1) how inputs (i.e. datasets) are generated or loaded, (2) how compiled artifacts are called, (3) how accuracy is calculated and (4) the performance is measured.

D.1 Ranking QoS via Rank-Biased Overlap (BM25, KB)

For RAG kernels, accuracy is measured as Rank-Biased Overlap (RBO) between the exact and approximate ranked document lists, with persistence parameter $p = 0.95$. The user needs to implement `compute_similarity` (which computes the QoS metric) and `evaluate_fn` (a driver function, which contains the command line and collects the QoS metric and time), then pass `evaluate_fn` to `ar.tune()`:

QoS metric (RBO):

```
def compute_similarity(gt_list, approx_list,
                       p=0.95):
    max_depth = min(1000,
        max(len(gt_list), len(approx_list)))
    rbo_raw = 0.0
    gt_set, approx_set = set(), set()
    for d in range(1, max_depth + 1):
        if d <= len(gt_list):
            gt_set.add(gt_list[d - 1])
        if d <= len(approx_list):
            approx_set.add(approx_list[d - 1])
        overlap = len(gt_set & approx_set)
        rbo_raw += (p**(d-1)) * (overlap / d)
    norm = (1-p) * sum(
        p**(d-1)
        for d in range(1, max_depth+1))
    if norm == 0.0:
        return 0.0
    return (1-p) * rbo_raw / norm
```

evaluate_fn and tuning entry point

```
def evaluate_fn(config):
    exec_path = compile_mlir_to_native_exec(
        manager.apply_config(
            annotated_mlir, config),
        tag="bm25")
    times, accuracies = [], []
```

```
    for confidence in sample_confidences():
        result = run_exec(
            exec_path,
            [docs_path, query,
             str(confidence)])
        time_ms, approx_list = \
            parse_kernel_out(result.stdout)
        accuracies.append(
            compute_similarity(
                gt_list, approx_list,
                p=0.95))
        times.append(time_ms)
    return mean(times), mean(accuracies)

ar.tune(
    mlir_source=annotated_mlir,
    evaluate_fn=evaluate_fn,
    accuracy_threshold=float(
        os.environ.get(
            "ACCURACY_THRESHOLD", "0.9")),
    time_budget=12000,
    result_callback=result_logger("bm25"),
)
```

The same metric and driver structure is reused for the KB benchmark.

D.2 Centroid QoS via Normalized ℓ_2 Error (K-means)

For the K-means clustering kernel, accuracy is defined as $1 - \|\mathbf{y}_{\text{exact}} - \mathbf{y}_{\text{approx}}\|_2 \;/\; \|\mathbf{y}_{\text{exact}}\|_2$, meaning comparing the optimal assignment of approximate centroids with ground-truth centroids:

QoS metric (normalized ℓ_2):

```
def compute_similarity(gt, approx):
    best_map, min_dist = {}, float("inf")
    for perm in itertools.permutations(
            sorted(approx.keys())):
        mapping = dict(
            zip(sorted(gt.keys()), perm))
        total = sum(
            math.sqrt(sum(
                (a-b)**2
                for a, b in zip(
                    gt[k],
                    approx[mapping[k]])))
            for k in gt)
        if total < min_dist:
            min_dist, best_map = \
                total, mapping
    y_exact = [c for k in sorted(gt)
               for c in gt[k]]
    y_approx = [c for k in sorted(gt)
                for c in approx[best_map[k]]]
    norm_diff = math.sqrt(sum(
        (e-a)**2
        for e, a in zip(y_exact, y_approx)))
    norm_exact = math.sqrt(
        sum(e**2 for e in y_exact))
    if norm_exact == 0.0:
        return 1.0
    return 1.0 - norm_diff / norm_exact
```

D.3 Unified Tuning Interface

Regardless of which metric is used, all benchmarks share the same `ar.tune()` call. Users implement only `evaluate_fn` callback, returning a `(time_ms, accuracy)` pair where `accuracy` is any real-valued score in $[0, 1]$ meaningful to the application:

```
result = ar.tune(
    mlir_source=annotated_mlir,
    evaluate_fn=evaluate_fn,
    accuracy_threshold=0.9,
    time_budget=12000,
    result_callback=result_logger("bm25"),
)
```