Distill: Domain-Specific Compilation for Cognitive Models

Ján Veselý*§ Raghavendra Pradyumna Pothukuchi*§ Ketaki Joshi* Samyak Gupta†‡
Jonathan D. Cohen† Abhishek Bhattacharjee*
*Yale University, USA †Princeton University, USA

***Abstract*—Computational models of cognition enable a better understanding of the human brain and behavior, psychiatric and neurological illnesses, clinical interventions to treat illnesses, and also offer a path towards human-like artificial intelligence. Cognitive models are also, however, laborious to develop, requiring composition of many types of computational tasks, and suffer from poor performance as they are generally designed using high-level languages like Python. In this work, we present *Distill*, a domain-specific compilation tool to accelerate cognitive models while continuing to offer cognitive scientists the ability to develop their models in flexible high-level languages. *Distill* uses domain-specific knowledge to compile Python-based cognitive models into LLVM IR, carefully stripping away features like dynamic typing and memory management that add performance overheads without being necessary for the underlying computation of the models. The net effect is an average of 27× performance improvement in model execution over state-of-the-art techniques using Pyston and PyPy. Distill also repurposes classical compiler data flow analyses to reveal properties about data flow in cognitive models that are useful to cognitive scientists. Distill is publicly available, integrated in the PsyNeuLink cognitive modeling environment, and is already being used by researchers in the brain sciences.**

***Index Terms*—Domain-specific compilation, cognitive models, human brain, JIT compilers, Python.**

I. Introduction

Computational models that simulate the processes underlying human cognition advance our understanding of the human brain and mind. They describe how stimuli are acted upon by various neural or mental mechanisms to produce cognitive function. Insights from cognitive modeling have influenced not only the brain, psychological, and cognitive sciences, but also the field of artificial intelligence (AI), from the onset of artificial neural networks to recent advances in deep learning for gameplay and scientific computing [1], [2]. Cognitive models are expected to augment AI by offering brain-like intelligence not currently captured by deep learning (e.g., relational reasoning [3], planning [4], and more).

Cognitive models are computationally demanding. They are typically run hundreds of thousands of times to estimate the best model parameters to explain experimental data (e.g., human responses to psychological tasks), to assess the dynamics of cognitive processes over time steps, or to collect distributions of outcomes when the models include stochastic elements.

Cognitive scientists typically use Python to rapidly prototype cognitive models using optimized scientific computing libraries [5]–[9]. Unfortunately, models developed in Python run slowly. As more sophisticated cognitive models are built to capture advanced brain processes, Python's inefficiency worsens to the extent that some cognitive models can take several days or weeks to run, hindering scientific progress.

Dynamic compilation tools like PyPy [10] and Pyston [11] can accelerate cognitive models, but only partially. PyPy and Pyston cannot optimize complex dependencies in cognitive models because of the runtime checks needed for Python's dynamic data structures and dynamic typing. Large-scale cognitive models also require integration of sub-models developed across environments (e.g., PyTorch [8], Emergent [12], NEURON [9] or PsyNeuLink [7]); it is difficult to design compilers that optimize across computations expressed in several environments. All these aspects of cognitive models also obscure the natural parallelism available in cognitive models, and impede the ability to offload portions of the models onto hardware accelerators for which they are otherwise suitable.

New domain-specific languages for cognitive modeling would likely maximize performance, but require large-scale community buy-in and porting of many models already built across many research institutions using Python. Additionally, cognitive models are heterogeneous, integrating components with varying levels of biological fidelity, developed in different frameworks and research groups; e.g., a single model can include neurally accurate descriptions of some brain structures, an artificial neural network from machine learning to determine the attention allocated to inputs, and a behavioral model of control to modulate the pathways. Extreme heterogeneity impedes the design of a canonical set of language constructs and software tools needed for a domain-specific language.

In response, we build *Distill*, a dynamic compilation tool that exploits the domain knowledge of cognitive modeling to generate efficient code for the models. *Distill* aggressively eliminates Python's dynamic code, and generates LLVM IR for all the components in a model, including those developed in ancillary environments (e.g., Pytorch) and the frameworks used to run the models (e.g., PsyNeuLink). *Distill* is inspired by the observation that cognitive scientists, like scientists in other communities [13], [14], use Python because it is flexible, easy and provides access to optimized scientific computing libraries [5], [6], but do not need many of the dynamic language features that degrade performance. Eliding these dynamic features improves performance in itself, and also enables the use of existing optimizations in the LLVM framework to further

§ Joint first authors of the research highlighted in this paper.
‡Samyak Gupta began contributing to this work as an undergraduate student at Rutgers University via a summer research project in 2019 at Yale University.

978-1-6654-0584-3/22 © 2022 IEEE

boost performance. The choice of LLVM IR also permits leveraging LLVM's existing code generation backends for CPU architectures and accelerators, with no change.

Distill accelerates cognitive model execution by an average of 27× and maximum of 923× compared to Pyston and PyPy for a suite of well-known cognitive models. *Distill* enables one widely-studied cognitive model to execute in under five seconds, even though it originally failed to complete within twenty-four hours. *Distill* also extracts parallelism from the models and targets multi-core CPUs and GPUs, resulting in additional 4.9× and 6.4× speedups, respectively.

Lowering entire cognitive models, including sub-models from other environments and the execution framework, into LLVM IR offers an additional benefit—the ability to use compiler analysis to infer semantic properties about the model. *Distill* automates several types of model-level analysis that have traditionally been manually undertaken by scientists in labor-intensive and tedious ways. These analyses also permit *Distill* to discover user-guided optimizations specific to cognitive models.

We demonstrate two examples of *Distill*'s analyses and user-guided optimizations. First, *Distill* identifies cases where entire models can be verified to be equivalent, and for some models, recognizes when certain complex nodes are equivalent with—and hence, can be replaced by—simpler modules that have an analytical solution. Second, *Distill* calculates the impact of a cognitive model's parameters on its outputs and finds their optimal values entirely with compiler analysis built on LLVM's value range propagation and scalar evolution passes. Ordinarily, parameter estimation requires over hundreds to thousands of model runs, but *Distill* automates this step, saving days to weeks of modeling effort. Moreover, because of the general utility of our enhancements to LLVM's passes (i.e., extending support for integers to floating point), we have submitted a patch to the LLVM community for mainline integration.

Distill's design is guided by three main principles. First, we wish to avoid requiring cognitive scientists to change the source-code of their models or frameworks. Second, we delegate performance extraction to the compiler, allowing scientists to focus on creating models in the manner most intuitive to them. Third, we minimize software engineering effort, reusing LLVM IR and its associated infrastructure.

To summarize, our research contributions are:

1) *Distill*, a compilation tool that exploits domain-specific knowledge to provide near-native execution speeds for cognitive models, along with support to offload computations on accelerators. *Distill* does not require changes to source code and reuses existing LLVM infrastructure.
2) Discovery that user-guided analyses and optimization can be performed by compiler analysis, and incorporation of this idea into *Distill*.
3) Evaluation of *Distill*-accelerated models on single and multicore CPUs and GPU.

Distill is integrated with PsyNeuLink [7], a state-of-the-art cognitive modeling framework[1]. PsyNeuLink, along with the emerging Model Description Format (MDF) [15], enables the import and execution of sub-models developed across various modeling environments. *Distill* is being used in several leading cognitive science research labs and in the classroom internationally. Overall, *Distill* enables the design of larger and more complex cognitive models than previously possible. This is an important and necessary step towards the longstanding research goal of understanding and replicating human cognition.

II. Background and Motivation

A. Cognitive Models: Structure and Computation

Cognitive models are used to fit experimental data collected from humans performing psychological tasks, simulating cognitive processes, producing idealized outcomes, or for what-if analysis to understand the impact of tunable structures and parameters. Cognitive models are represented as graphs, where nodes are sub-processes or computational functions. Edges represent projections of signals between nodes. Nodes perform their computation when activation conditions are met (e.g., the arrival of an input, or, the passing of a specified time period).

Figure 1 illustrates the predator-prey task [16] that is used to study the role of cognitive control in allocating attention. An intelligent agent, either a human or non-human primate, is given a controller and shown a screen with three entities—a player that is positioned with the controller, a prey that the player must capture, and a predator that the player must avoid. The agent's attention is limited, and there is a cost for paying attention to an entity. Attention determines the accuracy with which the agent can perceive that entity's location. The agent does not have to distribute its attention fully.

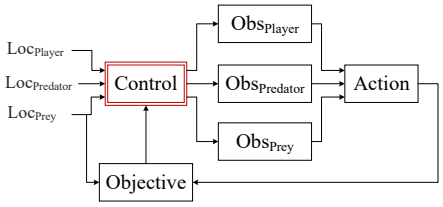

Fig. 1: A computational cognitive model of an intelligent agent performing the predator-prey task.

The role of attention in modulating perceived locations is modeled through the *Control*, *Objective* and *Obs* nodes. *Control* takes the exact 2-dimensional (2D) coordinates of all the entities (*Loc* values) per time step. The exact locations are obtained from an external environment like the gameplay interface that is used by the agent. The *Control* node allocates attention to each of the entities, determining the variances of three 2-D Gaussian distributions whose means are the actual locations of the entities. These distributions are sent to the *Obs* nodes that sample from them to generate the observed locations. Finally, *Action* calculates the player's movement for the time-step based on the observed positions of the three on-screen entities.

To identify the best allocation of attention for each entity, *Control* searches over all possible attention allocations, evaluating the cost of each allocation and the quality of the associated

[1]PsyNeuLink, integrated with *Distill*, is publicly available at https://github.com/PrincetonUniversity/PsyNeuLink/tree/master.

move. The cost of each allocation is calculated by *Control*, and the quality of the move is computed by *Objective*. *Objective* uses the direction given by *Action* and the true location of the prey to compute the goodness of the move. *Control* then selects the parameters that have the lowest cost. The entire process, from reading the input locations to searching over allocations, is repeated per time-step until the prey or the player is captured. Scientists are interested in both the final outcome of the task, and the temporal dynamics of decision-making.

Figure 1 shows the basic predator-prey model, but advanced variants can include the use of memory to recall previous locations, neural networks trained on experimental data to generate moves, or visual processors that extract locations from screen frames. These added components may be developed in PyTorch, Emergent, NEURON, and others.

B. The PsyNeuLink Framework

While cognitive models have historically been developed in many environments, recent efforts focus on a single "lingua franca" environment that can exchange models built across environments [15]. PsyNeuLink, the environment in which we prototype *Distill*, is prominent among emerging standardized environments that accept models specified in MDF, a common format to represent models from several environments including PyTorch and ONNX [17]. Despite its nascence, PsyNeuLink is already used in leading cognitive science research laboratories and classrooms worldwide (e.g., at Princeton University, Arizona State University, University of Leiden). Furthermore, the MDF project, which is contributed by an even larger body of researchers [15], greatly expands the reach of PsyNeuLink.

Figure 2 shows the predator-prey task modeled in PsyNeuLink. Model nodes are referred to as *Mechanisms* (`ProcessingMechanism` or `OptimizationControlMechanism`), and the model is a `Composition`. Each node contains a function which could be a PsyNeuLink-defined function (e.g., the `GaussianDistort` and `GridSearch` functions used in the `prey_obs` and `control` nodes, respectively) or could be user-defined (e.g., the `action_fn` in the `action_mech` node). Inputs are defined as a dictionary and the composition is run as many times as specified by `num_trials`.

The PsyNeuLink library contains functions common in the cognitive sciences. Users can also define their own functions. These functions perform numerical computations to model neural or mental processing, and contain a subset of Python. The current MDF specification [18], to which PsyNeuLink conforms, limits these functions to arithmetic, boolean, relational, and conditional operators, as well as lists of homogeneous types, tuples, arrays, Python built-in functions (e.g., sum, len, max, int and float conversion), numpy functions (e.g., tanh, exp, sqrt) and attributes (e.g., shape, flatten). MDF does not currently allow generators, list comprehensions, I/O operations, as well as `try` and `except` constructs in these functions. This is a common standard in this domain—e.g., other environments like NeuroML [19] for neuronal modeling, TorchScript for PyTorch model creation [20], or even generic high-performance dynamic

```
#Import psyneulink and numpy libraries
import psyneulink as pnl
import numpy as np
from psyneulink.core.components.functions…
from psyneulink.core.components.mechanisms…
…

#Loc nodes
prey_loc = ProcessingMechanism(size=2,name=…)
…

#Obs nodes with a PsyNeuLink library function
prey_obs =
ProcessingMechanism(size=2,function=GaussianDistort
                    ,name=…)
…

#User defined action function
def action_fn(positions):
    predator_pos = positions[0]
    …
    #floating point computations to obtain move
    #e.g., numpy.sqrt, numpy.exp,…
    …
    return move

#Action node with the user defined function
action_mech =
pnl.ProcessingMechanism(function=action_fn,
    input_ports=["obs_predator",…],name="Action"…)

#Compose nodes into a model
agent_comp = Composition(name="The Model")
agent_comp.add_node(action_mech)
…

#Add controller that uses the GridSearch function
control =
OptimizationControlMechanism(agent_rep=agent_comp,
                             function=GridSearch(…),…)
agent_comp.add_controller(control)
…

#Specify inputs
input_dict =
{player_pos:[[XX,XX]],predator_pos:[[YY,YY]],prey_p
os:[[ZZ,ZZ]]}

#Run
run_results = agent_comp.run(inputs=input_dict,
                     num_trials=1000,mode=…)
```

Fig. 2: Specifying the predator-prey model in PsyNeuLink.

compilation frameworks like Numba [21] use a similar subset of Python. This knowledge is not (but should be) used to aggressively optimize cognitive models.

C. Cognitive Model Execution

Figure 3 shows the high-level steps behind running a model in PsyNeuLink. When the model is run (in the format shown in the last line of Figure 2), PsyNeuLink first performs a sanitization check to ensure that the nodes are properly connected. It runs through all nodes, initializing all parameters and inputs with default values and propagating inter-node signals. The shapes of each node's inputs and outputs in the sanitization run must match those used in the actual run.

```
#Sanitize the model
sanitize()

#Parse inputs
parsed_inputs = parse_run_inputs(inputs)

#Run each trial
results = []
for trial_num in range(num_trials):
    #Get the input for the trial
    input = parsed_inputs[trial_num % len(parsed_inputs)]

    #Execute
    trial_output = execute_trial(inputs=input,
                                 scheduler=…)
    …
    #Handle outputs
    results.append(trial_output)

return results

#Execute one trial
def execute_trial(inputs=…):
    …
    while not termination_cond:
        #Find all nodes ready to run in this iteration
        #and run them
        ready_nodes = scheduler.run(…)
        for node in ready_nodes:
            node.execute(inputs=…)
    …
    #Collect results from the output ports of nodes
    #marked as outputs
    result = []

    for node in output_nodes:
        result.append(node.output_port.value)

    return result
```

Fig. 3: Running a model in PsyNeuLink.

Next, PsyNeuLink prepares a list of inputs from the input dictionary, such that each element of the list is an input to the model for one trial. The model is run until the completion of the trial (function `execute_trial` in Figure 3). A trial terminates when certain conditions are met; e.g., after a move for the player has been selected. During the trial, the scheduler identifies the nodes that are ready to run in each iteration based on the activation conditions that are explicitly specified per node. Examples of such conditions include waiting until other nodes are run a certain number of times, until the outputs of particular nodes stabilize, or after an amount of time has elapsed. Finally, the trial outputs are collected and returned.

D. Shortcomings of Dynamic Compilation Tools

Existing dynamic compilation tools like Pyston [11] and PyPy [10] miss many optimization opportunities for cognitive models because of several reasons. First, they cannot easily identify opportunities to reduce the runtime overheads required for tracking control flow. For example, the predator-prey model in Section II-A is run many times for a single input, but the path of execution is the same for all these runs. This is typical of cognitive models, but takes significant resources for PyPy and Pyston to track.

Second, existing dynamic compilation tools do not fully eliminate unnecessary dynamic Python features; e.g., inter-node signals have a fixed type, and also a fixed shape across runs. Thus, dynamic Python structures such as lists and dictionaries that are used to hold these values can be safely compiled to static data structures. However, changing a data structure requires updating all the accesses to that structure in the entire program, but existing tools usually focus only on individual functions and cannot undertake such aggressive optimizations.

Third, Pyston and PyPy cannot optimize across computations from different frameworks, and across scheduling invocations between executions of the model nodes. When a model uses computations from multiple environments like PyTorch and PsyNeuLink, even if the separate components are compiled, optimization does not cross these frameworks. Additionally, execution frequently switches between nodes and the scheduling logic that identifies which nodes are ready to run. Transitions back and forth between the model nodes and scheduler logic limit the scope of compiler optimizations, switching execution between compiled and interpreted modes.

Finally, available dynamic compilation tools cannot automatically extract parallelism from the models or offload computations to accelerators like GPUs. This is a wasted opportunity as there are several dimensions along which computations in cognitive models can be parallelized. For example, in the predator-prey model, the evaluations for each combination of attention allocations could have been run in parallel. When multiple samples are drawn from the distributions of observed locations, each sample and subsequent action could also be computed in parallel. While one might consider leveraging existing multithreading and GPU programming libraries for Python, they all require scientists to explicitly identify such parallel computations and mark functions to be offloaded to a GPU. A more desirable solution is to automate these steps so that cognitive scientists can solely focus on their designs rather than grapple with parallel programming constructs.

These shortcomings lead to cascading slowdowns. For example, unoptimized data structures not only have longer access times, but also prevent subsequent optimization passes by hindering the propagation of values and references. Multithreading with Python does not result in parallel execution because the threads are serialized by the Global Interpreter Lock [22], unless the threads run compiled code, in which case they do not have to take the lock. To maximally parallelize, it is important to compile the Python threads.

III. Distill: Domain-Specific Compilation for Cognitive Models

Cognitive models are computational graphs with complex scheduling rules. They are constructed in Python, and can fuse heterogeneous sub-models developed in multiple frameworks. Dynamic compilation with existing tools is not effective. *Distill* has domain-specific knowledge about the structure of model execution, and the expressions and data structures used in the models (Sections II-B and II-C). It uses this knowledge to optimize the models to near-native execution speeds.

While Python's dynamic features ease model construction, they are not actually necessary for model execution. *Distill*

aggressively eliminates dynamic features and converts dynamic data structures into statically-defined ones, yielding substantial acceleration. *Distill* also extracts parallelism and computations that are offloaded to GPUs. Lowering the entire model and the execution framework into LLVM enables compiler analyses to infer high-level semantic information about the models. Next, we describe *Distill*'s workflow.

A. Type and Shape Extraction

As described in Section II-A, repeated computations in the models have the same types and shapes for the values. Thus, the first step in compiling cognitive models is to deduce these types and shapes. Fortunately, this information is readily available from PsyNeuLink's sanitization run (Section II-C). By construction, the types and shapes of the inputs and outputs of each node during the sanitization run match those from subsequent post-sanitization runs. *Distill* uses this information to infer the types and shapes of all computations in the model.

B. Dynamic to Static Data Structure Conversion

Cognitive models use dynamic Python data structures like dictionaries and lists for node inputs, parameters, and outputs. Their shapes (and keys, for dictionaries) are execution-invariant. We convert these entities into statically-defined structures.

We create two structures that hold the values of the outputs of all nodes in the current and previous iterations (see the inner loop inside the `execute_trial` function in Figure 3). Node outputs are written to these top-level structures. We need two structures because multiple nodes running in the same iteration consume the values created in the previous iteration.

Next, we create separate structures for read-only and read-write parameters. Parameters exist at the node level (e.g., the attention levels in the *Control* node of the predator-prey model), or as arguments to functions (e.g., the amplitude argument of a function that computes a sinusoid). Such functions are defined in the nodes or in the framework's standard library. Creating separate structures eases parallelization, enabling threads to make local copies of the read-write parameter structure.

PsyNeuLink additionally contains two structures, one for the set of inputs for all trials (the variable `parsed_inputs` in Figure 3) and another for overall outputs in the trials. We convert these two entities into arrays.

Finally, the original computations often use strings as keys to fetch data. We convert strings to enumerated entries (enums), that are used as offsets to index into the structures.

Knowledge of the sizes and types of the model's parameters and outputs is available during sanitization, and for the inputs, this information is available whenever they are read. When *Distill* cannot infer the shapes of intermediate variables statically, it does not compile the models. We have not encountered this case in multiple years of working with cognitive models.

For models that select the parameter configuration with the minimal cost (e.g., attention allocation in the predator-prey model), multiple parameters may give the same minimal cost. In such cases, it is customary to randomly pick one parameter choice. To implement such constructs, we use reservoir sampling [23] so that we do not need to store a variable number of potential parameter choices, and then choose one from that list. With reservoir sampling, we can have a fixed size data structure that is statically defined.

Eliminating dynamic data structures significantly reduces their access times, and enables several optimizations. In their new format, it is easy to propagate values and references for subsequent optimizations. In addition, the data structures are now compact, improving cache performance.

C. Identifying Code Paths for Compilation

Dynamic compilation requires expensive tracking to determine which code paths to compile and when. In our case, we know that the key computations reside in per-node `execute` methods; we compile them because they are repeatedly invoked. This obviates the need to run expensive hot path analysis. We also compile metadata tracking functions, but not initialization and visualization code because they are not repeatedly invoked.

D. Generating LLVM IR

Distill is built on the llvmlite package [24], and generates LLVM IR for all model nodes and their functions.

1) Code Specialization: Standard library functions are expressed with pre-defined templates specialized to the types with which they are called. In the original execution, a function could have been invoked with different types of parameters due to Python's polymorphic semantics. *Distill* generates monomorphic code, creating a separate version of the function for each lexical instance it is invoked. All nodes have a generic template with the basic structure of a node (called a Mechanism in PsyNeuLink), that is filled with the node's computation.

2) Generating Code for Multiple Libraries and Frameworks: Recall that cognitive models can use computations from other libraries and frameworks. *Distill* takes these computations commonly used in cognitive models and generates LLVM IR. This includes simple functions from the NumPy library (e.g., the `logistic` function), and neural networks and optimizers from PyTorch. Lowering these different computations to a common IR allows optimization to span across them, resulting in more efficient code. The common IR also enables the application of compiler analysis for inferring high-level semantic information about the models, as we describe later.

Figure 4 shows the LLVM IR for the predator-prey model from Section II-B. *Distill* creates the literal struct types for read-write and read-only parameters, the outputs of all nodes, and model inputs and outputs. Then, it generates the code for the remaining execution. These structures are instantiated (not shown in the figure) before model execution; i.e., before the `@run_PREDATOR_PREY_COMPOSITION` function is called.

The code structure in Figure 4 aligns with the original PsyNeuLink model structure (Figure 2), and the PsyNeuLink execution framework (Figure 3), and is self-explanatory. *Distill* generates LLVM IR for the scheduling logic and uses a statically-sized boolean array for tracking the nodes ready to run in each iteration (in the `@exec_trial` function of Figure 4) instead of lists in the original code. *Distill* generates IR for

```
;Structure types
;Read-write parameters
"{ { { { { [1 x double], { [1 x double],…"

;Read-only parameters
"{ { { { { double, double, double, double,…"

;Outputs of all nodes
"{ { { [2 x double] },…"

;Model input (for one trial)
[3 x [1 x [2 x double]]]

;Model output (for one trial)
[1 x [2 x double]]

define void @run_PREDATOR_PREY_COMPOSITION (…) {
    …
    run_loop_body:
        ;get the input for each trial
        %.20 = load i32, i32* %.7, align 4
        %.21 = urem i32 %.14, %.20
        …
        ;call execution of one trial
        call void @exec_trial(…)
        store { [2 x double] } %.65, { [2 x double] }*
            %.64, align 8
        …
    …
}

define void @exec_trial(…) {
    …
    ;Create run status array
    %sched_init:
        %run_set = alloca [9 x i1], align 1
        …

    ;check termination condition
    %scheduling_loop_condition:
        …

    ;check run conditions for each node
    %sched_run:
        …
        %run_cond_ACTION = and i1 %.490, %.503
        …

    ;run nodes that are ready
    %invoke__ProcessingMechanism_ACTION:
        …
        call void @_ProcessingMechanism_ACTION(..)
    …
}

define void @_ProcessingMechanism_ACTION (…)    {
    …
    ;call the user defined function
    %invoke__UserDefinedFunction__187:
        …
        call void @_UserDefinedFunction__187(…)
    …
}

define void @_UserDefinedFunction__187(…) {
    …
    ;code generated for numpy operations
    %body:
        …
        %.133 = fadd double %.131, %.132
        …
    …
}
…
```

Fig. 4: LLVM snippets generated from compiling the predator-prey model.

all the Numpy operations in the user-defined `action_fn` that was part of the model. Figure 5 shows the user-defined action function of the predator-prey model, and Figure 6 shows the code generated by *Distill* for this function, after being optimized by LLVM's standard passes at the O2 level.

```
#User defined action function
#Uses social forces (attraction and repulsion) to
compute movement

def action_fn(positions):
    #Unpack positions. Each position is a pair of X
    #and Y coordinates
    predator_pos = positions[0]
    player_pos = positions[1]
    prey_pos = positions[2]

    #Get directions away from the predator and
    #towards the prey
    pred_to_player = player_pos - predator_pos
    player_to_prey = prey_pos - player_pos

    #Get distances between the player and the
    #predator and prey
    distance_to_predator =
     np.sqrt(pred_to_player[0] * pred_to_player[0] +
     pred_to_player[1] * pred_to_player[1])

    distance_to_prey = np.sqrt(player_to_prey[0] *
     player_to_prey[0] + player_to_prey[1] *
     player_to_prey[1])

    #Normalized directions
    pred_to_player_norm = pred_to_player /
     distance_to_predator
    player_to_prey_norm = player_to_prey /
     distance_to_prey

    #Weighted directions. Weights are designed so
    #that an entity closer to the player reduces the
    #player's response to the other entity.
    pred_to_player_wt = pred_to_player_norm *
     (distance_to_prey / (distance_to_predator +
     distance_to_prey))
    player_to_prey_wt = player_to_prey_norm *
     (distance_to_predator / (distance_to_predator +
     distance_to_prey))

    return pred_to_player_wt + player_to_prey_wt
```

Fig. 5: User-defined action function for the predator-prey model as expressed in PsyNeuLink.

Compiling the entire execution into a common IR enables aggressive optimization and permits compiler analysis for semantic information tracking. For example, replacing the calls to NumPy functions (e.g., `numpy.sqrt`) with LLVM instructions and intrinsics (e.g., `@llvm.sqrt.f64`) allows compiler analyses to track information across these calls. Additionally, the entire action function can be inlined where it is invoked, for further optimization and analyses.

E. Optimizations

After *Distill* generates IR, we run LLVM's standard optimization passes. These passes, like constant propagation and loop invariant code motion, work across multiple frameworks, and across computations from the model and its scheduler to create

optimized code. For example, they identify that when *Control* in the predator-prey model creates an attention allocation, *Obs* nodes can be run without explicit scheduler checks to identify ready nodes. Additionally, generating a common IR removes the invocation of the Python interpreter during the entire course of model execution, significantly improving performance. Finally, the compiled code does not hold the Global Interpreter Lock, enabling true parallel execution when multithreading is used.

F. Parallelism and GPU Acceleration

When a cognitive model contains a node that performs an exhaustive grid search (e.g., the *Control* node in the predator-prey model), each evaluation can be run in parallel. *Distill* can automatically extract multicore parallelism and offload computations to accelerators like GPUs.

For multi-threading, *Distill* spawns one Python thread per core. Each thread is assigned a segment of the grid search space, in which it evaluates parameters using functions compiled in the previous step. Each thread maintains a local copy of the read-write parameters and the node outputs that it writes to. Since threads only run compiled code, they do not need to synchronize with the Global Interpreter Lock.

For GPUs, we use the NVPTX backend included with LLVM [25] to generate NVPTX IR. This process generates a kernel for the evaluation function where each thread evaluates one point in the grid search space. The generated kernel is imported to CUDA by PyCuda [26] and executed on GPUs.

Models that sample from psuedo-random number generators (PRNGs)—e.g., sampling location distributions in the predator-prey model—use independent PRNGs for all evaluations for reproducibility. The state of the PRNG is a read-write parameter that is used and restored on every invocation of the evaluation function. This allows threads running in parallel to draw the same random numbers. As we discuss, replicating the state for each invocation has significant storage overheads.

G. Putting It All Together

Distill aggressively eliminates dynamic features used in cognitive models, and generates LLVM IR for the computations in them, even if they come from different environments. This allows model-wide optimizations, avoids invoking the interpreter, and naturally enables the extraction and exploitation of parallelism. Existing tools like PyPy and Pyston do not perform any of these optimizations as effectively.

IV. Augmenting Distill with Model Analysis

The organization of nodes in a model and the CDFG of the LLVM IR generated by *Distill* are closely connected. This inspired us to augment *Distill* with compiler analysis that provides model-level information to users. Analyzing a model's outcome when parameters or nodes are modified is an important aspect of cognitive modeling. For example, a researcher may want to assess the overall objective in the predator-prey model if a new node that amplifies the perceived threat from the predator is added to the model.

```
define void @_UserDefinedFunction__187({ double, double,
 i32, double, double, double }* noalias nocapture
 nonnull readnone %.1, { [1 x double], [1 x double], [1
 x [5 x i32]], [1 x double] }* noalias nocapture nonnull
 readnone %.2, [3 x [2 x double]]* noalias nocapture
 nonnull readonly %.3, [1 x [2 x double]]* noalias
 nocapture nonnull %.4) local_unnamed_addr #8 {
    entry:
        %.8.elt = getelementptr inbounds [3 x [2 x
          double]], [3 x [2 x double]]* %.3, i64 0, i64
          0, i64 0
        %.8.unpack = load double, double* %.8.elt, align
          8
        %.8.elt1 = getelementptr inbounds [3 x [2 x
          double]], [3 x [2 x double]]* %.3, i64 0, i64
          0, i64 1
        %.8.unpack2 = load double, double* %.8.elt1,
          align 8
        %.13.elt = getelementptr [3 x [2 x double]], [3 x
          [2 x double]]* %.3, i64 0, i64 1, i64 0
        %.13.unpack = load double, double* %.13.elt,
          align 8
        %.13.elt4 = getelementptr [3 x [2 x double]], [3
          x [2 x double]]* %.3, i64 0, i64 1, i64 1
        %.13.unpack5 = load double, double* %.13.elt4,
          align 8
        %.18.elt = getelementptr [3 x [2 x double]], [3 x
          [2 x double]]* %.3, i64 0, i64 2, i64 0
        %.18.unpack = load double, double* %.18.elt,
          align 8
        %.18.elt11 = getelementptr [3 x [2 x double]], [3
          x [2 x double]]* %.3, i64 0, i64 2, i64 1
        %.18.unpack12 = load double, double* %.18.elt11,
          align 8
        %.25 = fsub double %.13.unpack, %.8.unpack
        %.28 = fsub double %.13.unpack5, %.8.unpack2
        %.37 = fsub double %.18.unpack, %.13.unpack
        %.40 = fsub double %.18.unpack12, %.13.unpack5
        %.51 = fmul double %.25, %.25
        %.58 = fmul double %.28, %.28
        %.59 = fadd double %.51, %.58
        %.60 = tail call double @llvm.sqrt.f64(double
        %.59)
        %.69 = fmul double %.37, %.37
        %.76 = fmul double %.40, %.40
        %.77 = fadd double %.69, %.76
        %.78 = tail call double @llvm.sqrt.f64(double
         %.77)
        %.84 = fdiv double %.25, %.60
        %.86 = fdiv double %.28, %.60
        %.94 = fdiv double %.37, %.78
        %.96 = fdiv double %.40, %.78
        %.105 = fadd double %.60, %.78
        %.106 = fdiv double %.78, %.105
        %.108 = fmul double %.84, %.106
        %.110 = fmul double %.86, %.106
        %.120 = fdiv double %.60, %.105
        %.122 = fmul double %.94, %.120
        %.124 = fmul double %.96, %.120
        %.133 = fadd double %.108, %.122
        %.136 = fadd double %.110, %.124
        %.139.repack = getelementptr inbounds [1 x [2 x
          double]], [1 x [2 x double]]* %.4, i64 0, i64
          0, i64 0
        store double %.133, double* %.139.repack, align 8
        %.139.repack14 = getelementptr inbounds [1 x [2 x
          double]], [1 x [2 x double]]* %.4, i64 0, i64
          0, i64 1
        store double %.136, double* %.139.repack14, align
          8
        ret void
}
```

Fig. 6: Generated LLVM code by *Distill* (after O2 optimization) for the user-defined action function in Figure 5.

We perform several such analyses entirely in the compiler by modifying the analyses present in LLVM. Sometimes, it becomes unnecessary to run the models at all to obtain such information, saving time and effort. Model-level analyses also enable user-guided optimizations; we envision *Distill* presenting cognitive scientists with multiple code generation alternatives with slightly different numerical properties and performance.

A. Sensitivity to Parameter Values

Cognitive modelers often wish to identify the impact of model parameter values on the model outputs. Typically, this is done by running the model with all the choices of parameter values. However, in several cases, we can perform such analysis entirely in the compiler using value range propagation.

Value range propagation (VRP) is a dataflow analysis that determines ranges of variables based on control flow, type restrictions, and operations used. For example, VRP can determine that $exp(x)$ can only ever be a positive number or *NaN*, and a commonly used *Logistic* function can be shown to always output values in the range (0,1]. We extend LLVM's existing integer-only implementation of VRP to also support floating point types and common floating point operations.

Our extensions to LLVM's VRP mirror its existing algorithm and code, with three differences. First, we make floating point ranges inclusive at both ends to avoid issues with representing infinities. Second, the possibility of NaNs is tracked separately in addition to the ranges. Finally, range arithmetic conservatively rounds boundaries (lower boundary toward -Inf, and upper boundary towards +Inf).

Extending VRP to floating point ranges is useful beyond analyzing cognitive models. Many floating point operations need special handling in the presence of special values like negative zero, not-a-number, or infinities. While the compiler can be instructed to optimize these using fast-math optimization flags, these are currently set globally per compilation unit or per function, or tracked in a limited way. Floating point ranges can be used to determine the absence of such special values for each operation, and fast-math optimizations can be applied without breaking strict semantics. This is especially useful for GPU targets, which often have specialized fast instructions that do not fully adhere to IEEE floating point semantics.

B. Estimating Convergence Times

Researchers are often interested in the temporal dynamics of brain processes that accumulate evidence over time. For these models, they wish to discover the estimated time by which evidence accumulation leads to a decision as model parameters are varied. *Distill* performs such analyses using the scalar evolution pass with LLVM. Scalar evolution (SCEV) extends VRP to loops to track value ranges across loop iterations and calculating the number of loop iterations if possible. Similar to VRP, we extend LLVM's SCEV pass to support floating point types and to calculate the minimum number of loop iterations. When integer values are needed (e.g., for loop iteration count), the values are rounded. Variable ranges at a loop exit enable continuation of range analysis beyond loops.

C. Adaptive Mesh Refinement for Subspace Search

Many cognitive models search a parameter space; for these models, it is often useful to find regions of the space with noteworthy behavior. We use *Distill*'s VRP to progressively narrow an example cognitive model's subspace of interest, akin to adaptive mesh refinement. Crucially, we estimate the best parameters without running the model, saving time.

Consider, for example, the predator-prey model. This model uses grid search to find the best attention allocation. For simplicity, consider that we want to find the best attention allocation for the prey when a fixed attention is allocated to the predator and player. The conventional approach is to run the model for various levels of attention (among 100 possible levels) for the prey. Moreover, for each level, the model must be run several times to obtain the output distribution.

Figure 7 shows how VRP and binary search are used to find the optimal attention allocation for the prey. In this case, our VRP uses a 2σ range for the Gaussian distributions. The X-axis of the chart is the attention level allocated to the prey and the Y-axis is a cost metric to evaluate the allocation. The boxes show how the search space is progressively refined. For example, the first iteration of the analysis finds that the range of the metric's value is lower in the region between 2.4 and 4.6, than in the region from 0 to 2.4. Therefore, *Distill* performs another binary search to find the metric's value in the region from 2.4 to 4.6 and so on. Eventually it finds the optimal allocation, which is close to 4.6 after about 7 rounds. In contrast, achieving the same outcome without VRP requires several hundreds of evaluations, as also shown in Figure 7.

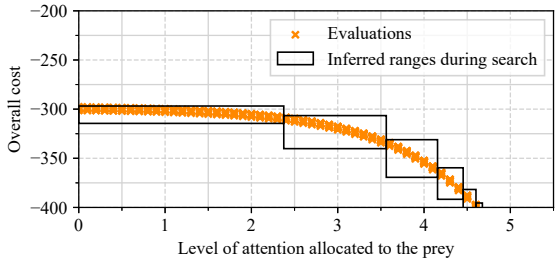

Fig. 7: VRP-based binary search for the minimum cost in the predator-prey model along with actual model evaluations.

D. Clone Detection for Model Similarity Analysis

Understanding whether a node or an entire model is equivalent to another is beneficial in several ways. In the simplest case, it helps verify correctness when a researcher creates an alternative version of the model, without changing the computations. This can happen when models are changed to be more intuitive without affecting their computational behavior. Model similarity analysis also enables substituting complex nodes with equivalent nodes that have simpler computation.

We use LLVM's existing `FunctionComparator` framework to detect equivalent functions. Consider the Drift Diffusion Model (DDM) and Leaky Competing Accumulator (LCA) functions [27] to simulate decision-making. DDM simulates two-choice decision-making and has an analytical solution,

while LCA simulates multi-choice decision-making. DDM (Equation 1), uses a and c as model parameters and $\mathcal{N}()$ as Gaussian noise. LCA (Equation 2) uses multiple inputs and outputs, with r, k as parameters, and $\mathcal{N}$ as generic noise.

$$Output(T) = Output(T-\Delta T) + a \times Input(T) \times \Delta T + \mathcal{N}_G(0, c\sqrt{\Delta T}) \quad (1)$$

$$Outputs(T) = Output(T-\Delta T) + (Inputs(T) - r \times Outputs(T-\Delta T)) \times \Delta T + \sqrt{\Delta T} \times \mathcal{N} + k \quad (2)$$

Invocations that use LCA with certain parameters (e.g., $r = 0$, $k = 0$ and $\mathcal{N} = \mathcal{N}_G(0,1)$, are equivalent to the DDM (e.g., with $a = 1$ and $c = 1$) for which analytical solutions to the dynamics are available. Clone detection can identify these cases, enabling the user to replace LCA with an analytical alternative, saving thousands of executions to identify the dynamics.

Aggressive inlining extends model similarity analysis beyond functions to entire models. *Distill* found that a model for the bistable perception of a Necker cube [28], and its hand-tuned vectorized version were equivalent, even though they differed in structure, node count, and computation of each node. This was possible because our clone detection analysis works at the IR level, independent of the original model's structure.

V. Experimental Setup

We evaluate *Distill* by using it to accelerate a selection of cognitive models designed in PsyNeuLink. We use PsyNeuLink's Python implementation as baseline and compare execution using Python-3.6.9 (*CPython*), as well as two JIT enhanced implementations, pyston-2.1.0 (*Pyston*), and pypy3-7.3.2 (*PyPy*). We also run PyPy without JIT compilation (*PyPy-noJIT*). Since *Distill* works with all Python implementations, we report *Distill* model execution in all four environments.

A. Cognitive Models Used in Our Evaluations

Necker Cube: This model simulates the perception of a subject when a bistable stimulus is shown, typically the line drawing of a cube which can appear to either project out or into the screen [28]. The model contains one node per vertex and evaluates when the subject's perception oscillates between the two orientations due to gradual changes in the nodes' values. We evaluate three variants of the model: *Necker Cube S*, which is the model for a 3-vertex drawing, *Necker Cube M*, which uses a cube with 8 vertices, and *Vectorized Necker Cube* which is a manually vectorized version of Necker cube M.

Predator Prey: We use 4 variants of the predator-prey model: S, M, L and XL that have 2, 4, 6 and 100 levels of attention per entity (prey, predator and player). *Predator Prey XL* is representative of the complexity of emerging models.

Botvinick Stroop: This model simulates the conflict when processing the name of a color, and the ink color with which the name is written [29]. It calculates decision energy, which changes over time due to the stimulus i.e., a colored word. This evolution is of interest to cognitive scientists.

We also consider two extended versions of this model, *Extended Stroop A* and *Extended Stroop B* that include the task of finger-pointing in addition to color naming. Two DDM nodes (one for color-naming and the other for finger-pointing) are added to the output of the Stroop model to produce a final decision. Versions A and B differ in how the inputs to the DDMs are computed and how its outputs determine the overall reward. While they are different conceptually, they are equivalent computationally, as detected by *Distill*.

Multitasking: This model simulates conflict among neural pathways responsible for representation in processing distinct stimuli and performing separate tasks. While the brain is subject to many types of representational conflict, this model includes a neural network (designed in PyTorch) that gives the color and shape of a stimulus image. This information, along with the choice of tasks, is passed to an LCA module (designed in PsyNeuLink) that accumulates evidence to generate a decision. A distribution of response times and histogram of correct/incorrect responses are collected. This is a heterogeneous model that spans PyTorch and PsyNeulink.

B. Evaluation Infrastructure

Our test machine has a 3.20 GHz Intel Core i7-8700 CPU with 6 cores and 12 threads, and 16 GB of 2666 MHz DDR4 DRAM. This machine also has a GeForce GTX 1060 with 3 GB GDDR5 and CUDA 11.1 that we use for the GPU experiments.

We collect all execution times using the *pytest-benchmark* package, and report the averages. We perform two warm-up runs before collecting data, and therefore, exclude compilation time, unless otherwise stated.

VI. Performance Evaluation

Figure 8 shows the running time of the models with the different Python implementations (i.e., *CPython*, *PyPy*, *PyPy-nojit* and *Pyston*) both without and with *Distill*. For the predator-prey model, we only use the smallest variant in this chart, and analyze the remaining variants separately. Execution times are normalized to those obtained from the corresponding *CPython* implementation, and are plotted on a logarithmic scale.

Consider the executions without *Distill* first. The execution times with *PyPy* and *PyPy-noJIT* are 67% and 71% *higher* than the standard *CPython* execution times, on average. The poor performance of *PyPy* is surprising in that it claims to improve performance and lower memory usage. *Pyston* has a 33% *lower* execution time than *CPython*, on average.

When *Distill* is used, execution times are 96%, 95%, 95% and 96% *faster* than the standard execution for the respective environments, on average. This translates to average speedups of up to 27×, and a maximum speedup of $\approx 923\times$ for the *Botvinick Stroop* model over *Pyston*, and a maximum speedup of $\approx 896\times$ for the *Multitasking* model over *CPython*. Existing JIT compilers fare poorly because they are designed for generic Python usage while *Distill* exploits domain-specific information for aggressive optimization, including eliminating the switching between PsyNeuLink and PyTorch for *Multitasking*.

It is also noteworthy that not all benchmarks run successfully with *PyPy* and *Pyston*. The *Botvinick Stroop* model and the XL variant of the *Predator Prey* model fail to complete with *PyPy* after exhausting all 16 GB of memory available on our

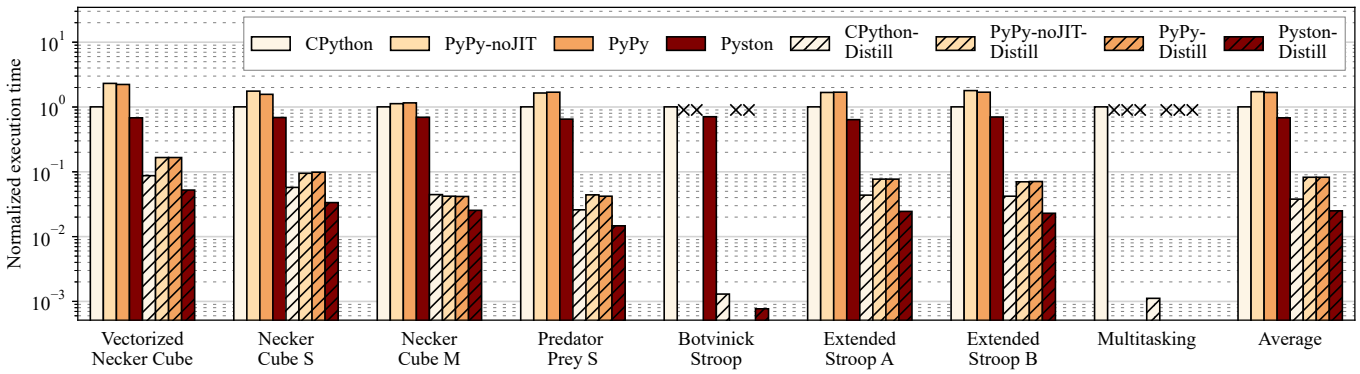

Fig. 8: Runtimes using different Python execution frameworks, with and without *Distill*, normalized to the respective execution times with CPython. Models that did not run successfully are shown with the × symbol. The *Botvinick Stroop* model ran out of memory with *PyPy*, and the *Multitasking* model uses PyTorch, which is not supported by *PyPy* and *Pyston*.

test system. Both *Pyston* and *PyPy* cannot run the *Multitasking* model because they do not support PyTorch.

A. Scaling Model Sizes

Figure 9 shows the normalized execution time of the four variants of the predator-prey model for *CPython* and *Distill*. The variants have 2, 4, 6, and 100 attention levels per entity, corresponding to 8, 64, 216 and 1,000,000 evaluations, respectively. Values are normalized to the execution time of the *S* variant with *CPython*, and are plotted on a log scale.

Figure 9 shows that the runtime for the S, M, and L variants with *Distill* remains nearly the same while the baseline *CPython* execution takes an order of magnitude longer time for each step up in the model size. The XL variant does not complete even after a full day with *CPython* alone, while *Distill* executes in 4.4 s. Moreover, despite the number of evaluations increasing by 4600× from L to XL, the running time with *Distill* only goes up by ≈330× from about 0.02 s to 4.4 s. As cognitive models become more complex to realistically capture human behavior, *Distill* is critical to ensure reasonable runtimes.

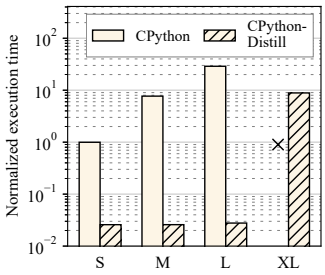

Fig. 9: Normalized execution times of the *Predator Prey* model as a function of increasing model sizes.

B. Model-Wide Optimizations

We create an environment (*CPython-Distill-node-only*) where *Distill* generates optimized code per node but the optimizations do not cross node boundaries and the PsyNeuLink scheduler logic is excluded from compilation. Figure 10 shows the execution time of *Botvinick Stroop* with this environment and the *CPython-Distill* design that compiles the entire model and scheduling logic, normalized to the *CPython* baseline. Compared to *CPython*, *Distill*-node-only compilation and default model-wide compilation result in 71% and 99.9% faster execution, or 3.4× and 774× speedups, respectively. Per-node compilation is not sufficient because the execution switches between scheduling logic in Python and the compiled nodes. This highlights the importance of model-wide compilation that spans the nodes and PsyNeuLink's scheduling logic.

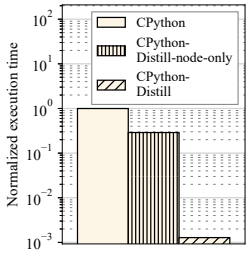

Fig. 10: Execution time of the *Botvinick Stroop* model.

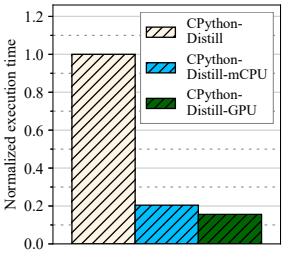

Fig. 11: Parallelized execution of the *Predator Prey XL* model.

C. Parallel and GPU Execution

Figure 11 shows the normalized execution times when the *Predator Prey XL* model (the largest of all the models evaluated) is run on a 12-threaded multicore CPU and a GPU. The values are normalized to the execution time of the *CPython-Distill* environment. Compared to single thread execution, multi-threaded CPU and GPU executions result in 4.9× and 6.4× performance improvements, corresponding to execution times of about 0.9 s and 0.7 s respectively. Recall that this model did not complete execution even after 24 hours with the standard *CPython* environment or with *PyPy* and *Pyston*. Importantly, *Distill* automatically generated parallel code for multi-threaded CPU and GPU execution without any user input.

Figure 11 also shows that the GPU execution is only about 28% faster than the multi-threaded CPU execution. This is because of memory access divergence. Our implementation replicates the state of the PRNGs used in each thread. Each thread uses four PRNGs to obtain the observed coordinates of the three entities, and for reservoir sampling, resulting in a total PRNG state of nearly 7.5 kB per thread. Such large storage stresses the GPU memory hierarchy, resulting in slower

execution. In fact, we find that ≈80% of the accesses to the L2 cache originate from the local memory, confirming that memory divergence is the main cause of less-than-ideal speedup.

While we could have used GPU-friendly PRNGs, we did not do so because the choice of a PRNG affects the values of the model output [30]. There is no theoretical analysis on how a new PRNG would impact the predator-prey model.

D. Cost of Compilation and Execution Time Analysis

The total execution time of a model includes the time to construct the input structures from Python and extract outputs to Python in addition to running the model's computations. These overheads are present whenever the model is run. There are also one-time overheads to construct the model parameter structures (a model with a given parameter set can be run many times with various inputs) and for the compilation process.

Figure 12 shows the cost of dynamic compilation and breakdown of the total execution times for two of our large models, the *Predator Prey XL* and *Multitasking* models. The values are normalized to the total execution time of the respective models with O0 optimization. Even though compilation times are significant, they are amortized because the models are typically run hundreds to thousands of times after compilation.

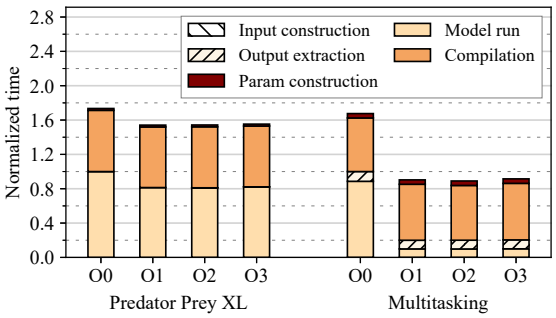

Fig. 12: Compilation and breakdown of total execution times.

Figure 12 also shows that standard LLVM optimizations are very effective in accelerating the *Multitasking* model. This model, unlike *Predator Prey* has a more complex structure to begin with. Therefore, the impact of compilation and subsequent optimization is even more pronounced.

VII. Related Work

JIT-based Python implementations like PyPy [10] and Pyston [11] are closest in spirit to *Distill*. But, unlike PyPy and Pyston, *Distill* does not need to support the Python language in its entirety, and instead focuses only on aspects required by cognitive models. Our focus on compiling only a subset of Python resembles Numba [21], [31]. However, Numba lacks the domain-specific information for optimizing cognitive models. It also requires code annotations, which we avoid.

We are not the first to notice the potential to leverage domain-specific knowledge to improve Python. Tuplex [14] uses knowledge of data types and hot paths to construct efficient data processing pipelines. This enables extraction and compilation of expected hot paths with exception checks that fall back to fully interpreted Python. Unlike Tuplex, *Distill* benefits from domain knowledge that a Python fallback is unnecessary for cognitive models. This enables *Distill* to compile models without any checks, and use the same IR for analysis.

Weld [13] also exploits the benefits of representing complete programs in a single IR for high performance. Unlike Weld, *Distill* relies on LLVM IR rather than designing a new IR. SONNC [32] targets neural networks, plugs into MATLAB, and performs static optimizations similar to *Distill*, but requires source code changes. There are many compiler frameworks for machine learning models (see [33] for a survey), which are different from cognitive models. Delite [34] targets embedded domain-specific languages for heterogeneous parallel devices.

Lastly, there have been several techniques proposed for advanced clone detection, which we believe will benefit the analysis of cognitive models [35]–[38].

VIII. Conclusions & Future Work

Cognitive models are vital to help explain the processes behind human cognition. *Distill* examines the role of compilers in supporting robust and high-performance modeling of cognitive processes. Beyond offering large performance improvements necessary to support complex cognitive models, we present the suitability of compiler analyses for cognitive modeling analyses. We propose and implement modifications to a production compiler suite to accelerate cognitive models, and provide rich feedback to cognitive modelers. All our contributions are part of open-source projects and released for public use.

Looking ahead, we plan to extend *Distill* to include advanced loop optimizations and parallelization using Polly [39]. We will also explore compiler optimizations that further accelerate cognitive models by embracing numerical deviation from non-compiled execution, but continue to preserve statistical features (e.g., stochastic optimizations, alternative PRNGs). Finally, we will explore analysis techniques that enable *Distill* to offer modelers statistical information of value (e.g., if a particular output belongs to a particular distribution).

Acknowledgements

We are grateful to Tobias Grosser for shepherding our paper. We also thank William Hallahan, Mark Santolucito, Anurag Khandelwal, Adrian Sampson, Malte Schwarzkopf, Caroline Trippel, Santosh Nagarakatte, Jay Lim, and Lin Zhong for their feedback on drafts of this paper. This work was supported in part by the National Science Foundation's (NSF) awards 1916817, 2040682, 1253700, 1337147, a Computing Innovation Fellowship (under NSF grant 2127309 to the Computing Research Association), the National Institute of Mental Health's award 5R21MH117548, as well as the Templeton World Charitable Foundation's award TWCF0305. The opinions expressed in this publication are those of the authors and do not necessarily reflect the views of these funding agencies.

Artifact Availability

The code, scripts and publicly available models used to generate this manuscript are archived on Zenodo [40]. An earlier version of this article appeared on arXiv [41].

REFERENCES

[1] D. Kumaran, D. Hassabis, and J. L. McClelland, "What learning systems do intelligent agents need? Complementary learning systems theory updated," *Trends in cognitive sciences*, vol. 20, no. 7, pp. 512–534, 2016. doi: https://doi.org/10.1016/j.tics.2016.05.004

[2] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, no. 7873, pp. 583–589, Aug 2021. doi: https://doi.org/10.1038/s41586-021-03819-2

[3] K. J. Holyoak, "Analogy and relational reasoning," in *The Oxford handbook of thinking and reasoning*, K. J. Holyoak and R. G. Morrison, Eds. Oxford University Press, 2012, pp. 234–259.

[4] M. K. Ho, D. Abel, J. D. Cohen, M. L. Littman, and T. L. Griffiths, "The efficiency of human cognition reflects planned information processing," in *AAAI Conference on Artificial Intelligence*, Apr. 2020. doi: https://doi.org/10.1609/aaai.v34i02.5485

[5] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020. doi: https://doi.org/10.1038/s41592-019-0686-2

[6] T. E. Oliphant, *Guide to NumPy*, 1st ed. Trelgol Publishing USA, 2006.

[7] "PsyNeuLink," accessed on 05/24/2020. [Online]. Available: https://princetonuniversity.github.io/PsyNeuLink/

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[9] M. L. Hines and N. T. Carnevale, "The NEURON simulation environment," *Neural Computation*, vol. 9, no. 6, pp. 1179–1209, 1997. doi: https://doi.org/10.1162/neco.1997.9.6.1179

[10] "PyPy," Dec 2019. [Online]. Available: https://www.pypy.org/

[11] "Pyston," 2020. [Online]. Available: https://github.com/pyston/pyston/releases/tag/v2.0

[12] B. Aisa, B. Mingus, and R. O'Reilly, "The Emergent neural modeling system," *Neural Networks*, vol. 21, no. 8, pp. 1146 – 1152, 2008. doi: https://doi.org/10.1016/j.neunet.2008.06.016

[13] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, S. Madden, and M. Zaharia, "Evaluating end-to-end optimization for data analytics applications in Weld," in *VLDB Endowment*, may 2018. doi: https://doi.org/10.14778/3213880.3213890

[14] L. Spiegelberg, R. Yesantharao, M. Schwarzkopf, and T. Kraska, "Tuplex: Data science in Python at native code speed," in *International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021. doi: https://doi.org/10.1145/3448016.3457244

[15] Model Exchange & Convergence Initiative, "ModECI model description format (MDF)," 2020. [Online]. Available: https://modeci.org/

[16] T. Willke, S. Yoo, M. Capotă, S. Musslick, B. Hayden, and J. Cohen, "A comparison of non-human primate and deep reinforcement learning agent performance in a virtual pursuit-avoidance task," in *Reinforcement Learning and Decision Making*, 2019. doi: https://doi.org/10.1101/567545

[17] "Open neural network exchange," 2019. [Online]. Available: https://onnx.ai/

[18] Model Exchange & Convergence Initiative, "Specification of ModECI," 2020. [Online]. Available: https://github.com/ModECI/MDF/blob/main/docs/README.md

[19] R. C. Cannon, P. Gleeson, S. Crook, G. Ganapathy, B. Marin, E. Piasini, and R. A. Silver, "LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2," *Frontiers in Neuroinformatics*, vol. 8, p. 79, 2014. doi: https://doi.org/10.3389/fninf.2014.00079

[20] "Torchscript language reference," 2019. [Online]. Available: https://pytorch.org/docs/stable/jit_language_reference.html

[21] "Numba: A high performance Python compiler." [Online]. Available: http://numba.pydata.org/

[22] D. Beazley, "Understanding the Python GIL," in *PyCON Python Conference. Atlanta, Georgia*, 2010.

[23] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, p. 37–57, Mar 1985. doi: https://doi.org/10.1145/3147.3165

[24] "A lightweight LLVM python binding for writing JIT compilers." [Online]. Available: https://github.com/numba/llvmlite

[25] J. Holewinski, "PTX back-end: GPU programming with LLVM," in *The Ohio State University. LLVM Developer's Meeting. November*, vol. 18, 2011.

[26] A. Klöckner, "PyCUDA: Even simpler GPU programming with Python," in *Nvidia GTC*, Sep. 2010.

[27] R. Bogacz, M. Usher, J. Zhang, and J. L. McClelland, "Extending a biologically inspired model of choice: multi-alternatives, nonlinearity and value-based multidimensional choice," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 362, no. 1485, pp. 1655–1670, 2007. doi: https://doi.org/10.1098/rstb.2007.2059

[28] L. A. Necker, "Observations on some remarkable optical phænomena seen in switzerland; and on an optical phænomenon which occurs on viewing a figure of a crystal or geometrical solid," *The London and Edinburgh Philosophical Magazine and Journal of Science*, vol. 1, no. 5, pp. 329–337, 1832. doi: https://doi.org/10.1080/14786443208647909

[29] M. M. Botvinick, T. S. Braver, D. M. Barch, C. S. Carter, and J. D. Cohen, "Conflict monitoring and cognitive control." *Psychological review*, vol. 108, no. 3, p. 624, 2001. doi: https://doi.org/10.1037/0033-295X.108.3.624

[30] T. H. Click, A. Liu, and G. A. Kaminski, "Quality of random number generators significantly affects results of Monte Carlo simulations for organic and biological systems," *Journal of Computational Chemistry*, vol. 32, no. 3, pp. 513–524, 2011. doi: https://doi.org/10.1002/jcc.21638

[31] "Numba: Supported python features." [Online]. Available: https://numba.pydata.org/numba-doc/dev/reference/pysupported.html

[32] L. C. McAfee and K. Olukotun, "Utilizing static analysis and code generation to accelerate neural networks," in *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress, 2012. [Online]. Available: http://icml.cc/2012/papers/903.pdf

[33] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2021. doi: https://doi.org/10.1109/TPDS.2020.3030548

[34] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, Apr 2014. doi: https://doi.org/10.1145/2584665

[35] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *International Conference on Software Engineering*, 2008. doi: https://doi.org/10.1145/1368088.1368132

[36] I. Keivanloo, C. K. Roy, and J. Rilling, "Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning," in *International Workshop on Software Clones (IWSC)*, 2012. doi: https://doi.org/10.1109/IWSC.2012.6227864

[37] P. Schugerl, J. Rilling, and P. Charland, "Reasoning about global clones: Scalable semantic clone detection," in *IEEE Annual Computer Software and Applications Conference*, 2011. doi: https://doi.org/10.1109/COMPSAC.2011.69

[38] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for simulink models," in *IEEE International Conference on Software Maintenance*, 2012. doi: https://doi.org/10.1109/ICSM.2012.6405285

[39] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012. doi: https://doi.org/10.1142/S0129626412500107

[40] J. Vesely, R. P. Pothukuchi, K. Joshi, S. Gupta, J. D. Cohen, and A. Bhattacharjee, "Distill: Domain-Specific Compilation for Cognitive Models Evaluation Artifact." Zenodo, Jan. 2022. doi: https://doi.org/10.5281/zenodo.5866935

[41] ——, "Distill: Domain-specific compilation for cognitive models," 2021, *arXiv 2110.15425*. [Online]. Available: https://arxiv.org/abs/2110.15425