Automatic Tracing in Task-Based Runtime Systems

Rohan Yadav
Stanford University
Stanford, California, USA
rohany@cs.stanford.edu

Michael Bauer
NVIDIA
Santa Clara, California, USA
mbauer@nvidia.com

David Broman
KTH Royal Institute of Technology
Stockholm, Sweden
dbro@kth.se

Michael Garland
NVIDIA
Santa Clara, California, USA
mgarland@nvidia.com

Alex Aiken
Stanford University
Stanford, California, USA
aiken@cs.stanford.edu

Fredrik Kjolstad
Stanford University
Stanford, California, USA
kjolstad@cs.stanford.edu

Abstract

Implicitly parallel task-based runtime systems often perform dynamic analysis to discover dependencies in and extract parallelism from sequential programs. Dependence analysis becomes expensive as task granularity drops below a threshold. Tracing techniques have been developed where programmers annotate repeated program fragments (traces) issued by the application, and the runtime system memoizes the dependence analysis for those fragments, greatly reducing overhead when the fragments are executed again. However, manual trace annotation can be brittle and not easily applicable to complex programs built through the composition of independent components. We introduce Apophenia, a system that automatically traces the dependence analysis of task-based runtime systems, removing the burden of manual annotations from programmers and enabling new and complex programs to be traced. Apophenia identifies traces dynamically through a series of dynamic string analyses, which find repeated program fragments in the stream of tasks issued to the runtime system. We show that Apophenia is able to come between 0.92x–1.03x the performance of manually traced programs, and is able to effectively trace previously untraced programs to yield speedups of between 0.91x–2.82x on the Perlmutter and Eos supercomputers.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**.

Keywords: Dynamic Analysis; Runtime Systems; Tracing

ACM Reference Format:
Rohan Yadav, Michael Bauer, David Broman, Michael Garland, Alex Aiken, and Fredrik Kjolstad. 2025. Automatic Tracing in Task-Based Runtime Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3669940.3707237

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707237

1 Introduction

Implicitly parallel programming systems [3, 10, 12, 26, 42] automatically extract parallelism from a sequential source program through different forms of dynamic dependence analysis. Automatic parallelization and communication inference has enabled composable high-level libraries [7, 41] to be built on top of implicitly parallel task-based runtime systems. However, the cost of the dependence analysis affects the performance of implicitly parallel systems at scale and places a floor on the minimum problem size that can be executed efficiently [34]. Applications with tasks that are too small to amortize the cost of dependence analysis is dominated by it and run at low efficiency.

To improve the performance of implicitly parallel task-based runtime systems, researchers have proposed techniques [24, 25] to memoize, or *trace*, the dependence analysis. Tracing records the results of the dependence analysis for an issued program fragment, and then replays the results of the analysis the next time an identical program fragment is issued. Tracing has been shown to yield significant speedups by eliminating the cost of the dependence analysis on iterative programs. For example, tracing can reduce the per-task overhead in the Legion [10] runtime system from ~1ms to ~100μs [8], widening the scope of applications for which task-based runtime systems can be effective.

A significant limitation of existing tracing techniques is that they require the programmer to annotate repeatedly issued program fragments with stop/start markers for the runtime system. Programmer inserted annotations derail an important feature of implicitly parallel programming systems—their correctness under program composition. As users develop modular programs that pass data from one component to another, the runtime system ensures that computations launched by different modules maintain sequential semantics by implicitly inserting the necessary data movement and synchronization. However, programmer introduced trace annotations do not obey these composition principles, and the correct placement of trace annotations when composing

complex software becomes unclear. Functions defined in a third-party library may contain operations that cannot by traced by a practical tracing implementation, or may issue a different sequence of operations on each invocation. Each of these cases result in runtime errors, due to the incorrect trace annotations constructing an ill-formed sequence of operations. Furthermore, even simple programs using high-level implicitly parallel libraries can have traces that do not correspond to syntactic loop structures in the source program, making it difficult to correctly place tracing annotations. We elaborate on such an example program in Section 2.

In order to improve programmer productivity and to enable the tracing of modular high-level programs, implicitly parallel task-based systems should automatically identify repeated sequences of operations, memoize their analysis results and cheaply replay the analysis as needed. We call this the problem of *automatic trace identification*, which is similar to Just-In-Time (JIT) compilation in the context of dynamic language implementations [18, 20, 28]. JIT compilers for dynamic languages interpret bytecode during program startup, and compile bytecode to native instructions as repeatedly invoked program fragments become hot. Following this architecture, implicitly parallel task-based runtimes should interpret issued operations with a dynamic dependence analysis, and switch to an analysis-free compiled execution once repeated sequences of operations are encountered.

We introduce our system Apophenia[1], that acts as a JIT compiler for the dependence analysis of an implicitly parallel task-based runtime system. The key challenge that Apophenia faces is the *identification* of repeated sequences of operations produced by the target program. Unlike JIT compilers, the input to a task-based runtime system is a stream of tasks that lacks information about control flow such as basic block labels or function definitions. As such, Apophenia cannot rely on these code landmarks or predictable execution flow to identify repeated sequences of operations. Instead, Apophenia analyzes the input stream of operations to find repetitions by solving a series of online string analysis problems.

To demonstrate Apophenia, we develop an implementation within the Legion [10] runtime system as a front-end component that sits between the application and Legion's dependence analysis engine. As operations are issued to Legion, Apophenia performs a series of dynamic analyses to identify repeatedly issued sequences of operations, and correspondingly invokes Legion's tracing engine [24] to memoize and replay dependence analysis on these sequences. While our prototype targets Legion, we believe that the ideas in Apophenia can be directly applied to other task-based runtime systems that perform a dynamic dependence analysis.

The specific contributions of this work are:

1. A formulation of the desirable properties of traces to identify (Section 3).
2. Algorithms to dynamically identify traces in an application's stream of operations (Section 4).
3. An implementation of Apophenia that targets the Legion [10] runtime system.

[1] Apophenia is the tendency to notice patterns between unrelated things.

```python
import cupynumeric as np
# Generate random system.
A = np.random.rand(N,N)
b = np.random.rand(N)
# Initialize solution and
# extract diagonal.
x = np.zeros(A.shape[1])
d = np.diag(A)
R = A - np.diag(d)
# Jacobi iteration.
for i in range(iters):
  x = (b - np.dot(R, x)) / d
```

(a) Python source code.

```
DOT(R, x1, t1)
SUB(b, t1, t2)
DIV(t2, d, x2) # Iteration 1
DOT(R, x2, t1)
SUB(b, t1, t2)
DIV(t2, d, x1) # Iteration 2
DOT(R, x1, t1)
SUB(b, t1, t2)
DIV(t2, d, x2) # Iteration 3
DOT(R, x2, t1)
SUB(b, t1, t2)
DIV(t2, d, x1) # Iteration 4
```

(b) Main loop task stream.

Figure 1. A cuPyNumeric [7] program and the stream of tasks it issues at runtime. An intuitive trace around the main loop does not correspond to a repeated program fragment.

To evaluate Apophenia, we apply it to the largest and most complex Legion applications written to this date, including production-grade scientific simulations and machine learning applications. We show that on up to 64 GPUs of the Perlmutter and Eos supercomputers, Apophenia is able to achieve between 0.92x–1.03x the performance of manually traced code, and is able to effectively trace previously untraced code built from the composition of high-level components to yield end-to-end speedups of between 0.91x–2.82x. As such, Apophenia is able to insulate programmers against the overheads of task-based runtime systems on varying applications and problem sizes, transparently and without programmer intervention.

2 Motivating Example

We now show an example of high-level implicitly parallel code where it is difficult for a programmer to place tracing annotations. As part of developing the example, we provide necessary background on the Legion [10] runtime system.

Figure 1a performs Jacobi iteration using cuPyNumeric [7], a distributed drop-in replacement for NumPy. cuPyNumeric distributes NumPy through a dynamic translation to Legion. cuPyNumeric implements NumPy operations by issuing one or more Legion *tasks*, which are designated functions registered with the runtime system. Each NumPy array is mapped to a Legion *region*, which is a multi-dimensional array tracked by Legion. Each task takes a list of regions as arguments. The stream of tasks launched by the main loop of the cuPyNumeric program is in Figure 1b. For each task, the first two arguments denote the inputs, while the third argument is the output. Legion extracts parallelism from the issued stream of tasks by analyzing the data dependencies between tasks and the usage of their region arguments [9].

To trace a program fragment, the programmer issues a `tbegin(id)` call (standing for "trace begin") before and a `tend(id)` call after the fragment. The first time Legion executes a trace with a particular `id`, it records the results of the dependence analysis, and then replays the results when executing the same trace `id` again [24]. For a trace to be valid, the sequence of tasks and their region arguments encapsulated by `tbegin(id)` and `tend(id)` calls must be exactly the same for a given `id`. The same region arguments must be used across trace invocations as the dependence analysis is affected by the usages of the regions and how they are partitioned. While we consider regions for a Legion implementation of Apophenia, this restriction generalizes to any form of argument that affects the dependence analysis.

A natural attempt to trace the program in Figure 1a would place the `tbegin` and `tend` around the body of the main `for` loop. However, this annotation results in an invalid trace , for a subtle reason that requires knowledge of the internals of cuPyNumeric. The problem with this natural annotation is the loop-carried use of the Python variable `x`, which is bound to different cuPyNumeric arrays (regions) at different points of execution. Upon entering loop iteration i, `x` is bound to a region arbitrarily named `x1`, which is used as an argument for the first `dot` operation. As execution proceeds, cuPyNumeric allocates a new region `x2` for the result of the division with `d`, and binds the Python variable `x` to the region `x2`. Therefore, the next iteration $i + 1$ issues a `dot` on `x2`, causing iteration $i + 1$ to issue a different sequence of tasks than iteration i! Issuing a different sequence of tasks with the same trace `id` is a violation of the conditions to use tracing, and the runtime system may either raise an error or fall back to the expensive dependence analysis. This program illustrates a real-world case where abstraction and composition make it difficult to apply the low-level tracing technique.

To correctly trace the program in Figure 1a, a programmer must either add trace annotations around every two iterations of the main loop, or use two different trace ID's for each different iteration's repetition pattern. This steady state of groups of two iterations is achieved because when `x` is assigned, the region it refers to can be collected and immediately reused by cuPyNumeric. Relying on this steady state is brittle, as the addition of more operations in the main loop or a change in cuPyNumeric's region allocation policy could perturb the way in which the necessary steady state for tracing is achieved. Instead, Apophenia dynamically analyzes the stream of tasks and automatically discovers what fragments of the application should be traced, removing this concern from the programmer.

3 What Are Good Traces?

The overarching goal of Apophenia is to reduce the amount of time the runtime spends performing dynamic dependence analysis by selecting traces to replay. A simple model of a tasking runtime system's dependence analysis is that the runtime spends time α analyzing each task. The first time a trace is issued, the dependence analysis results are memoized, so the runtime spends time α_m (memoization time) on each task in the trace, where α_m is slightly larger than α. Then, on subsequent executions of the trace, there is some constant c amount of overhead to replaying the trace, but every task in the trace only incurs an analysis cost of α_r (replaying time), where $\alpha_r \ll \alpha$.

Using this model of the runtime system, we derive several properties of traces that Apophenia should find. First, the selected traces should maximize the number of traced operations to minimize the number of tasks that contribute an α to the overall analysis cost. Next, the selected traces should be relatively long so that the constant replay cost c does not accumulate. Finally, the set of selected traces should be small, so that Apophenia does not continually memoize new traces and pay α_m per task in each new trace. Intuitively, the ideal set of traces corresponds to the loops in the target program.

We now concretize the good traces that Apophenia should find as the solutions of a concrete optimization problem. Consider the sequence of tasks S constructed from a complete execution of the target program. A system for automatic trace identification must construct from S

- A set of traces T, containing sub-strings of S,
- A function $f : T \rightarrow$ interval set, mapping each $t \in T$ to a set of intervals in S that are *matched* by t,

that maximizes the *coverage* of f, defined by $\text{coverage}(T, f) = \sum_{t \in T} \sum_{i \in f(t)} |i|$, subject to the constraints

1. $\forall t \in T$, t is longer than a minimum length,
2. $\bigcup_{t \in T} f(t)$ is a disjoint set of intervals.

Multiple solutions exist for this problem, so we prefer solutions that first maximize the number of matched intervals ($\sum_{t \in T} |f(t)|$), and then minimize the total number of selected traces ($|T|$). Maximizing $\text{coverage}(T, f)$ directly minimizes the number of untraced tasks, and selecting a small set of traces that repeats many times minimizes the memoization cost of α_m per task. Finally, a minimum length is placed on traces to ensure that the constant replay cost c can be effectively amortized. We present a concrete problem instance and example solutions in Figure 2.

The presented optimization problem precisely defines the properties of traces that a system like Apophenia should attempt to find, but it does not directly yield an algorithm to discover good solutions. Additionally, the optimization problem is structured in a post-hoc formulation, where an optimal solution is constructed from the results of the entire program execution. In practice, a system like Apophenia must construct the solution (T, f) in an online manner, using the currently visible prefix of the sequence of tasks launched by the application. In the next section, we discuss algorithms for dynamically finding good solutions to this optimization problem through a set of string processing algorithms.

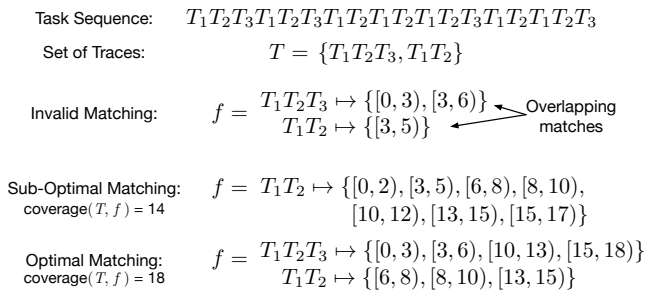


Figure 2. Example of a task stream and fixed trace set T with an invalid matching function f, and two matching functions with different coverage(T, f).

4 Trace Identification

Dynamically finding good traces requires processing information about the tasks seen so far, and then using that information to record and replay traces in the future. An overview of Apophenia's dynamic analysis procedure is sketched in Algorithm 1. Apophenia has two components that correspond to the targets of the optimization problem in Section 3. The *trace finder* constructs the candidate set of traces T by accumulating the tasks issued by the application into a buffer, and asynchronously mining the buffer to find candidate traces. The *trace replayer* then constructs the matching function f by ingesting the candidate traces into a trie, and identifying candidate traces in the application stream by maintaining pointers into the trie that represent potential matches. Apophenia intercepts calls to target runtime's `ExecuteTask` function, and forwards a potentially different set of tasks and trace markers to the runtime. A concrete example of how Apophenia identifies a trace in an application is shown in Figure 3. We now describe each of these components in detail.

4.1 A Stream of Tokens

An insight of our work is that automatic trace identification is inherently an online string analysis problem of finding repeated sub-sequences in the application's task stream. As seen in Figure 1b, the task stream is not just a list of identifiers—tasks have arguments that must also be the same across iterations to be used in a trace. To capture all aspects of a task that can affect the dependence analysis, Apophenia constructs a hash from each task and its region arguments. Converting the input stream of tasks into a stream of hash tokens enables more direct application of string processing techniques, and straightforward handling of traceable operations that are not tasks.

4.2 Finding Traces With High Coverage

Apophenia's trace finder records tasks as they are issued by the application into a buffer (we describe a refinement to this scheme in Section 4.4). Once the buffer fills up, Apophenia launches an asynchronous analysis of the buffer to find a set of traces within the buffer that maximize the coverage of the buffer. We discuss previous ideas that are related to this goal, and then describe the solution used in Apophenia.[2]

Algorithm 1: Apophenia's Dynamic Analysis.

```
/* Initialize token history buffer B and pending async
   analyses J. */
B, J ← [], []
/* Initialize trie of candidates C, potential current
   traces A, and pending tasks P. */
C, A, P ← Trie(), [], []
/* Discussed in Section 4.2. */
TraceFinder (H)
    B ← B + [H]
    if ShouldAnalyzeHistory(B) then
        /* What subset of the history to analyze is
           discussed in Section 4.4. */
        B′ ← GetAnalysisSubset(B)
        /* Find repeated sub-strings. */
        j ← async FindRepeats(B′)
        J ← J + [j]
        B ← MaybeClearHistory(B)
/* Discussed in Section 4.3. */
TraceReplayer (T, H)
    if ∃ j ∈ J, j is complete then
        IngestCandidates(j, C)
    P ← P + [T]
    /* Advance all potential traces by H in the trie
       if possible. Remove impossible traces, and
       extract fully matched candidates. */
    A ← AdvanceActiveCandidates(C, A, H)
    A ← FilterInvalidCandidates(C, A)
    D, A ← FilterCompletedCandidates(C, A)
    if |D| > 0 then
        /* Select one of the pending candidates to
           replay. Execute any tasks before it, and
           issue a trace replay for the candidate. */
        R ← SelectReplayTrace(D, P, A)
        P, A ← ExecuteAndReplay(R, P, A)
/* Applications issue tasks through Apophenia's
   ExecuteTask function. */
ExecuteTask (T)
    H ← Hash(T)
    TraceFinder(H)
    TraceReplayer(T, H)
```

Existing Techniques. The Lempel-Ziv family of algorithms use repeated sub-strings for compression. Algorithms like LZ77 [35, 44, 45] maintain a sliding window of previous tokens to search for repeats in when encoding upcoming tokens. The LZW [39] algorithm avoids the use of a sliding window by only increasing the length of any candidate repeat by a single token at a time. While not directly finding a set of repeats with high coverage, similar algorithms that use a sliding window would need to maintain and search in a

[2]We discuss more related work in Section 7.

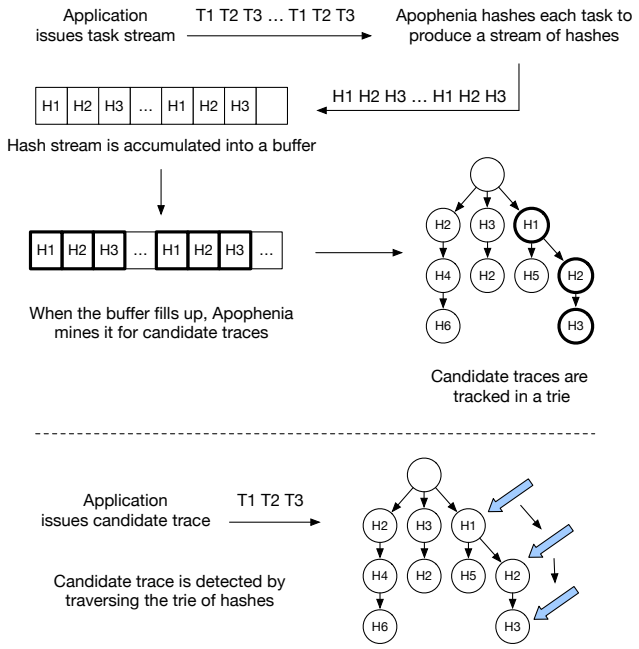


Figure 3. Visualization of Apophenia's dynamic analysis.

window the size of the analyzed buffer, resulting in quadratic time complexity. In order to recognize a trace of length n, an LZW-style algorithm would also need to encounter the trace $n-1$ times. We wanted an algorithm that is sub-quadratic in order to scale to large buffer sizes. Real-world applications we discuss in Section 6 have traces that contain more than 2000 tasks, requiring token buffers of at least twice that size to detect a single repeat.

Within the programming languages community, recent work by Sisco et al. [33] used a technique called *tandem repeat analysis* [36] to find loops in the netlists that result from compiling hardware description languages. A tandem repeat is a sub-string α that repeats contiguously within a larger string S, such that α^k is a sub-string of S, for some k. Despite the success that Sisco et al. had using tandem repeat analysis, we found that even simple real world cuPyNumeric programs did not contain enough tandem repeats for the analysis to reliably identify a trace set with high coverage. The reason is that while these real-world programs tended to have repetitive main loops, there would often be irregularly appearing computations such as convergence checks or statistics calculations that occur infrequently between loop iterations. As such, the strings that represented these programs would not contain tandem repeats, but instead repeated sub-strings separated by other tokens.

A relaxation of tandem repeat analysis is to search for non-overlapping repeated sub-strings, which removes the contiguity requirement on the repeats. Concretely, given the string *ababab*, *abab* is an overlapping repeat, while *ab* is non-overlapping. We could use non-overlapping repeated sub-strings to assemble a set of traces T and a disjoint mapping f that achieves high coverage. While there exist standard suffix-tree algorithms to find repeated sub-strings, we found that the natural extensions of these algorithms to detect non-overlapping repeated sub-strings also resulted in quadratic runtime complexity.

Our Algorithm. In this work, we design a repeat finding algorithm that is directly aware of the optimization problem in Section 3 and runs in $O(n \log(n))$, where n is the size of the token history buffer. At a high level, our algorithm makes a pass through a suffix array constructed from the input buffer to collect a set of candidate repeats. It then greedily selects the largest repeated sub-strings that do not overlap with any previously chosen sub-strings. Psuedocode for our algorithm is in Algorithm 2[3], which takes a string S and returns a set of sub-strings that achieve high coverage of S. We assume that the reader is knowledgeable about suffix arrays and their structural properties. However, understanding the algorithm in Algorithm 2 is not required to understand its usage in Apophenia, as discussed in Section 4.3 and Section 4.4.

As a first step, we construct a suffix array and longest common prefix array from the input buffer of tokens. We then iterate through adjacent pairs of suffixes to construct a set of *candidate repeats*, which are tuples of sub-strings defined by their length, the repeated sub-string, and its starting position in S. These candidates are constructed based on whether or not the shared prefix between adjacent suffix array entries overlap. Once all of the candidates have been constructed, we sort the candidates to greedily select candidates in order of length, and select as many occurrences of a particular sub-string as possible. We only select candidates that do not overlap with any previously selected candidates, and then deduplicate the chosen set of candidates as the result. A sample execution of Algorithm 2 is shown in Figure 4.

Our algorithm can be implemented with time complexity $O(n \log(n))$. Linear time algorithms exist for suffix array and LCP array construction [23]. Two candidates are generated for each entry in the suffix array, so sorting the candidates takes $O(n \log(n))$ time. The interval intersection step can be reduced to constant time by leveraging the candidate iteration order, so the entire loop executes in $O(n)$ time. In particular, an array of length $|S|$ can be maintained, and as each candidate is selected, all positions covered by the candidate are marked. Then, as candidates are iterated over in decreasing length and increasing start position order, interval intersection can be checked by checking if the start or end of an interval is marked. Finally, the deduplication can be done by generating a unique ID for each candidate sub-string in the candidate generation phase, and adjusting

[3]We also present a standalone implementation of the algorithm available at https://github.com/david-broman/matching-substrings.

Algorithm 2: Non-overlapping repeated sub-strings.

```
1  FindRepeats (S)
2      SA, LCP ← SuffixArray(S)
       /* Candidates are tuples of string length, the
          repeated sub-string, and starting position. */
3      C ← [ ]
4      foreach i ∈ [0, |SA| − 1) do
           /* Extract adjacent suffix array entries and
              their overlap length. */
5          s1, s2, p ← SA[i], SA[i + 1], LCP[i]
6          if [s1 : s1 + p) ∩ [s2 : s2 + p) = ∅ then
               /* S[s1 : s1 + p] and S[s2 : s2 + p] are
                  repeated strings that do not overlap in
                  S, so they are candidates. */
7              r ← S[s1 : s1 + p]
8              C ← C + [(p, r, s1), (p, r, s2)]
9          else
               /* S[s1 : s1 + p] and S[s2 : s2 + p] overlap in
                  S. Assume s2 > s1, the other case is
                  symmetric. In this case, the overlap is
                  a collection of repeats of S[s1 : s1 + d],
                  by the structure of the suffix array. */
10             d ← s2 − s1
               /* Break prefix into two chunks of
                  repeated pieces of S[s1 : s1 + d]. */
11             l ← (p + d)/2
               /* Remove trailing tokens. */
12             l ← l − (l%d)
13             r ← S[s1 : s1 + l]
14             C ← C + [(l, r, s1), (l, r, s1 + l)]
       /* Sort the candidates by decreasing length and by
          increasing sub-string and start position. */
15     Sort(C)
       /* Greedily collect sub-strings that do not
          overlap with previously chosen sub-strings. */
16     I, R ← [ ], [ ]
17     foreach (l, _, s) ∈ C do
18         if [s, s + l) does not intersect I then
19             I ← I + [[s, s + l)]
20             R ← R + [S[s : s + l]]
21     return R
```

the candidate representation to be a tuple of length, ID and starting position; using this sort order allows deduplication to be done at each iteration of the candidate selection loop.

Our algorithm aims to find good solutions to the optimization problem in Section 3 by identifying long repeated sub-strings and selecting as many as possible that do not overlap with each other. We trade off between an optimal solution to the optimization problem to instead find good solutions and maintain a lower asymptotic runtime. There are two such heuristics in our algorithm. First, when adjacent suffix array entries have a repetition, we consider only the maximal length repetition instead of all sub-strings of the repetition. Second, when we select which candidates to keep, we greedily choose the largest candidates instead of performing a bin-packing computation. Our algorithm

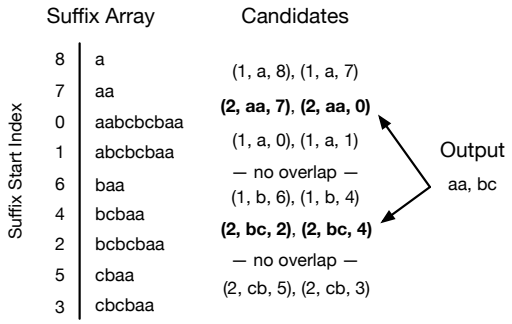


Figure 4. Execution of Algorithm 2 on "aabcbcbaa". The candidates for each suffix pair is shown between the pair.

is guaranteed to find the longest repeated sub-string, but due to the second heuristic, we cannot provide theoretical guarantees about the other chosen sub-strings. We show in Section 6 that Apophenia using our algorithm is able to identify good traces in complex, real-world applications.

4.3 Recognizing and Replaying Candidate Traces

Apophenia's trace replayer uses Algorithm 2 to find candidate traces from the application's history of tasks. In this section, we discuss how Apophenia's trace replayer identifies and selects these candidate traces from the task stream to record and replay. Our design of the trace replayer has two major goals. First, the per-task overhead must be low, as it is imperative for performance for the application to issue as many tasks into the runtime as possible so that the runtime can either replay traces or perform dependence analysis ahead of execution. Slowing down the task launch rate would result in exposed latency from various sources in the runtime. Second, Apophenia must balance exploration and exploitation when selecting traces. As more information about the application is gained, Apophenia should switch to better traces as it finds them. However, Apophenia should not leave a steady state of replaying a particular trace until it is confident that performance can be improved, as memoization of the dependence analysis for new traces has a cost.

As discussed previously, Apophenia accumulates a history of tasks launched by the application and asynchronously uses Algorithm 2 to select candidate traces. Asynchronous analysis of task histories is important to avoid stalling the application by waiting for the analysis to finish before accepting the next task from the application.

When an asynchronous analysis completes, Apophenia ingests the results into a trie that maintains the current set of candidate traces. Along with this trie, Apophenia maintains a set of pointers into the trie that represent potential matched traces. As tasks are issued, Apophenia updates the set of pointers by creating new pointers for each new task, stepping any existing pointers down the trie if possible, and removing

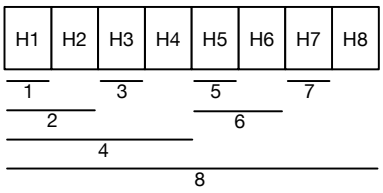


Figure 5. Visualization of Apophenia's buffer sampling strategy on a buffer of size 8. After processing the i'th task, Apophenia mines the buffer slice labeled i.

any pointers that are made invalid. Once a pointer reaches a leaf of the trie and has matched a trace, Apophenia has the option to forward the trace to the tasking runtime, wrapped by `tbegin` and `tend` calls.

Apophenia uses a scoring function to select which matched trace to replay when faced with multiple valid choices. The scoring function is based on the length of the candidate trace multiplied by a count of the number of times the trace has appeared. In calculation of the score, we impose a maximum value of the count that can be used, and exponentially decay the value of the count by how many tasks have been encountered since the trace last appeared. Finally, we increase the score slightly if a trace has already been replayed.

Our scoring function encodes heuristics about trace selection and aims to balance exploration and exploitation. We naturally prefer long traces over shorter ones, as longer traces have the potential to eliminate more runtime overhead. The capping of the appearance count allows for Apophenia to eventually switch from a trace that appeared early during program execution to a better trace that appears later in the execution. Next, decaying the appearance count ensures that a seemingly promising trace that occurs infrequently, does not eventually hit a threshold, and disrupts a steady state. Finally, since recording new traces has a cost, when faced with traces of a similar score, we bias Apophenia towards a trace it has already replayed.

4.4 Achieving Responsiveness and Quality

Apophenia's trace finder accumulates tasks into a buffer and mines the buffer for traces using Algorithm 2. An important question is what should the size of that buffer be? The size of this buffer trades off between responsiveness of the Apophenia's trace identification and the quality of traces Apophenia is able to find. With a small buffer, Apophenia can identify traces early but will not be able to identify traces in programs with large loops. Meanwhile, a large buffer allows Apophenia to identify long traces in complex applications but introduces significant startup delay in smaller applications.

We did not want end users to be required to continually adjust the buffer size parameter as their application changes. As such, some strategy to adapt the buffer size along this tradeoff space is necessary. We found that a strategy that attempts to dynamically resize the buffer based on what traces to find is unsatisfactory, as the system is unable to differentiate between an application currently not repeating operations versus an application repeating a sequence of operations larger than the buffer size. Instead, we propose a strategy that selects a large fixed buffer size upfront, and then samples smaller pieces of the buffer in a principled manner to be responsive to the occurrence of short traces.

Apophenia samples from the buffer guided by the *ruler function* sequence [40], which provides a practically useful sampling strategy with provable guarantees. The ruler function counts the number of times a number can be evenly divided by two. Applying it to the sequence 1, 2, 3, 4, . . . yields the sequence 0, 1, 0, 2, ... Raising the sequence to the power two yields 1, 2, 1, 4, . . ., which we can interpret as subsets of the buffer to analyze. For example, with a buffer size of four, as tasks arrive Apophenia would first analyze the first task, then the first two tasks, then the third task, and finally all four tasks. A visualization of this sampling policy is in Figure 5. This sampling policy lets Apophenia quickly react to changes in the application by analyzing recent pieces of the buffer while allowing larger traces to be found by infrequently analyzing longer components of the buffer. For example, sampling the full buffer in Figure 5 is required to find a trace that repeats in positions H2-H4 and H5-H7. In practice, we use the exponentiated ruler function as the multiples of a larger constant (such as 250) to sample the buffer with. Finally, given that our algorithm in Section 4 runs in $O(n \log(n))$, we show that our sampling strategy increases the total runtime complexity of processing the buffer by only an extra log factor, yielding a total of $O(n \log^2(n))$. This technique enables all of the experiments in Section 6 to be run with the same buffer size configuration parameter.

5 Implementation Discussion

We now discuss important aspects of a realistic implementation of Apophenia. In particular, we discuss the specifics of implementing Apophenia in a distributed context and a decision not to perform speculation when replaying traces.

5.1 Distributing the Analysis

Apophenia's analysis as presented in Section 4 is sequential, processing tasks as they are issued by the application. In a distributed setting, Apophenia leverages Legion's *dynamic control replication* [8] to act as a sequential analysis, except for one component, which we discuss next. With control replication, the application executes on each node and Legion shards the dependence analysis and execution across nodes. The main restriction of control replication is that the application must issue the same sequence of tasks on every node. We implement Apophenia as a layer between the application and Legion, meaning that Apophenia intercepts calls into the Legion runtime from the application and forwards a

(possibly different) set of calls into Legion. As such, Apophenia inherits the control replication requirements of the application. In particular, each node must agree on which traces to replay and when during program execution to record and replay the traces.

The only source of non-determinism in Apophenia that may result in divergent decisions between nodes is the asynchronous processing of token buffers described in Section 4.2. An instance of Apophenia exists on each node of the target machine, and each instance maintains a local history buffer of tasks to run asynchronous analyses on. The asynchronous analysis may complete earlier on one node than another, resulting in that node replaying a trace before another node has identified that trace as a candidate. However, making the analysis synchronous would result in stalling the application until analyses complete. We resolve this tension by having each node agree on a count of processed operations to issue before ingesting the results of an asynchronous analysis. If any node had to wait on an asynchronous analysis to complete, all nodes increase their count of operations to wait on for the next analysis. This strategy reaches a steady state where analysis results are ingested in a deterministic manner without stalling the application.

5.2 (The Lack of) Speculation

Speculation is a common technique in computer architecture to efficiently execute programs with data-dependent control flow. As Apophenia has similarities to speculative components in architecture like trace caches (Section 7), a natural design decision was if Apophenia should speculate on whether traces would be issued by the application. Our implementation of Apophenia does not speculate and waits for the entirety of a trace to arrive before issuing the trace to Legion. The relative costs of different operations within the Legion runtime system made the potential upside of speculation not worth the implementation complexity.

Legion employs a pipelined architecture where a task flows through three stages: 1) the application phase, where the task is launched (into Apophenia), 2) the analysis phase, where the task is analyzed or replayed as part of a trace, and 3) the execution phase, where the task is executed. Depending on the cost ratio of the application and analysis phases, speculation may be beneficial as Apophenia waits for an entire trace to pass through the application phase. Legion's analysis phase is an order of magnitude more expensive than the application phase, letting the application phase run far ahead of the analysis phase. Thus, waiting for an entire trace to be issued by the application rarely stalls the pipeline and gets exposed in the overall runtime. Thus, we found that designing a trace prediction algorithm and implementing a backup-rollback-recover scheme on speculation failures was not worth the complexity.

6 Evaluation

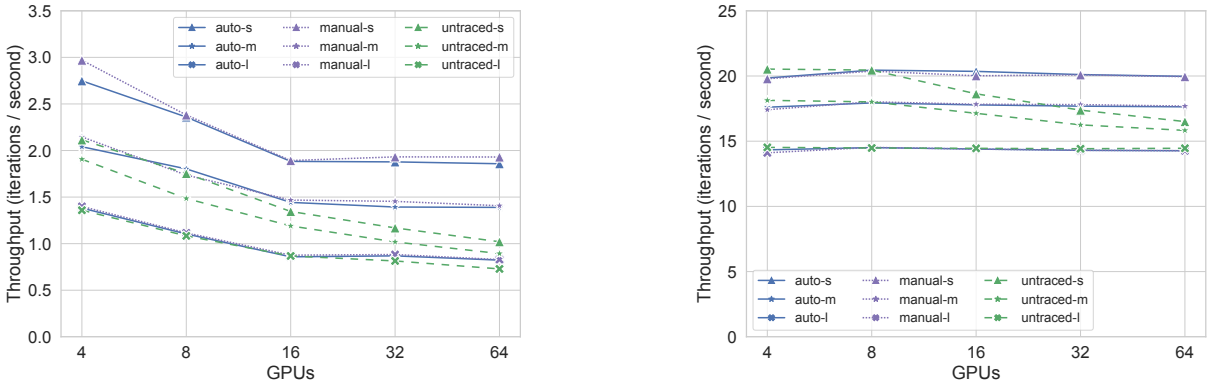
Overview. We evaluate Apophenia on the largest and most complex Legion applications written to date, including production scientific simulations and a distributed deep learning framework. Our results show that Apophenia is able to effectively find traces in complex programs with lower overhead, enabling programmers to experience the benefits of tracing without manual effort and allowing a more general set of applications to be traced.

Experimental Setup. We evaluated Apophenia on the Eos and Perlmutter supercomputers. Each node of Eos is an NVIDIA DGX H100, containing 8 H100 GPUs with 80 GB of memory and a 112 core Intel Xeon Platinum. Each node of Perlmutter contains 4 NVIDIA A100 GPUs with 40 GB of memory and a 64 core AMD EPYC 7763. Nodes of Eos are connected with an Infiniband interconnect, while Perlmutter uses a Slingshot interconnect. We compile Legion on Eos with the UCX networking module, and use the GASNet-EX [11] networking module on Perlmutter. We do not execute each application on both Perlmutter and Eos due to differences between the local environments on each machine. In our experiments, we evaluate the relative performance differences between traced and untraced programs, and comparisons between machines are not significant.

6.1 Weak Scaling

In this section, we discuss weak scaling results of applications using Apophenia, as shown in Figure 6 and Figure 7. In a weak scaling study, we increase the problem size as the size of the target machine grows to keep the problem size per processor constant. For each application, we perform a sweep over different sizes of the problem to vary the task granularity, thus affecting the impact of runtime overhead. These different problem sizes are denoted in the graph by the "-s", "-m" and "-l" label suffixes which stand for small, medium and large. At smaller problem sizes, more runtime overhead can be exposed, while larger problem sizes make it easier to hide runtime overhead. In each weak-scaling plot, we report the steady-state throughput of each configuration and problem size after a number of warmup iterations (discussed in Section 6.3). We report throughput in iterations per second achieved by each configuration, so within a particular problem size, higher is better; across problem sizes, the smaller problem sizes will achieve a higher iterations per second than the larger problem sizes.

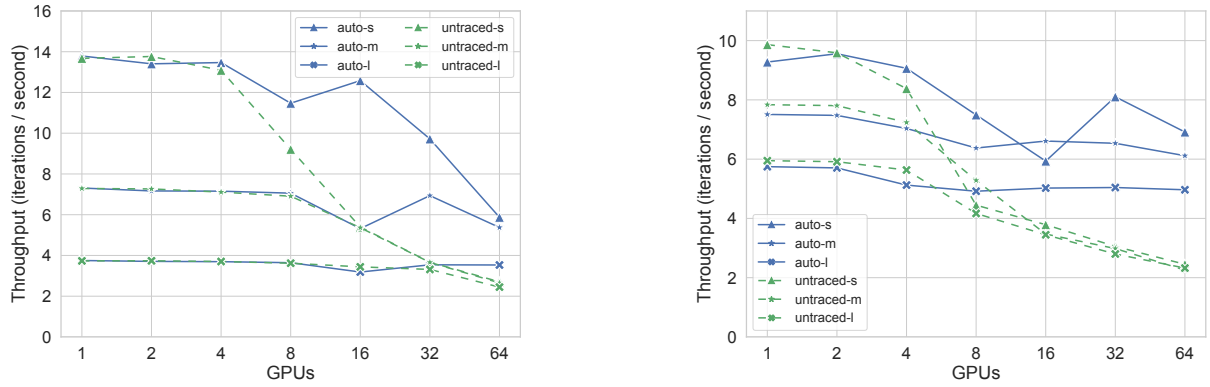
S3D. S3D [37] is a production combustion chemistry simulation code that has been developed over the course of many years by different scientists and engineers. The Legion port of S3D implements the right-hand-side function of the Runge-Kutta scheme, and interoperates with the legacy Fortran+MPI driver of the simulation. The integration between



(a) S3D (Perlmutter)

(b) HTR-Solver (Perlmutter)

Figure 6. Weak scaling on previously traced Legion applications, where Apophenia ("auto") performs competitively.



(a) CFD (Eos)

(b) TorchSWE (Eos)

Figure 7. Weak scaling on cuPyNumeric applications, where Apophenia ("auto") outperforms the untraced version.

Legion and the legacy Fortran+MPI code leads to various constraints that the manual trace annotations interact with. For example, during the first 10 iterations, a hand-off between Legion and Fortran+MPI must occur every iteration, while after the first 10 iterations a hand-off is required only every 10 iterations. While not unmanageable, these interactions have led to relatively complicated logic to manually trace the main loop. We scale S3D on Perlmutter, and compare the performance of Apophenia to manually traced and untraced versions of S3D. The results are shown in Figure 6a. Even on a single node, tracing has a noticeable performance impact on the smaller problem sizes and affects the scalability of S3D. Apophenia achieves within 0.92x–1.03x of the performance of the manually traced version, and between 0.98x–1.82x speedups over the untraced version. Manual annotations can slightly outperform Apophenia by leveraging application knowledge to select traces that have lower replay overhead.

HTR. HTR [17] is a production hypersonic aerothermodynamics application. HTR performs multi-physics simulations of hypersonic flows at high enthalpies and Mach numbers, such as for simulations of the reentry of spacecraft into the atmosphere. Like S3D, we evaluate Apophenia's performance on HTR on Perlmutter, and compare it against a manually traced version and an untraced version. While HTR without tracing performs competitively to the traced version at small GPU counts, Figure 6b shows that tracing is necessary for performance at scale. Apophenia achieves within 0.99x–1.01x of the performance of the manually traced version, and between 0.96x–1.21x speedups over the untraced version.

CFD. CFD is a cuPyNumeric application that solves the Navier-Stokes equations for 2D channel flow [5]. Unlike S3D and HTR, there is not a manually traced version of CFD, due to the difficulties around composition discussed in Section 2. Developing a manually traced implementation of CFD would

require either rewriting the application to remove any dynamic region allocation, or manual examination of allocator logs to find the number of iterations in the steady state. As a result, we compare CFD with Apophenia to the standard untraced version on different problem sizes, which is the performance that cuPyNumeric users are able to achieve today.

Figure 7a shows weak scaling results for CFD on Eos. These results are similar to HTR, where leveraging tracing is necessary for performance at scale. On the smallest problem size, even though the tracing removes a large amount of runtime overhead, the tasks are too small to hide the communication latency at larger scales, leading to the observed fall off in performance. On larger problems, CFD with Apophenia is able to maintain high performance while the untraced version falls off, yielding between 0.92x–2.64x speedups.

TorchSWE. TorchSWE is a cuPyNumeric port of the MPI-based TorchSWE [13] shallow-water equation solver, and is the largest cuPyNumeric application developed so far. Similarly to CFD, there is no manually traced version to compare to. However, unlike CFD, performing a rewrite of TorchSWE to enable manual tracing would be difficult, as TorchSWE contains an order of magnitude more lines of code. Weak scaling results for TorchSWE on Eos are shown in Figure 7b, which show that TorchSWE is significantly impacted by Legion runtime overhead without tracing.

These results demonstrate that there does not exist a problem size for TorchSWE on Eos that can hide Legion's runtime overhead without tracing. Even the large problem size, which nearly reaches the GPU's memory capacity, exposes Legion runtime overhead at 8 GPUs. The reason for this is that TorchSWE maintains a large number of fields for each simulated point, and issues different array operations on each field. The amount of data needed for each element in the simulation does not allow the task granularity to be easily increased to the untraced Legion minimum of $\tilde{1}$ms per task, as each new element added increases the memory footprint more than it increases the average task granularity. For such applications, leveraging tracing is a requirement, and Apophenia enables complex applications like TorchSWE to do so automatically. TorchSWE itself contains enough task parallelism to hide communication latencies, but needs tracing to first lower runtime overhead. With Apophenia, we are able to achieve between 0.91x–2.82x speedup on TorchSWE, achieving nearly perfect scalability on 64 GPUs.

6.2 Strong-Scaling

We now move from scientific simulation codes to distributed deep neural network training with FlexFlow [22, 38]. FlexFlow is a deep neural network framework that searches for hybrid parallelization strategies for different layers of the network. We perform a strong-scaling experiment with FlexFlow on Eos to train the largest (`pilot1`) network from the CANDLE [1] initiative[4]. A strong-scaling study fixes the problem size on a single processor, and increases the number of processors while keeping total problem size constant. To strong scale the training, we fix the batch size for single GPU, and then increases the number of GPUs available.

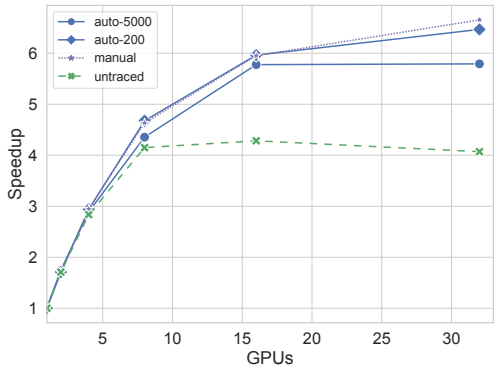


Figure 8. Strong scaling of FlexFlow on Eos.

We compare the performance of FlexFlow with manual trace annotations, two configurations of Apophenia (discussed next), and no tracing. As seen in Figure 8, as FlexFlow scales up, the tasks become smaller and begin to expose Legion runtime overhead without tracing, leading to slowdowns when scaling up. The two configurations of Apophenia differ in the maximum trace length to be replayed (Apophenia's history buffer is the same, but recorded traces are broken into pieces of a given maximum size). The first (auto-5000) is the standard configuration with no maximum, as used in all other experiments, and the second (auto-200) has a maximum length of 200 tasks, which is similar to the length of the manually annotated trace. As FlexFlow strong scales, the cost of Legion issuing the trace replay starts to become exposed as the execution time of the trace decreases, leading to shorter traces exposing less latency, and thus performing better[5]. On 32 GPUs, the configuration of Apophenia with a maximum trace length of 200 achieves between 0.97x the performance of the manually traced FlexFlow, and achieves a 1.5x speedup over the untraced FlexFlow.

6.3 Overheads of Apophenia

We now discuss the overheads that Apophenia imposes over standard execution with Legion. While we inherit the overheads of Legion's existing tracing infrastructure [24] (the cost of memoizing traces), Apophenia imposes two new

[4]Due to engineering limitations in FlexFlow at the time of writing, the network was parallelized only with data parallelism.
[5]The Legion team is aware of this shortcoming and plans to address it in the future.

sources of overhead to measure: 1) the overhead on task launches and 2) the time taken until a steady state is reached.

As discussed in Section 4, Apophenia intercepts the application's task launches and performs some analysis work before forwarding the task launches to Legion. This analysis work includes launching asynchronous token buffer processing jobs and manipulating traversals of the trie data structures used for online trace identification. To quantify this overhead, we ran a two node experiment on Perlmutter and measured the time it took to launch (not analyze or execute) Legion tasks with and without Apophenia enabled. We ran a two node experiment to ensure that the coordination logic discussed in Section 5.1 was included in timing. We found that task launching took on average 7μs without Apophenia, and on average 12μs with Apophenia. While Apophenia increases the task launch overhead, this overhead is still significantly lower than the amount of time it takes to replay a task as part of a trace, which is 100μs. As such, the task launching cost of Apophenia can still be effectively hidden by the asynchronous runtime architecture. The asynchronous analysis jobs that Apophenia launches to process task histories do not affect the critical path, and utilize Legion's background worker threads. While in theory these jobs could compete for the resources necessary for Legion's dependence analysis, we have not yet encountered an application where they caused a detriment in performance.

To measure the time taken until Apophenia reaches a steady state of replaying traces on our iterative applications, we report the number of iterations until a steady state is reached. Figure 9 contains the iteration counts needed for each application in Section 6.1 and Section 6.2, which range from 30 to 300. These simulation and machine learning workloads would be run in production for a significantly larger number of iterations, so speedup in the steady state corresponds closely to end-to-end speedup. We note that the cuPyNumeric applications have a larger number of required warmup iterations due to the dynamic behavior discussed in Section 2, where a single application-level iteration of the program does not necessarily correspond to a repeated sequence of tasks.

In terms of resource utilization, Apophenia requires a modest amount of CPU memory to store the history buffer of tasks for analysis. Apophenia runs the asynchronous string analysis (Section 4.2) on Legion's background worker threads. We have not found these resource requirements to impact application performance or memory utilization.

6.4 Trace Search

To give intuition about the search process that Apophenia performs, we constructed a visualization of the amount of runtime overhead that Apophenia is removing over time. Figure 10 is a visualization of S3D over time (for 70 iterations), where each for task launched by S3D, we display how many

Application	Iterations Until Steady State
S3D	50
HTR	50
CFD	300
TorchSWE	300
FlexFlow	30

Figure 9. Warmup iterations before Apophenia reaches a replaying steady state.

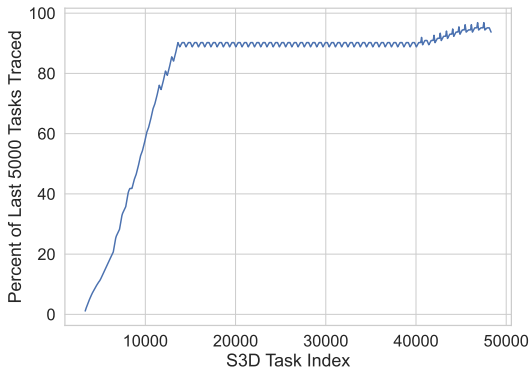


Figure 10. Visualization of Apophenia finding traces in S3D.

of the previous 5000 tasks were traced. For iterative computations, this procedure yields the expected result, where Apophenia spends time during program startup discovering new traces, and then settles into a steady state. The amount of traced operations increases slightly by the end of the execution, as Apophenia finds a better set of traces that lowers the number of untraced operations.

7 Related Work

Just-In-Time Compilers. Just-In-Time (JIT) compilers [18, 20, 28] for dynamic languages have a tiered execution system, where the target language is first translated to bytecode, which is executed by an interpreter. Frequently executed program fragments are then compiled into native instructions for significantly faster execution. Apophenia employs a similar architecture where a task-based runtime system's dynamic analysis acts as the slow but general interpreter, and uses a tracing engine as the fast but specialized compiler.

Tracing-based JIT compilers such as TraceMonkey [19] record sequences of instructions executed at runtime and generate optimized code for those sequences. Method-based JIT compilers identify frequently invoked functions in the target program and compile type-specialized versions of those functions. JIT compilers identify the desired instruction sequences or methods to compile by relying on code landmarks like function definitions and basic block addresses to maintain counters of frequently executed program fragments. Since Apophenia views an unrolled stream of tasks, it

must employ novel techniques for identification of traceable program fragments.

JIT compilers also perform dynamic analysis to recover data structures like call-graphs from the target program. Sampling-based methods [43] have been developed to balance runtime cost of profiling each function call with the accuracy of the sampled data structure. Discovering traces in our work requires for long contiguous sequences of issued tasks to be analyzed together, as a trace must repeat several times to be considered by Apophenia. Breaking up these sequences with independent and non-contiguous samples can lead to a loss of precision when discovering traces. Instead, Apophenia employs an always-on approach where all tasks are analyzed, and uses a sub-sampling method on the set of collected tasks to manage the trade off between responsiveness of the trace analysis and length of the discovered traces. An always-on approach is cheaper to use in the task-based runtime system context than within a standard JIT compiler as tasks are relatively coarse when compared to bytecode instructions.

Trace Caches. Trace caches [31] have been used in processors to improve instruction fetching bandwidth. At a high level, trace caches record the common jump paths taken through basic blocks, and pre-fetch those paths when revisiting the same basic blocks. Apophenia shares a similar architecture to trace caches, which also use patterns in running programs to improve the performance of a slower dynamic component (in this case, the control-dependent instruction fetching). Similarly to JIT compilers, trace caches also use landmarks in executing programs to guide their decisions, which Apophenia is not able to exploit. Also, by virtue of being implemented in hardware, the mechanisms that trace caches must be simpler than the kinds of analyses Apophenia can use, which are implemented in software.

String Analysis. Section 4.2 contains a partial discussion of related string analysis works—we continue the discussion here. The most relevant string processing problem in the bio-informatics community is *motif finding* [14], which is the problem of finding short (5–20 token long), fixed-length repeated strings in a larger corpus. The focus on a short and fixed sub-string length and a tendency to use genomic information to guide the search makes these techniques not applicable to our problem. Algorithms for document fingerprinting such as Moss [32] have been developed that accurately identify copies between documents. In particular, these techniques are guaranteed to detect if repetitions of at least a minimum size exist across documents. Fingerprinting techniques are useful to detect whether there exist repeated sub-strings, but do not directly aid in finding the sub-strings themselves that have high coverage.

Inspector-Executor Frameworks. Apophenia is similar in spirit to Inspector-Executor (I/E) frameworks that dynamically analyze program behavior and then perform optimizations [29, 30]. I/E frameworks generally focus on recording information related to array accesses and use knowledge of these accesses to perform compiler optimizations that parallelize or distributed loops. In contrast, Apophenia observes a dynamic sequence of tasks and searches for repeated sub-sequences of tasks to record as traces.

Task-Based Runtime Systems. Several task-based runtime systems have been developed for high performance computing [3, 10, 12], data science [16, 42], and machine learning [6, 26]. One axis of runtime overhead that these different systems impose on applications is the cost of dependence analysis. The cost of dependence analysis is directly related to the expressivity and flexibility of the runtime system's programming model. Legion has an expressive data model that supports *content-based coherence* [9], leading to a relatively expensive dependence analysis. As a result, tracing [24] was developed to reduce the costs of the dependence analysis. Both the StarPU [2] and PARSEC [21] runtime systems have modes that perform a dynamic dependence analysis to extract parallelism, and these modes have been shown to add overheads over the explicitly-parallel, analysis free modes [34]. Tracing techniques could be applied within these runtime systems to lower the overheads of the dynamic, implicitly parallel modes.

Techniques similar to tracing have also been developed in other runtime systems to lower overheads. A tracing-like technique called Execution Templates [25] was developed to cache control plane decisions in runtime systems for cloud-based environments. The Dask [16] runtime system exposes an API for users to explicitly construct and optimize task graphs [15], which is lower-level but more efficient than the standard individual task launching API. The Ray [26] runtime system has recently added an execution mode called "Compiled Graphs" [4], where users build explicit computation graphs and issue them to Ray for lower overhead replay. Finally, the CUDA runtime exposes a similar feature to tracing called CUDA Graphs [27], where users may record a sequence of CUDA kernel launches and replay the sequence with lower overheads. Techniques used in Apophenia could potentially be applied to these systems to remove the requirement for users to be involved in the memoization and optimization of these computations.

8 Conclusion

In this work, we introduce Apophenia, a system and framework for task-based runtime systems to automatically trace the dependence analyses for repeated program. By automatically detecting traces, Apophenia is able to improve programmer productivity by insulating programmers against changing task granularity, and enable new applications to

take advantage of tracing. We develop an implementation of Apophenia that targets the Legion runtime system and show that on the most complex Legion applications written to this date, Apophenia is able to match the performance of manually traced code, and effectively optimize currently untraceable programs to improve the performance at scale by up to 2.82x.

Acknowledgements

We thank Wonchan Lee, Manolis Papadakis and Shriram Jagannathan for their assistance with Legate. We thank Seshu Yamajala for his assistance in running the S3D simulation. We thank Elliot Slaughter for his assistance in debugging and running Regent programs. We thank Mario Di Renzo and Caetano Melone for their assistance in running the HTR simulation. We thank Zhihao Jia and Colin Unger for their assistance in running FlexFlow. We thank Roshni Sahoo for her assistance in developing formal optimization problem in Section 3. We thank Danny Sleator and Sam Westrick for suggestions and pointers to related work around the string analysis component of this work. We thank Wei Wu for discussions about the PARSEC runtime system, and Cedric Augonnet for discussions about the StarPU runtime system. We thank (in no particular order) James Dong, AJ Root, Chris Gyurgyik, Rubens Lacouture, Shiv Sundram, Scott Kovach and Olivia Hsu for their discussions and feedback on this manuscript. Rohan Yadav was supported by an NVIDIA Graduate Fellowship, and part of this work was done while Rohan Yadav was an intern at NVIDIA Research. This work was in part supported by the National Science Foundation under Grant CCF-2216964 and by Digital Futures at KTH Royal Institute of Technology. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 using NERSC award ASCR-ERCAP0026353.

References

[1] [n. d.]. CANDLE | Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer — wordpress.cels.anl.gov. https://wordpress.cels.anl.gov/candle/. [Accessed 06-05-2024].

[2] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. 2017. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems* (2017), 1–1. https://doi.org/10.1109/TPDS.2017.2766064

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. https://doi.org/10.1002/cpe.1631

[4] Ray Authors. 2024. *Ray Compiled Graph Documentation.* Technical Report. AnyScale. https://docs.ray.io/en/latest/ray-core/ray-dag.html

[5] Lorena Barba and Gilbert Forsyth. 2019. CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education* 2, 16 (2019), 21. https://doi.org/10.21105/jose.00021

[6] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. 2022. Pathways: Asynchronous Distributed Dataflow for ML. arXiv:2203.12533 [cs.DC]

[7] Michael Bauer and Michael Garland. 2019. Legate NumPy: accelerated and distributed array computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 23, 23 pages. https://doi.org/10.1145/3295500.3356175

[8] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. 2021. Scaling implicit parallelism via dynamic control replication. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/3437801.3441587

[9] Michael Bauer, Elliott Slaughter, Sean Treichler, Wonchan Lee, Michael Garland, and Alex Aiken. 2023. Visibility Algorithms for Dynamic Dependence Analysis and Distributed Coherence. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 218–231. https://doi.org/10.1145/3572848.3577515

[10] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 66, 11 pages.

[11] Dan Bonachea and Paul H. Hargrove. 2018. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18) (Lecture Notes in Computer Science, Vol. 11882)*. Springer International Publishing. https://doi.org/10.25344/S4QP4W https://doi.org/10.25344/S4QP4W.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2011. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 1151–1158. https://doi.org/10.1109/IPDPS.2011.281

[13] Pi-Yueh Chuang. 2021. *TorchSWE: GPU shallow-water equation solver.*

[14] Modan K. Das and Ho-Kwok Dai. 2007. A survey of DNA motif finding algorithms. *BMC Bioinformatics* 8, 7 (01 Nov 2007), S21. https://doi.org/10.1186/1471-2105-8-S7-S21

[15] Dask. 2024. *Dask Computation Stages.* Technical Report. Dask. https://docs.dask.org/en/stable/phases-of-computation.html

[16] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling.* http://dask.pydata.org

[17] Mario Di Renzo, Lin Fu, and Javier Urzay. 2020. HTR solver: An open-source exascale-oriented task-based multi-GPU high-order code for hypersonic aerothermodynamics. *Computer Physics Communications* 255 (2020), 107262. https://doi.org/10.1016/j.cpc.2020.107262

[18] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 465–478.

https://doi.org/10.1145/1542476.1542528

[19] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.* 44, 6 (June 2009), 465–478. https://doi.org/10.1145/1543135.1542528

[20] Urs Hölzle and David Ungar. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.* 18, 4 (jul 1996), 355–400. https://doi.org/10.1145/233561.233562

[21] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. 2017. Dynamic task discovery in PaRSEC: a data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (Denver, Colorado) *(ScalA '17)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/3148226.3148233

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. *CoRR* abs/1807.05358 (2018). arXiv:1807.05358 http://arxiv.org/abs/1807.05358

[23] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. 2001. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Combinatorial Pattern Matching*, Amihood Amir (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 181–192.

[24] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. 2018. Dynamic tracing: memoization of task graphs for dynamic task-based runtimes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 34, 13 pages.

[25] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. 2017. Execution templates: caching control plane decisions for strong scaling of data analytics. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 513–526.

[26] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *CoRR* abs/1712.05889 (2017). arXiv:1712.05889 http://arxiv.org/abs/1712.05889

[27] NVIDIA. 2024. *CUDA Graph Documentation*. Technical Report. NVIDIA. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html

[28] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The java hotspotTM server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Monterey, California) *(JVM'01)*. USENIX Association, USA, 1.

[29] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed memory code generation for mixed Irregular/Regular computations. *SIGPLAN Not.* 50, 8 (jan 2015), 65–75. https://doi.org/10.1145/2858788.2688515

[30] Mahesh Ravishankar, John Eisenlohr, Louis-Noel Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2012. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.30

[31] E. Rotenberg, S. Bennett, and J.E. Smith. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. 24–34. https://doi.org/10.1109/MICRO.1996.566447

[32] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 76–85. https://doi.org/10.1145/872757.872770

[33] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. 2023. Loop Rerolling for Hardware Decompilation. *Proc. ACM Program. Lang.* 7, PLDI, Article 123 (jun 2023), 23 pages. https://doi.org/10.1145/3591237

[34] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Leek, Sean Treichlerk, Patrick McCormick, and Alex Aiken. 2020. Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00066

[35] James A. Storer and Thomas G. Szymanski. 1982. Data compression via textual substitution. *J. ACM* 29, 4 (oct 1982), 928–951. https://doi.org/10.1145/322344.322346

[36] Jens Stoye and Dan Gusfield. 2002. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science* 270, 1 (2002), 843–856. https://doi.org/10.1016/S0304-3975(01)00121-9

[37] Sean Treichler, Michael Bauer, Ankit Bhagatwala, Giulio Borghesi, Ramanan Sankaran, Hemanth Kolla, Patrick Mccormick, Elliott Slaughter, Wonchan Lee, Alex Aiken, and Jacqueline H. Chen. 2017. S3D-Legion: An Exascale Software for Direct Numerical Simulation of Turbulent Combustion with Complex Multicomponent Chemistry. (11 2017). https://doi.org/10.1201/b21930-12

[38] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 267–284. https://www.usenix.org/conference/osdi22/presentation/unger

[39] Welch. 1984. A Technique for High-Performance Data Compression. *Computer* 17, 6 (1984), 8–19. https://doi.org/10.1109/MC.1984.1659158

[40] Wikipedia. 2024. Ruler function — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Ruler%20function&oldid=1193825609. [Online; accessed 02-May-2024].

[41] Rohan Yadav, Wonchan Lee, Melih Elibol, Manolis Papadakis, Taylor Lee-Patti, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. 2023. Legate Sparse: Distributed Sparse Computing in Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages. https://doi.org/10.1145/3581784.3607033

[42] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 2.

[43] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. 2006. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 263–271. https://doi.org/10.1145/1133981.1134012

[44] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977),

337–343. https://doi.org/10.1109/TIT.1977.1055714

[45] J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934

A Artifact Appendix

A.1 Abstract

This artifact presents the computational artifact of Apophenia, a system that automatically traces Legion applications. This artifact is supported by an implementation of Apophenia within the Legion runtime system and a standalone implementation of the repeated sub-strings algorithm as described in the paper. We also provide artifacts for the subset of our benchmarks that are open-source.

We evaluated Apophenia on the Eos and Perlmutter supercomputers. Each node of Eos is an NVIDIA DGX H100, containing 8 H100 GPUs with 80 GB of memory and a 112 core Intel Xeon Platinum. Each node of Perlmutter contains 4 NVIDIA A100 GPUs with 40 GB of memory and a 64 core AMD EPYC 7763. Nodes of Eos are connected with an Infiniband interconnect, while Perlmutter uses a Slingshot interconnect. We compile Legion on Eos with the UCX networking module, and use the GASNet-EX networking module on Perlmutter.

A.2 Artifact check-list (meta-information)

- **Program:** A mixture of scientific and machine learning applications.
- **Compilation:** C++ and CUDA compiler.
- **Metrics:** Average throughput.
- **Publicly available?:** Some aspects are publicly available, others are closed source.
- **Archived (provide DOI)?:** Legion with Apophenia: https://doi.org/10.5281/zenodo.13900083.
  Repeated Substrings: https://doi.org/10.5281/zenodo.13900514.
  TorchSWE: https://doi.org/10.5281/zenodo.13900751.
  FlexFlow: https://doi.org/10.5281/zenodo.13900858.

A.3 Description

A.3.1 How to access. The version of Legion with Apophenia is available here. The standalone implementation of the repeated substrings algorithm is available here. The version of TorchSWE used for benchmarking is available here, though executing it with Apophenia requires a currently closed-source version of the Legate runtime and cuNumeric. The version of FlexFlow used for benchmarking is available here.

A.3.2 Hardware dependencies. Our experiments were run on server-class machines with multiple GPUs per node. While these are not necessary, the scaling and problem sizes that fit on each node will differ on different setups.

A.3.3 Software dependencies. We run all our experiments with Python 3.11. Aside from that, a standard supercomputer software stack (C++ compiler, CUDA installation, MPI installation) is expected.

A.4 Installation

Apophenia can be built and run with a standard Legion build, using the version of Legion from the artifact. The exact parameters to build Legion depend on the machine configuration. A sample Legion build for an Infiniband-based cluster with NVIDIA GPUs would invoke:

```
cd Legion/language
USE_CUDA=1 CONDUIT=ibv ./scripts/setup_env.sh
```

Since FlexFlow is fully open-source, we also provide build instructions. Using the provided version of Legion, FlexFlow can be built and installed for an NVIDIA machine with

```
export FF_GPU_BACKEND="cuda"
conda create -n flexflow
source activate flexflow
conda install -c conda-forge cmake make pillow \
  cmake-build-extension pybind11 numpy pandas \
  keras-preprocessing onnx transformers>=4.31.0 \
  sentencepiece einops
conda install -c pytorch pytorch torchvision torchaudio
conda install rust
pip3 install tensorflow notebook
cd FlexFlow
mkdir build
cd build
../config/config.linux
make -j
```

A.5 Experiment workflow

As a majority of our experiments are closed source, we do not provide a script that can run the full experiment suite. We do provide a command line that can be used to run the FlexFlow benchmark. The given command line is intended for SLURM based clusters, but additional configuration may be required depending on SLURM setup.

```
srun -N <NODES> \
 FlexFlow/build/examples/cpp/candle_uno/candle_uno \
  --warmup 30 \
  --batch-size 16384 \
  -ll:gpu <GPUS-PER-NODE> \
  -ll:fsize <GPU-MEM-IN-MBS> \
  -ll:util 4 \
  -ll:csize 30000 \
  -ll:zsize 5000 \
  -lg:enable_automatic_tracing \
  -lg:auto_trace:min_trace_length 25 \
  -lg:auto_trace:max_trace_length 200 \
  -lg:auto_trace:batchsize 5000 \
  -lg:auto_trace:identifier_algorithm \
      multi-scale \
  -lg:auto_trace:multi_scale_factor 500 \
  -lg:auto_trace:repeats_algorithm \
      quick_matching_of_substrings \
  -lg:inline_transitive_reduction \
  -lg:window 30000
```

A.6 Evaluation and expected results

We expect that when used on our benchmark applications, Apophenia finds and replays traces. On problem sizes that are Legion runtime-limited, this should result in speedup.

A.7 Experiment customization

Apophenia exposes several runtime configurations that are accepted by every Legion application for customizing the behavior.

1. `-lg:enable_automatic_tracing`: enable automatic tracing.
2. `-lg:auto_trace:min_trace_length <N>`: minimum length trace to consider.
3. `-lg:auto_trace:max_trace_length <N>`: maximum length trace to replay.
4. `-lg:auto_trace:batchsize <N>`: size of the task history buffer.
5. `-lg:auto_trace:multi_scale_factor <N>`: minimum size of the adaptive analysis.