Hardware Accelerated Semantic Declarative Memory Systems through CUDA and MapReduce

Mark Edmonds, Tanvir Atahary, Scott Douglass, and Tarek Taha

Abstract—Declarative memory enables cognitive agents to effectively store and retrieve factual memory in real-time. Increasing the capacity of a real-time agent's declarative memory increases an agent's ability to interact intelligently with its environment but requires a scalable retrieval system. This work represents an extension of the Accelerated Declarative Memory (ADM) system, referred to as Hardware Accelerated Declarative Memory (HADM), to execute retrievals on a GPU. HADM also presents improvements over ADM's CPU execution and considers critical behavior for indefinitely running declarative memories. The negative effects of a constant maximum associative strength are considered, and mitigating solutions are proposed. HADM utilizes a GPU to process the entire semantic network in parallel during retrievals, yielding significantly faster declarative retrievals. The resulting GPU-accelerated retrievals show an average speedup of approximately 70 times over the previous Service Oriented Architecture Declarative Memory (soaDM) implementation and an average speedup of approximately 5 times over ADM. HADM is the first GPU-accelerated declarative memory system in existence.

Index Terms—Declarative memory, ACT-R, semantic networks, parallel activation calculation

1 Introduction

A cognitive architecture is a hypothesis about the structural and behavioral mechanisms underlying cognitive activity. These architectures aim to enable agents to behave intelligently in complex environments by supporting the same high-level functionality as that of the human mind. Declarative memories offer a formalism for agent queries regarding domain knowledge given constraints and the agent's current context. For instance, a retrieval query might be for *toys* given that a *dog* is in the agent's environment. In this example, *toys* are the focus of the query while *dog* provides a bias towards particular toys (i.e. toys that are somehow related to dogs). In this work, the computational model underpinning the retrieval mechanism is designed to adhere to human performance. This work accelerates these retrievals using a Graphics Processing Unit (GPU).

The cognitive architecture community has been steadily increasing focus on providing cognitive agents with access to massive declarative memories [1], [2], [3], [4], [5], [6], [7], [8]. The need for larger declarative memories underscores the expansion of the applications of cognitive agents. Seeking to extend the practicality of existing declarative retrieval mechanisms, researchers are developing algorithms and computational frameworks that support massive stores of declarative knowledge and accelerate knowledge retrieval. Capacity and retrieval performance increases emerging from these efforts have the potential to dramatically change the modeling of human memory and the exploitation of declarative knowledge in agent-based software applications.

The Adaptive Control of Thought-Rational (ACT-R) is a theory of human cognition in the form of a cognitive architecture [9]. There are many other cognitive architectures; it is beyond the scope of this paper to review them (see [10] for a review). In this work, we focus on accelerating the declarative memory module of ACT-R. ACT-R's retrieval mechanism relies on each *chunk* (i.e. instance) of knowledge computing its *activation* (usefulness score) using a provided retrieval request and the current context. The retrieval request specifies top-down attributes the retrieved chunk must satisfy. The current context provides a bias (through a process called *spreading activation*) towards chunks connected to knowledge in the current context. The chunk of knowledge with the highest activation is returned as the result of the retrieval.

We call our retrieval system Hardware Accelerated Declarative Memory (HADM), and it faithfully reproduces the activation-based retrieval calculus found in ACT-R's declarative memory module. HADM represents a technical expansion of the Accelerated Declarative

- *M. Edmonds, T. Atahary, and T. Taha are with the Department of Electrical and Computer Engineering, University of Dayton, Dayton, OH 45469. E-mail: {edmondsm1, ataharyt1, ttaha1}@udayton.edu.*
- *S. Douglass is with the Air Force Research Lab, United States Air Force, Wright-Patterson AFB, Dayton, OH 45433. E-mail: scott.douglass.1@us.af.mil.*

Manuscript received 10 Mar. 2017; revised 12 June 2018; accepted 25 June 2018. Date of publication 23 Aug. 2018; date of current version 13 Feb. 2019. (Corresponding author: Mark Edmonds.)
Recommended for acceptance by P. Sadayappan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2866848

Memory (ADM) system presented in [1]. HADM is part of an effort to increase the scale and performance of a larger hybrid Cognitively Enhanced Complex Event Processing (CECEP) architecture that is both cognitive and high-performance.

The Large Scale Cognitive Modeling (LSCM) initiative set out to develop a specification and execution framework with which Air Force Research Lab (AFRL) cognitive scientists could develop and deliver intelligent agents capable of meeting information representation and processing scale requirements of AFRL systems and missions. Several current AFRL research and development efforts are investigating ways to accelerate core components of the CECEP architecture using multi-core and GPU architectures [11]. At the start of the LSCM initiative, assessments of associative declarative memories available in cognitive architectures such as ACT-R concluded that algorithms realizing critical aspects of their associative retrieval processes were inherently serial. The HADM system described in this paper specifically increases the scalability of associative retrieval by realizing it with a parallelized algorithm executing on massively parallel hardware.

HADM expands the practicality of ACT-R's retrieval mechanism through the acceleration of activation calculation using the parallel computing platform CUDA. CUDA increases the parallelization of activation calculation by leveraging the many processing cores available on a GPU. The research effort developing HADM is producing a comprehensive high-performance declarative retrieval system, not just faster activation calculation. The HADM implementation is functionally equivalent to ACT-R's retrieval process and offers substantial performance gains. HADM's primary contribution is to further parallelize activation calculation and knowledge retrieval by utilizing massively parallel hardware in the form of a GPU. HADM is the first GPU-based declarative memory retrieval system.

The rest of this paper is organized as follows: Section 2 outlines ACT-R's retrieval mechanism, Section 3 discusses related work, Section 4 explains the origins and motivation of HADM, Section 5 describes improvements HADM makes over its predecessor, Section 6 covers the General Purpose Computing on Graphics Processing Units (GPGPU) implementation of declarative retrieval, Section 7 examines issues of a large, long-running declarative memory (DM), Section 8 describes the experimental setup, and Sections 9 and 10 are the experimental results and conclusion.

2 Retrieval in ACT-R

2.1 ACT-R Overview

ACT-R is a cognitive architecture that can be used to specify and execute computational process models of human cognition [9]. The architecture consists of a central production system and several modules. The central production system can be thought of as the director of cognition. The modules support knowledge processing in the central production system by performing module-specific processes and actions. Modules exist for vision, goal maintenance, situation representation, audition, motor control, and declarative memory [9].

TABLE 1
ACT-R's Activation Calculation Equations

Name	Equation
Activation	$A_i = B_i + \sum_j W_j S_{ji} + \epsilon$
Base-level learning (base-level activation)	$B_i = \ln\left(\sum_{j=1}^{n} t_j^{-d}\right)$
Associative strength	$S_{ji} = S_{\max} - \ln(\mathrm{fan}_j)$

ACT-R's declarative memory is grounded in knowledge representations known as chunks. Chunks are composed of key-value pairs that encapsulate a unique piece of explicit factual knowledge. Upon successful retrieval, the chunk with the highest computed activation is placed in a retrieval buffer and can influence the behavior of ACT-R's central production system.

2.2 ACT-R's Activation-Based Retrieval Calculus

The full ACT-R retrieval process is influenced by: (1) top-down constraints defined in retrieval requests; and (2) contextual priming effects caused by chunks present in buffers capable of spreading activation. To specify a retrieval request in an ACT-R production, a modeler typically specifies the type of chunk on which to focus the retrieval process and may specify additional top-down constraints that must be met by any successfully retrieved chunk. During the retrieval process, all chunks of the specified type, or derived from the specified type through chunk-type inheritance, are considered initial retrieval candidates. Chunks meeting all top-down constraints defined in retrieval requests are considered final retrieval candidates. The activations of each chunk in the final candidate set are computed, and the chunk with the highest activation is retrieved.

The equations governing the retrieval process in ACT-R are listed in Table 1. Chunk activation is primarily based on a base-level reflecting the prior usefulness of a chunk and spreading activation reflecting the degree to which other chunks in context are associated with a chunk. Base-level learning influences retrieval by increasing the activation of recently and/or frequently retrieved chunks [9]. Spreading activation influences retrieval by allowing contextual knowledge to prime chunks through shared knowledge structure and association [12].

The activation equation in Table 1 mathematically describes how chunks present in context buffers (indexed by j) produce activation values through strengths of association (S_{ji}) and activation weights (W_j) that are combined with base-level activations to determine context-specific activations. Fan (fan_j) is a numerical representation of the complexity of a piece of knowledge by maintaining a count of the number chunks connected (i.e. have relations) to this chunk (chunk j). This spreading activation-based retrieval process becomes a computational burden in real-time systems that produce large retrieval candidate sets. To extend the effectiveness of ACT-R's retrieval mathematics to larger declarative memories, the retrieval process of a large DM must be examined, and the activation calculation must be accelerated to maintain real-time performance.

3 Related Work

ACT-R's declarative memory module follows a lineage of factual memory storage research [9], [12], [13], [14], [15], [16]. In the past decade, there has been a substantial push to increase DM efficiency, largely focused on capacity and speed [2], [3], [4], [5], [6], [7], [17], [18], [19], [20].

Persistent-DM [3] utilized a relational database as declarative memory's storage backend. Persistent-DM showed a non-linearly increasing retrieval time as the number of chunks increased. Salvucci [4] used a database system to enable vast quantities of declarative knowledge using Wikipedia. This database-based retrieval system handles larger declarative memories than those analyzed in this work but offers no analysis on the latency of retrievals. Chen et al. [18] used a database to efficiently spread activation; however, it is unclear if the system enforces top-down constraints.

RML1 [2] expanded on the work of Douglass et al. [3]. RML1 took advantage of Erlang's inherent concurrency and vastly outperformed ACT-R's default retrieval mechanism. Derbinsky et al. [5] outperform Douglass et al. [3] through efficient *cue components*. Cue components constrain declarative retrievals to two cues: (1) a positive cue indicating the retrieved chunk *must* satisfy the specified relation and (2) a negative cue indicating the retrieved chunk *must not* satisfy the specified relation. Jones et al. [17] optimize spreading activation through lazy evaluation. Their approach maintains Soar and ACT-R spreading activation behavior while making performance gains. In our work, spreading activation is a low-cost operation in comparison to enforcing top-down constraints.

Grinberg et al. [21] showed a powerful approximate computation of spreading activation. Their work exploits the sparsity of each node's connectivity matrix. Paths along non-zero elements are constructed, and the number of times a node is reached via different paths is used as a measure of activation. This work utilized networks with 100 million nodes, however, while spreading activation is a critical part of the ACT-R retrieval process, in practice, candidate determination requires significantly more computational resources [1].

There has also been an effort to endow cognitive architectures with episodic memory, where the agent remembers not only facts but also experiences [22], [23], [24], [25]. While we believe this line of research is interesting and worthwhile, our work focuses on semantic knowledge retrieval instead of episodic.

Frost et al. [26] introduces Street Engine, a parallel computer architecture tailored to cognitive computing. Street Engine's production language is based on Soar [15], [25]. This architecture translates production rules directly into parallel hardware chips called *productors*. While the design and simulation of this architecture is promising, the architecture is currently unavailable in hardware. The initiative of this work seeks to realize accelerated retrievals in physical hardware.

HOMinE [20] integrates computational ontologies with ACT-R in an attempt to create a more scalable knowledge base by encoding chunk types to represent alternative configurations of a chunk. This enables cognitive modelers to utilize heterogeneous ontology sources without modifying ACT-R's retrieval mechanisms. While this work is related to create extremely large declarative memories, our work focus on accelerating the retrieval process.

Kelly et al. [19] expand the expressiveness of chunks using holographic vectors. These vectors enable similarity metrics, fault tolerance, and lossy compression. Holographic vectors expand the dimensionality of factual knowledge but also fundamentally change the retrieval calculus of ACT-R.

Rosenbloom [27] treats a cognitive cycle as a solution to a factor graph, where the present evidence is used to pass messages through the graph until convergence. Rosenbloom [7] optimizes the messages cycles per graph cycles (MC/GC). Their method caches messages from previous retrievals to reduce MC/GC, and they examine sending messages in parallel. This work is an attempt to lower the period of a cognitive cycle below 50ms, though only performs simulated parallelism and experiments.

ACT-R's declarative memory module is distinct from Database Management System (DBMS) because it uses knowledge activation to integrate: (1) context priming; (2) retrieval recency; (3) retrieval frequency; and (4) the structure of declarative knowledge into a memory system that matches the information structure of its environment. Through the increase and decay of activation across a declarative memory, knowledge that is frequently retrieved and used to shape action in a given context is more reliably and rapidly retrieved in future similar contexts. This retrieval process is complex and non-trivial to implement in a DBMS, though there have been attempts to do this for both ACT-R and Soar [3], [4], [5], [18].

When a declarative memory is used to match an intelligent agents actions to the information structure of an operational environment, retrieval likelihoods that optimize the behavior of the agent must be computed as quickly as possible. Under these changed circumstances, it is unacceptable for an agent with a computationally slow retrieval process to impede the mission to which it is contributing: wall-clock time must never lag behind mission execution time. Since the hardware-accelerated declarative memory described in this paper is intended for use in AFRL applications requiring retrieval from massive knowledge repositories, the primary objective of this work is to calculate activation in as little wall-clock time as possible and outperform previous ACT-R implementations of declarative memory.

4 HADM Background

HADM and its predecessor, ADM, represent a technical extension of the RML1 retrieval system presented in [2]. To take advantage of the large-scale declarative memory system in RML1, cognitive modelers author and execute models in a framework developed using the Erlang programming language [11]. To broaden the usefulness of the RML1 declarative memory system, RML1 was functionally isolated from the broader Erlang execution framework; it was re-implemented as a net-centric software service that can be used in generic service-oriented architectures. Developed using Erlang, this Service Oriented Architecture Declarative Memory (soaDM) provides a declarative memory system that can be controlled and accessed through a published network interface.

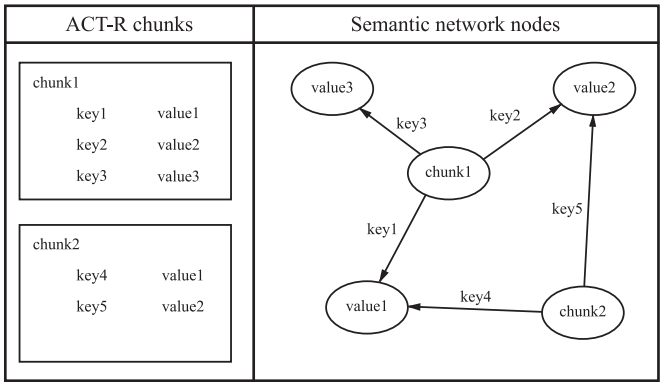

Fig. 1. Comparison of ACT-R chunk representation and soaDM/ADM semantic network representations.

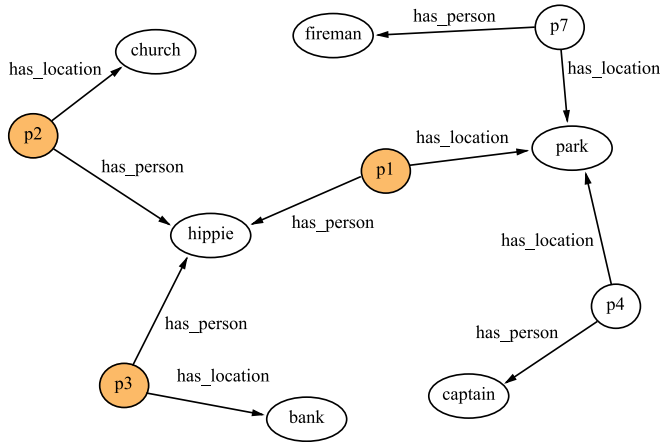

Fig. 2. Semantic network representing propositions {p1, p2, p3} and their corresponding relations in the semantic network.

4.1 Declarative Knowledge in Semantic Networks

Both soaDM and HADM represent declarative knowledge as a semantic network. *Nodes* in the semantic networks represent classes and instances. *Edges* in the networks represent "object properties" that capture relationships: (1) between classes; and (2) between classes and instances. The semantic network representation of declarative memory enables subsections of knowledge to be traversed, rather than simply iterating over the entire knowledge set.

Fig. 1 illustrates the mapping between frame-based chunks in ACT-R and semantic networks in soaDM and HADM. Chunks are realized as nodes, and key/value pairs are realized as directed edges. Nodes internally represent lists of data properties that capture relationships between the node and instances of data types (integer, float, etc.). Every relation in the network consists of a source node, a relation, and a destination node. For instance in Fig. 2, the proposition *p1* would have two relations, one of which would have a source node of *p1*, a relation of *has_person*, and a destination node of *hippie*.

The number of relations (connections) a node possesses represents the complexity of the knowledge the node represents. Each node maintains a *fan*, which represents this complexity through a count of relations. In semantic networks, a node's fan is the number edges directed at this node. In Fig. 1, *value1* has a fan of 2 while *value3* has a fan of 1.

One of the critical differences between the chunk and semantic network representation is that values used by chunks are expanded into full nodes. They have the same status in the network as the chunk nodes because they are literally other chunk nodes. This has powerful implications when attempting to access the network based on a value in a retrieval request.

4.2 Retrieval from Semantic Networks

A retrieval process yielding identical results as ACT-R can be realized in semantic networks using: (1) *activation sources*; and (2) *node property filters*. Activation sources specify: (1) *retrieval requirements*; and (2) *context primes*.

Retrieval requirements and node property filters are specified as tuples capturing: (1) a relation; and (2) a destination node of the relation. Together, retrieval requirements and node property filters are equivalent to ACT-R *top-down constraints*. The practical distinction between retrieval requirements and node property filters is when they are applied during the retrieval process. Retrieval requirements are applied in a top-down fashion while node property filters are used as bottom-up filters after finding a set of potential candidates. It is worth noting that soaDM, ADM, and HADM support ACT-R's symbolic retrieval requirements but do not support ACT-R's numerical comparators, partial matching, or blending. These features will be addressed in future work.

Context priming sources are specified as tuples capturing: (1) a relation; (2) a destination node of the relation; (3) a number corresponding to the total activation that that can spread from a context source; and (4) a number corresponding to the structural complexity of a source context. During retrieval: (1) activation is spread from activation sources; (2) node property filters are applied to nodes receiving activation; (3) nodes that received activation and survive property filters compute their activation; and (4) the node with the highest activation is determined. The retrieval process ultimately returns the set of relations (including the domain and range nodes) originating at the winning node.

Assuming identical chunk knowledge is available in ACT-R, Table 2 shows ACT-R retrieval requests and soaDM/ADM/HADM retrieval requirements and node property filters that are functionally equivalent. The first ACT-R retrieval request requires that any retrieved chunk be of *type proposition*. The first soaDM/ADM/HADM retrieval requirement spreads activation to all nodes related

TABLE 2
Retrieval Comparison Excluding Context Priming

ACT-R retrieval	Retrieval requirements	Node property filters
`+retrieval> isa proposition`	`type proposition`	
`+retrieval> isa proposition has_person hippie`	`type proposition`	`has_person hippie`
`+retrieval> isa proposition has_person hippie`	`has_person hippie`	`type proposition`

TABLE 3
Retrieval Comparison Including Context Priming

ACT-R retrieval	Retrieval requirements	Context priming sources	W_j
`+retrieval> isa proposition`	`type proposition`	`has_person hippie`	1/3

to *proposition* through the *type* relation. Identical activation calculation across all proposition chunks/nodes occurs in all retrieval systems.

The second and third retrieval requests impose an additional restriction on the retrieval process; retrieved chunks must also possess a key/value pair "*has_person hippie*" or candidate proposition nodes must be related to the hippie node via a has_person relation (semantic network). Note that swapping retrieval requirements and node filters in the third comparison spread activation from the *hippie* node and then requires that any candidate nodes be of type proposition. This swap illustrates that: (1) retrieval in a semantic network is not dependent on a *type* or *ISA* property; and (2) that different activation sources can be used to affect the same retrieval. This latter point is important; spreading activation from a low-fan node such as hippie can dramatically alter the complexity of retrieval. If the fan of *proposition* is 1 million, then activation is spread to potentially 1 million candidate nodes when *proposition* is used as a retrieval requirement activation source. Alternatively, if the fan of *hippie* is 25, only 25 candidate nodes are considered to receive spreading activation.

In soaDM, the modeler explicitly requests this swap between retrieval requirements and node property filters. In ADM and HADM, the optimal swap is found by querying the network for the node with the lowest fan. ADM and HADM then apply the other retrieval requirements as node property filters surrounding the node with the lowest fan. The process for finding the lowest fan is described in [1].

Table 3 shows how context priming sources are used in a semantic network to reproduce context-based spreading activation. Assuming the chunk *fireman* is available in one of ACT-R's activation source buffers, the retrieval processes in all three systems will yield identical results.

The semantic network representation allows for more optimized searching and traversal if activation calculation is effectively managed across candidate nodes. The critical challenge of achieving high-performance with the semantic network approach to retrieval is realizing activation calculation across all candidate nodes as quickly as possible. Both soaDM, ADM and HADM use the MapReduce computing model described in [28] to maximize the concurrency of candidate node activation calculation. The soaDM declarative system utilizes lightweight threads and Erlang's message passing to coordinate the execution of the retrieval process using MapReduce [9]. The ADM and HADM declarative systems use hash tables to coordinate retrievals. ADM executes on a CPU while HADM leverages the parallel resources of a GPGPU. All three retrievals systems execute an equivalent computing model that is equivalent to an ACT-R retrieval.

TABLE 4
HADM Host Hash Table Descriptions

Name	Description
Master relation list (MRL)	Enables lookup into subsections of network based on a key/value pair, where the name of the relation is the first lookup, and the node with the relation is the second lookup. Maintains unique relation IDs and effective fan values (see Section 5).
Name-to-Node map (NtN)	Maps node names to node references in the semantic network. Enables direct access to any node in the semantic network.

5 Optimizing Host Retrievals

HADM is an expansion of ADM to utilize a GPU for retrievals. HADM operates in two high-level modes: on the host (CPU, similar to ADM) and the device (GPU). Despite the focus of HADM being device retrievals, many improvements were made to the host implementation. This section outlines the intention and purpose of those improvements. The discussion focuses on HADM's use of a thread pool to manage and distribute parallel tasks.

5.1 Overview of Retrieval in ADM

In this subsection, we briefly outline the retrieval process implemented in ADM [1]. HADM utilizes a similar paradigm as ADM for host retrieval, but utilizes a thread pool to more efficiently saturate CPU resources. ADM utilizes two primary sources of retrieval speedups: (1) hash tables to allow constant-time lookup into any sub-section of the declarative network, and (2) an optimized candidate set determination that minimizes the number of nodes checked for top-down constraint satisfaction.

The description of ADM's two primary hash tables are listed in Table 4. These hash tables eliminate iteration and search present in prior retrieval systems through constant-time lookups. The Name-to-Node Map (NtN) enables direct access to every context-priming source; spreading activation is linear time in the number of nodes connected to the context-priming source.

ADM's more substantial contribution comes from optimizing candidate set determination. This optimization relies on the final candidate set being the intersection of all nodes fulfilling the top-down retrieval constraints. ADM utilizes the MRL to lookup the relations and nodes involved in each top-down retrieval constraint and selects the node with the smallest fan (see [1] for details). From this node, ADM applies all other top-down constraints as node filters.

5.2 Parallel Task Execution using a Thread Pool

Thread pools provide a number of worker threads, which complete tasks submitted to the thread pool. Because the worker threads live for the duration of the pool, thread pools have two critical properties: (1) they reduce the overhead of launching parallel work units since the threads are already instantiated and (2) allow the thread submitting work to the pool to determine the size of each task, while

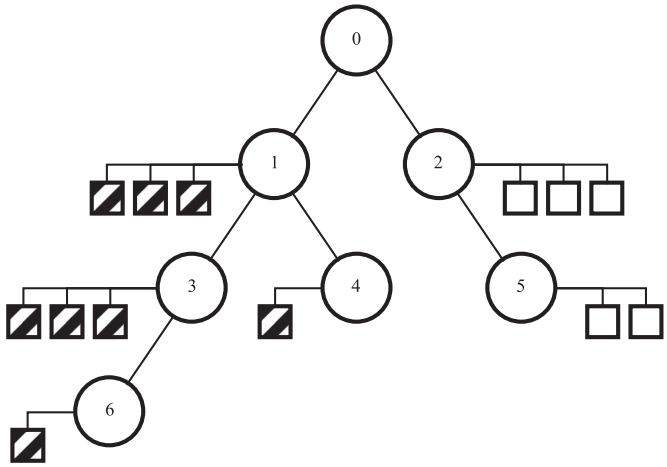

Fig. 3. Example network with multiple levels of candidates.

keeping the number of worker threads proportional to the hardware concurrency available.

When compared to launching a thread for each task, a threadpool's second property can drastically reduce the burden on the scheduler by mitigating CPU contention among threads, enabling more efficient parallelization for declarative retrievals. For HADM, these two properties allow the main HADM thread to simply submit tasks to the thread pool instead of launch for each parallelizable task. The HADM main thread also does not have to be concerned with how many worker threads are running, or if the worker threads are being properly saturated.

HADM's thread pool utilizes a work-stealing algorithm, based on [29], with bounded lock-free task queues. Because each queue has a fixed number of tasks, the size of each task is a scalable value based on the number of CPU cores. This design strikes a balance between having every worker thread finish all submitted tasks at once and allowing the main HADM thread to iteratively submit tasks to the thread pool based on network topology.

The thread pool uses a future for each task to access values in HADM main thread. The use of futures not only allows the HADM main thread to receive the return value of the task function, but it also adds a simple synchronization signal that the thread pool has completed all submitted tasks. The HADM main thread simply waits for all futures to make their return value available, which only occurs after the thread pool has emptied all task queues.

5.2.1 Candidate Determination using a Thread Pool

ADM's initial parallel execution, outlined in [1], made a top-level estimation about how to split CPU resources among a node's relations and subclasses if a *type* or *subclass_of* relation was specified as a top-down constraint. If a type or *subclass_of* relation was not specified, there is no estimation needed; CPU resources can be optimally split simply by polling the number of relations that point to the node (referred to as the node's in-relations).

To optimally split CPU resources when *type* or *subclass_of* is specified, the total number of nodes that could eventually be candidates is required. To illustrate the importance of this, consider the example in Fig. 3. A circle represents a class node, and squares represent instances of the class node (*i.e.* they possess a *type* relation to the circle). Connections between circles indicate the class hierarchy of the knowledge; *0* has *1* and *2* as subclasses, and so forth.

Suppose a retrieval request specified *type 1* as the only top-down constraint. The node *1* has three nodes connected through an in-relation of type and has two subclasses, nodes *3* and *4*, with three and one instances, respectively. Node *3* has an additional subclass, node *6*, which has one instance. All of these instances classify as an instance of *1* since they are instances of or instances of a derived class of *1*. Thus, the candidate set is all of the shaded boxes.

To show the importance of this example, consider how HADM could split CPU resources; from the top-level node *3*, HADM is completely unaware of the number of instances below node *3*. This number could be found by traversing the network, however, such a solution prevents maximizing parallelism or requires a multithreaded traversal. To optimally saturate CPU resources while traversing with a single thread, splitting the work must occur after the single thread is finished traversing the portion of the network.

A method that maximizes the saturation of all CPUs during candidate determination is optimal. To achieve this, a method needs to (1) start assigning tasks to workers as quickly as possible and (2) assign tasks in such a way that once the task queue is empty, all worker threads finish at approximately the same time. A thread pool enables both of these properties. The main thread can submit candidate determination tasks as it traverses the network while worker threads can begin executing tasks immediately. In Fig. 3, the HADM main thread accesses node *1*, and submits tasks to the thread pool to determine the candidacy of each instance. The thread pool generalizes the link between the structure and size of the network and the parallel work that must be completed.

The size of each task submitted to the thread pool is proportional to the amount of hardware concurrency available. This design decision was made while the thread pool used a single task queue for all tasks submitted. Scaling the size of each task reduced contention on the task queues. A typical task size for candidate determination is anywhere between 200 and 1,000 nodes per task (in practice, a single retrieval may require thousands of tasks to be submitted to the threadpool by HADM's main thread).

5.2.2 Spreading Activation and Activation Calculation using a Thread Pool

For spreading activation, a single task is submitted to the thread pool per context priming source. This decision reflects the simplicity of spreading activation: visit the node spreading activation, and distribute activation down the appropriate relations according to effective fan. The MRL and NtN enable all context priming sources in a maximum of 2 hash table lookups, followed by iterating over a node's in-relations, and finally adding activation contributions according to the spreading activation equation in Table 1. See Section 5.1 for more details regarding the MRL and NtN. In HADM, one task is submitted to the thread pool per context priming source spreading activation.

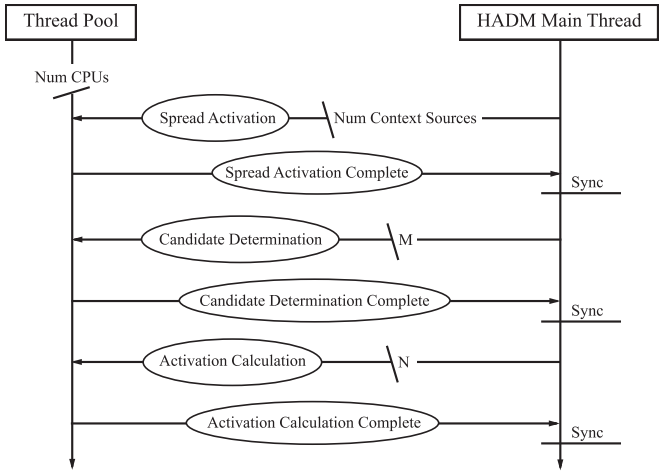

Fig. 4. HADM Host Retrieval Execution. The main thread submits tasks to the thread pool (right to left), and the thread pool sends responses in the form of futures (left to right). Each crossed line represents the thickness of the line. For the thread pool the thickness corresponds to the number of threads running and for each task (circles), the number of tasks submitted. *M* represents the total number of tasks submitted from the HADM main thread splitting in-relations and traversing the subsection of the network described in Section 5.2.1. *N* represents the total number of tasks submitted to the thread pool during activation calculation. Each *Sync* represents the main thread sleeping until all tasks in the thread pool have been completed.

The multithreading model for activation calculation remains largely the same as ADM's implementation. Each node that is a candidate for retrieval is placed into a task and submitted to the thread pool for completion. Each task submitted is the same size as the tasks submitted for candidate determination. However, in HADM this is achieved by submitting tasks to the thread pool, with each task containing portions of the candidate pool to compute. In ADM, threads were launched for each task to the same effect. Each task reports the node with the highest activation calculation through a reference contained in a future.

5.2.3 HADM's Host Retrieval Execution

The execution of HADM's main thread and thread pool follows a simple model: the HADM main thread submits tasks as it encounters work capable of being parallelized, as described in Sections 5.2.1 and 5.2.2, and waits for the submitted work to be completed by the thread pool. HADM's main thread follows the following retrieval execution, with each task submitting appropriate tasks to the thread pool: (1) spread activation, (2) candidate determination and (3) activation calculation. The HADM main thread collects the results of activation calculation, returning the node with the highest activation to working memory. This execution cycle between the thread pool and the HADM main thread is shown in Fig. 4.

6 Device Retrieval Implementation

The results of the ADM solution and HADM's host execution are markedly faster than previous implementations of declarative memory. ADM showed a 20x speedup over the soaDM solution [1], and the soaDM vastly outperformed traditional ACT-R (and every other retrieval system performing the full ACT-R retrieval calculus) [1], [2]. However, these acceleration goals fall short of goals of Air Force Research Lab (ARFL) Large Scale Cognitive Modeling (LSCM) objectives.

HADM's device retrieval execution is vastly different from HADM's host execution and ACT-R's retrieval. Each node in the semantic network is processed by a device thread with every retrieval. The HADM host execution directly accesses subsections of the semantic network, while ACT-R iterates over the knowledge. The entire device network is consulted in parallel with every device retrieval.

The device retrieval process can be summarized in four steps: (1) every node is visited by a device thread; (2) the device thread determines if this device node is a candidate; (3) if the node is a candidate, the device thread marks the node as a candidate and computes activation; and (4) the node with highest activation is returned to working memory. To prevent synchronization or network traversal on the device, each device node requires all information required to determine candidacy and if necessary, compute activation. This reduces the amount of synchronization at the cost of device memory.

6.1 Motivation

Building a declarative memory system from the ground up in C++ provided a significant speedup over previous DM implementations. However, declarative retrievals are inherently parallel; each node is independent of all other chunks for candidate determination and activation calculation. Spreading activation requires some communication between nodes (the node spreading to the node to the node receiving), but each node can compute the rest of the retrieval entirely on its own. In the face of this parallelism, the LSCM sought hardware architectures that could further accelerate declarative retrievals. General Purpose Computing on Graphics Processing Units (GPGPU) allows programmers to leverage the parallel computing capabilities of a Graphics Processing Unit, which can contain up to thousands of processing cores per unit. The parallel nature of declarative retrievals makes a GPU-based implementation an apt approach to acceleration.

6.2 Device Representation of Declarative Memory

To maximize the performance increase from using a GPU, each device thread operates completely independently of other threads. The device implementation of declarative memory can greatly leverage the host/device relationship to minimize synchronization. The device does not need to have a copy of the MRL, NtN, and other supporting data structures of DM; the host can effectively manage those structures while sending minimal information to the device for computation. For HADM, this means the device's semantic network should represent a network, but should not *be* a network. The processing of one device node should not interfere with the processing of another device thread, nor should device processes rely on traversing the network. HADM relies on the host to process and prepare the declarative memory to be loaded onto the device. HADM's hash tables are expanded to include the device representation of DM and are described in Section 6.2.1.

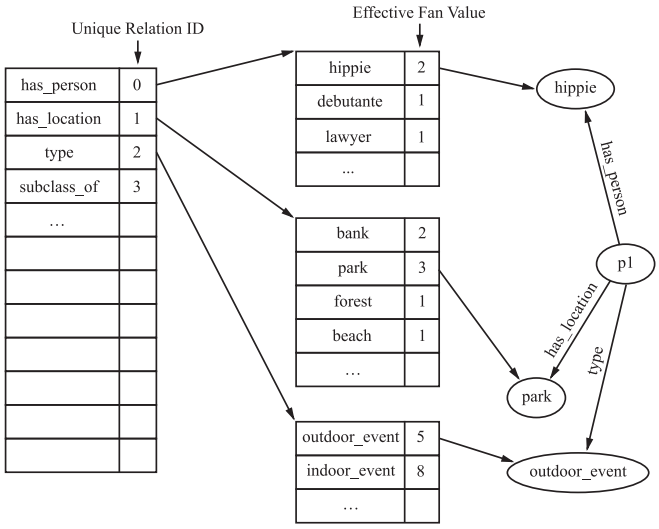

Fig. 5. Effective fan embedded in MRL and RL entries. Unique IDs are assigned to each relation (see Section 6.2.1).

6.2.1 ID Representation of Declarative Memory

On the device, the entire semantic network is represented through unique IDs. Each type of information has its own set of ID assignments, and each member of a type of information has a unique ID. An ID representation simplifies the structure and memory footprint of DM on the GPU. The MRL on the host is updated with another field; each value in the MRL is a pair between a unique ID and reference to a Relation List (RL). Each RL entry value is a pair between the effective fan value and a node reference, as described in Section 7.1. The resulting MRL, shown in Fig. 5, depicts the addition of unique relation IDs.

Each node in the network is also assigned a unique ID representation when the node is instantiated in the network. This information is contained within each node, and can be accessed through node references in the NtN described in [1]. With relations and node names assigned ID representations, the device DM needs an ID representation of data properties. A hash table is created for every key, value, and value type of a data property (for example, "Tom", "name", and "string", respectively). Using these hash tables, the host side can quickly map to an ID representation of every portion of a data property. These structures enable the host to effectively map every critical portion of the network to an ID. Tables 4 and 5 summarize the hash tables in HADM.

6.2.2 Copying Nodes from Host to Device

During host retrievals, the semantic network is accessed and traversed a number of times. Spreading activation requires visiting the source node and traversing references to nodes connected to the source node. If a type/subclass retrieval request is specified, candidate determination includes traversing all classes derived from the not specified in the retrieval request, as described in Section 5.2.1. Because the device network does not maintain an efficient method for network access (by design), the device nodes need a way to determine: (1) activation contributions from context priming sources and (2) ensure that proper type/subclass consideration is achieved when enforcing top-down retrieval constraints.

TABLE 5
HADM Device Hash Table Descriptions

Name	Description
Node ID to node reference	Maps unique node IDs to their respectively semantic network references.
Value to value ID	Maps every data property value in the network to a unique value ID.
Value type to value type ID	Maps every data property value type in the network to a unique value type ID.
Key to key ID	Maps every data property key in the network to a unique key ID.

Section 6.2 establishes that every device node should be able to determine its candidacy and activation calculation independently from all other device nodes. Thus, each device node needs lists of the following information: subclass node IDs, superclass node IDs, in-relations (each comprised of a relation ID and a node ID), out-relations (each comprised of a relation ID and a node ID), and data properties (each comprised of a value ID, value type ID, and a key ID).

Most of this device node structure is equivalent to the host node structure, with the exception that on the host, subclass and superclass nodes are stored through references and the network is traversed. When creating the device node, the host traverses the subclass and superclass nodes and collects all appropriate node IDs. Since each device node is aware of all of its sub and superclasses, it can independently determine its candidacy by comparing its internal lists to the constraints specified in the retrieval request. In effect, this removes the need to traverse the network to determine if a particular node is a subclass of the node specified by a type/subclass relation.

The copy of host nodes to device nodes requires a trivial intermediate conversion to a host-allocated version of the device node. Once the appropriate unique IDs have been copied into the device node structure on the host, the host uses the CUDA API to transfer the host-allocated device node onto device memory. The resulting allocation reference is inserted into a global device node table. The global device node table is a single dimension array of device node references. It is allocated after the initial ontology nodes have been loaded on the host. Device threads use their unique dimensions to determine which node in the global device node table they should process.

Network updates require accessing the entire device network with a specialized update request. This update request updates all values in the network appropriately (e.g. relations that need to be updated after adding a node to the network, fan, etc.). This network update procedure is functionally equivalent to ACT-R's update mechanism.

6.3 Device Retrieval Execution

The device paradigm operates from the perspective of each device thread; a device retrieval is centered on each device thread acting as an independent worker of the retrieval.

The device retrieval process follows the following steps: (1) convert the host retrieval request to a corresponding ID representation and transfer the request to the device, (2) launch enough device blocks and threads to ensure every

node in the device node table will determine its candidacy and if necessary, compute activation, (3) launch a distributed argmax to filter the resulting activation calculations down to the single node with the highest activation, (4) transfer the winning node's ID and retrieval computations to the host, and finally (5) convert the node ID to a node in the host network, update its activation calculation information, and return it to working memory.

6.3.1 ID representation of Retrieval Requests

The host's version of the retrieval request must be converted into an equivalent ID representation for device retrievals. The hash tables outlined in Table 5 are used to facilitate the conversion from strings to IDs. The MRL provides conversion from a relation to a relation ID. The NtN provides a node name to node ID conversion. The value to value ID, value type to value type ID, and the key to key ID provide converting data property filters to their corresponding ID equivalent. Once the string-based host retrieval request has been converted to its unique ID equivalent, the host-allocated device retrieval request is copied to device memory.

6.3.2 Device Candidate Determination and Activation Calculation

From the host, a retrieval kernel is launched with the corresponding device retrieval request. The dimensions of the kernel depend on the number of used portions within the device node table. For device retrievals, each block is launched with the maximum number of threads per block (*maxThreadsPerBlock*). The number of blocks needed for a retrieval is simply the number of nodes used in the device node table divided by *maxThreadsPerBlock* (only one grid is used for each retrieval). From here, the remaining algorithm is described from the perspective of a single device thread. Each device thread proceeds as shown in Algorithm 1.

Algorithm 1 oversimplifies the details of the retrieval process by indicating where the retrieval process follows ACT-R behavior. While these portions are lightly described here, more detailed explanations of ACT-R retrieval can be found in [1], [2], [9]. The device retrieval description is more concise than the host counterpart. The device retrieval checks each node for candidacy, rather than attempting to optimally access a subsection of the network and traversing the network appropriately. After launching the kernel on the device, the host waits for the device to signal that all blocks and threads have finished the initial phase of retrieval before proceeding (i.e. all device nodes have determined their candidacy and computed activation).

6.3.3 Retrieval Reduction

At this point in the device retrieval execution, all candidate nodes have computed activation. However, the device must locate the node with the highest activation, nor is there a trivial way to find it. HADM follows a distributed argmax reduction scheme. Once the distributed argmax finds the node with the highest activation, the device transfers the retrieval results back to the host, and the host uses the node ID to node reference hash table to look up the winning node on the host. The host then appropriately updates the information in the winning node and returns the winning node to working memory.

Algorithm 1. Single Device Node Candidate Determination and Activation Calculation

```
1: procedure EXECUTEDEVICERETRIEVAL(retrievalRequest, deviceNodeTable)
2:     idx ← blockDim * blockIdx + threadIdx
3:     if idx > size of deviceNodeTable then return        ▷verify idx validity
4:     node ← deviceNodeTable[idx]                         ▷access the node to process
5:     DETERMINECANDIDACY(node, retrievalRequest)
6:     if node is not a candidate then return
7:     CALCULATEACTIVATION(node, retrievalRequest)
8: procedure DETERMINECANDIDACY(node, retrievalRequest)
9:     candidate ← true                                    ▷assume node is a candidate, prove node is not
10:    for all constraint in retrievalRequest do
11:        if constraint's relation ID is type/subclass and constraint's node ID is not in node's superclass list then
12:            candidate ← false
13:        else if constraint is a data property filter and node does not posses the IDs as a property then
14:            candidate ← false
15:        else if constraint's relation ID and node ID are not in node's outRelations then
16:            candidate ← false
17: procedure CALCULATEACTIVATION(node, retrievalRequest)
18:    for all contextSource in retrievalRequest do
19:        for all outRel in node's outRelations do
20:            if outRel's nodeID matches contextSource's node ID then
21:                contribute to node's activation          ▷follows ACT-R spreading activation
22:    compute node's activation                           ▷follows ACT-R's retrieval calculus
```

6.3.4 Device Retrieval as MapReduce

The device retrieval execution still follows the basic tenants of a MapReduce-based retrieval, presented in [28]. The degree of parallelism is significantly higher than previous MapReduce-based declarative retrievals [1], [2]. The initial device kernel launch acts as a mapper, mapping each work unit (candidate determination and activation calculation) to a single device thread. The following two kernel launches act as reducers, reducing the results of the retrieval down to a single winning node.

7 Large Declarative Memory Issues

This section seeks to outline significant issues that arise when using large declarative memories. These findings resulted from the increased scale enabled by HADM.

With the exception of a few applications and research, declarative memories are traditionally small and do not run for extended periods of time. Several studies [1], [2], [3], [4], [11] describe attempts to utilize large declarative memories. Large declarative memories have unique and potentially problematic implications; specifically, the Associative

Strength equation in Table 1 can potentially render spreading activation detrimental to the retrieval process. A negative spreading activation contribution undermines the integrity of ACT-R's basis from human performance data, leading to behavior that is inconsistent with prior psychological data.

Fan is a numerical representation of the overall complexity of a piece of knowledge in the semantic network. This value is permitted to increase as new knowledge is added to DM (intuitively, new knowledge increases the complexity of related knowledge). As a DM grows, the value of fan for each affected chunk is adjusted appropriately to reflect the increased complexity. If the memory is allowed to grow unchecked, receiving spreading activation may result in a negative contribution to the overall activation of the chunk, putting a chunk related to the current context at a computational disadvantage for retrieval. This is the opposite behavior intended when activation spreads throughout the network. This occurs when $\ln(\text{fan}_j)$ becomes larger than $S_{\max}$, yielding in a negative S_{ji} (see equations in Table 1). Because a declarative memory is allowed to add knowledge during runtime, it is feasible that a fan_j becomes larger than the global $S_{\max}$. ACT-R was designed with relatively small, short behavior models in mind with $S_{\max}$ around 1.5 [9].

Researchers have been able to curtail this problem simply by increasing the $S_{\max}$ according to the size of DM. This requires tweaking declarative parameters for every ontology source. Some modelers are forced to set $S_{\max}$ as high as 20 [4]. While effective for the short term, this solution is not sustainable or scalable. This paper seeks to introduce and discuss ideas to prevent negative spreading activation contributions.

7.1 Mitigating Negative Spreading Activation Contributions through Effective Fan

soaDM and HADM utilize a previously unreported feature, referred to as effective fan. Effective fan allows modelers to specify a key/value pair to spread activation, rather than a traditional sole value. This type of spreading activation is shown in Table 2 with *has_person fireman*. With both a key and a value specified, spreading activation only passes activation to nodes connected to fireman through the *has_person* relation.

The fan_j of the resulting activation calculation is the number of nodes connected to fireman through the *has_person* relation. Thus, the term effective fan refers to the number of nodes that had activation spread to it from this context priming source. Traditional ACT-R behavior is achieved by sending a '*' as the key in the key/value pair, triggering activation to spread to all nodes connected to the value in the key/value pair.

At first glance, this introduces a large burden in HADM's optimized retrieval starting point. HADM iterates over all retrieval constraints, polling to find the node with lowest fan [1]. This is a cheap operation; it simply requires two hash lookups in the MRL and querying an array's size (the fan of the node). However, for effective fan, the overall fan is not a reliable indication of the smallest subset of potential retrieval candidates. Moreover, counting the effective fan during each retrieval is not computationally efficient and in practice drastically hinders the performance of retrievals (and in some cases, doubling overall retrieval time).

To prevent HADM from counting the effective fan during each retrieval, the effective fan is added and maintained in the MRL. The MRL is a hash table with every relation present in the entire semantic network as the key, and a reference to a RL is the value, as described in [1]. Each RL is another hash table. The key to each RL is the node name that receives the relation of the MRL (that is, the node which has the head of the directed edge pointing to it), and the value of each RL is a reference to the actual node. This relationship is visually shown in Fig. 5.

Fig. 5 also shows the additional structure added for effective fan usage. Each RL value becomes a pair, with the first value of the pair is the effective fan of this relation at the RL node (how many nodes have the MRL relation pointing to the node of the RL entry). Maintaining each effective fan is trivial; for each relation an incoming node possesses, HADM maintains the MRL and each RL by attempting to find each relation in the MRL, then each specified node in the corresponding RL. If HADM finds both already exist, increment the value of effective fan. If the relation or the proper RL entry does not exist, it creates the missing section and initializes effective fan to 1.

Effective fan is useful for two reasons: (1) it gives modelers a method to further specify retrieval influences and (2) often mitigates the likelihood of a node receiving a negative spreading activation contribution ($\ln(\text{fan}_j)$ becoming larger than $S_{\max}$). It is worth noting that the outcomes of utilizing effective fan over a traditional fan have not been verified against human performance data. Effective fan reduces the likelihood of a node receiving a negative spreading activation contribution simply by reducing the value of fan_j used in the associative strength equation of Table 1.

However, there is no guarantee that the value of a particular node's effective fan and fan are different (exactly the case when a node only has one relation pointing towards it). While effective fan could enable a lower overall $S_{\max}$ value, it still does not guarantee that a node cannot receive a negative spreading activation calculation contribution. The network could still grow beyond the value of $S_{\max}$, or a modeler could still use the traditional ACT-R behavior in a DM intended to run using solely effective fan.

7.2 Eliminating Negative Spreading Activation Contributions through a Dynamic Max Associative Strength

This paper seeks to introduce a new perspective of $S_{\max}$. While untested against human performance data, this idea is intended to spark a conversation and research along this direction. First, note the intention behind $S_{\max}$; the following observations summarize the properties of $S_{\max}$:

1) $S_{\max}$ should higher than the largest value of $\ln(\text{fan}_j)$ in the entire semantic network.
2) $S_{\max}$ should not be high enough over the largest value of $\ln(\text{fan}_j)$ to diminish the effects of base-level activation.

The first property indicates why a constant $S_{\max}$ will lead to negative spreading activation contributions if a declarative memory is allowed to grow indefinitely. The second

TABLE 6
Ontology Sources and Retrievals for Testing

Ontology source	Nodes computing activation	Retrieval requirements	Context priming sources		W	N
Fan effect	15	`type event`	`has_person` `has_location`	`hippie` `park`	1	3

Moby II (1 in 3)	145,073	`type synonym_relation`	`'*'`
			`'*'`
			`'*'`
				`taxing`
				`demanding`
				`straining`
					1	4

Moby II (1 in 2)	318,435	`type synonym_relation`	`word` `synonym`	`fair` `considerate`	1	3

Moby II (1 in 1)	1,281,763	`type synonym_relation`	`word`
			`synonym`
			`word`
				`mazy`
				`whimsical`
				`flighty`
					1	4

Moby II (full)	2,520,245	`type synonym_relation`	`word`
			`synonym`
			`word`
				`mazy`
				`whimsical`
				`flighty`
					1	4

property underscores why naively and endlessly increasing $S_{\max}$ with a growing DM changes the behavior of retrievals over the entire life of an agent.

To create a scalable and sustainable $S_{\max}$, we introduce the concept of a variable $S_{\max}$. This variable $S_{\max}$ would always remain higher than the largest $\ln(\text{fan}_j)$ in the network, eliminating the possibility of a negative spreading activation contributions. To address the second observation, we propose a variable $S_{\max}$ that is either: (1) a constant value higher than $\ln(\text{fan}_j)$ or (2) higher than $\ln(\text{fan}_j)$ by a function of the complexity/size of the network. The constant value approach would not diminish the effects of base-level activation but would yield different effects from base-level activation as $\ln(\text{fan}_j)$ grows (i.e. the effect of the constant value decreases as $\ln(\text{fan}_j)$ grows).

The discussion of effective fan and a variable $S_{\max}$ is not the purpose of this research. However, negative spreading activation contributions threaten the integrity of ACT-R's governing equations as a DM grows indefinitely. Given the importance of this issue for large, scalable declarative memories, the discussion here is meant to spark a conversation among the cognitive modeling community to address these issues and verify their validity against human performance data. Naive changes the behavior of $S_{\max}$ may result in models that are inconsistent with prior psychological data.

8 Experimental Setup

The principal ontology for testing HADM is the Moby II thesaurus, which is also used in [1], [2]. The Moby II thesaurus contains 30 thousand root words with 2.5 million synonym relations. The fan effect ontology from ACT-R is used to compare the performance with a small number of nodes. Table 6 shows the various ontology sources and the retrievals executed with the particular ontology. The Moby II (1 in 1) contains every synonym without spaces and the Moby II (full) contains every synonym. Note that when the context priming source includes a key in the key-value pair, effective fan is utilized. When '*' is used as the key, traditional ACT-R spreading activation behavior occurs.

The retrievals executed in Table 6 were chosen to maximize the number of nodes computing activation. These queries specify a single top-down constraint to maximize the percentage of the network computing activation; i.e. they enforce a stress-test of the system in a worst-case scenario where the entire semantic network must compute activation.

8.1 Hardware and Software Details

The hardware configuration contained two Intel Xeon CPU E5-2630 CPUs. Each CPU has 6 cores with support for 12 threads at a frequency of 2.3 GHz. An NVIDIA Tesla K20 was used as the GPU for HADM's device retrieval. The testing machine ran CentOS 6.7.

9 Results

Each retrieval in Table 6 was run 180 times and the average results are shown below in Table 7. HADM (host indicates a retrieval using only the host implementation of DM, while HADM (device) indicates a retrieval run using the device implementation of DM. The Moby II (full) ontology was not loaded into soaDM due to soaDM's performance with an ontology half the size and the linear performance of soaDM.

TABLE 7
Average Retrieval Times (in ms) over 180 Retrievals

Retrieval system	Fan effect	Moby II (1 in 3)	Moby II (1 in 2)	Moby II (1 in 1)	Moby II (full)
soaDM	6.800	287.734	593.127	2655.831	
HADM (host)	1.751	23.953	41.987	143.563	247.022
HADM (device)	0.492	3.608	6.480	27.321	53.309

Figs. 6 and 7 show a graphical representation of the results in Table 7. Fig. 6 shows the performance comparison between the soaDM, HADM (host), and HADM (device) retrieval systems.

Table 7, Figs. 6 and 7 show the expected linear performance of each retrieval system. The HADM device retrieval proves to be the most performant retrieval system in every retrieval test. This is somewhat surprising; there is

TABLE 8
Average Speedup between Retrieval Systems

	Fan effect	Moby II (1 in 3)	Moby II (1 in 2)	Moby II (1 in 1)	Moby II (full)	Average speedup
HADM (host) vs. soaDM	3.9	12.0	14.1	18.5		12.1
HADM (device) vs. soaDM	13.8	79.8	91.5	97.2		70.6
HADM (device) vs. HADM (host)	3.6	6.6	6.5	5.3	4.6	5.3

nontrivial overhead of launching kernels and transferring data between the host and device. Edmonds et al. [1] indicated the final HADM system would optimally switch between a single-threaded host, multi-threaded host, and device implementations of DM because of this overhead consideration. However, given these results, the device will always be used when available.

The exact performance comparisons are shown in Table 8. Each entry in Table 8 represents the relative performance of one retrieval to another for each ontology source. For example, the first row (HADM (host) versus soaDM) shows the amount of retrieval time decrease the HADM host implementation achieves over the soaDM implementation of DM. The final column of Table 8 shows the overall average increase between systems.

9.1 Profiling Analysis

Here, we present an analysis of the performance of HADM's host and device execution. Fig. 8 shows the proportion of time spent at each step of HADM's host execution (see Fig. 4 for algorithmic details). Fig. 9 shows the same plot for HADM's device execution (see Section 6 for algorithmic details). All proportions are averaged over 60 retrievals using the retrieval requests shown in Table 6.

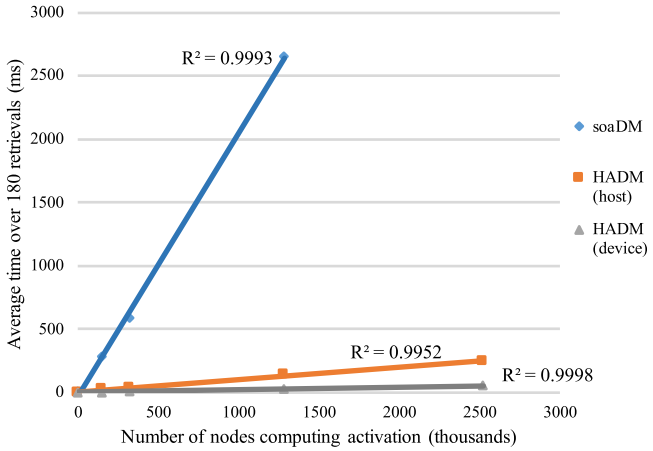

Fig. 6. soaDM, HADM (host), and HADM (device) performance comparison. We fit a linear model for each trend and all implementations have a very high R^2 value.

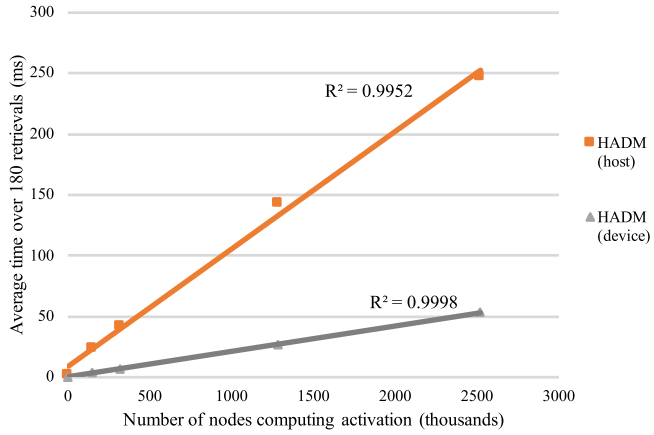

Fig. 7. HADM (host) and HADM (device) performance comparison. We fit the same linear model and see highly-linear performance for both the host and the device.

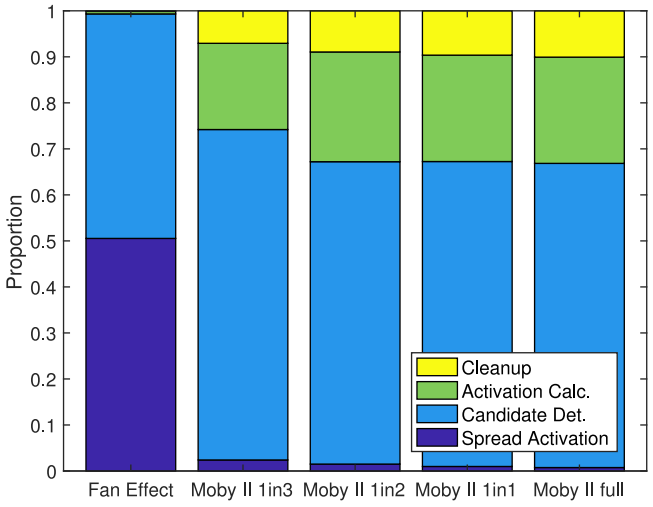

Fig. 8. HADM (host) profiling results. Shows the proportion of time spent in each sub-task of the retrieval process against each retrieval ontology using retrievals in Table 6.

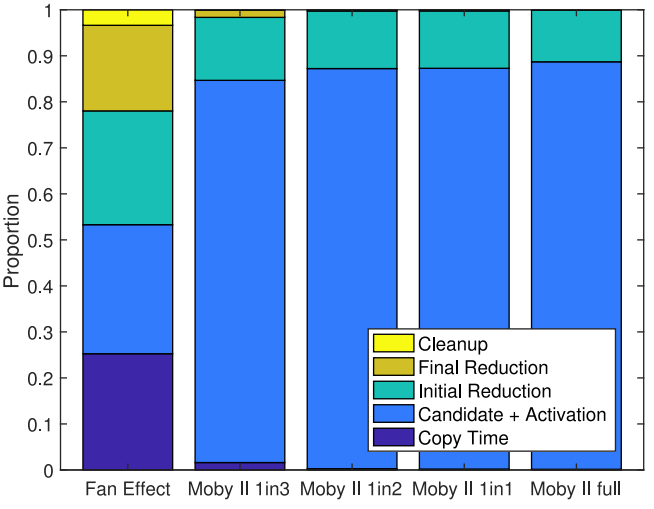

Fig. 9. HADM (device) profiling results. Shows the proportion of time spent in each sub-task of the retrieval process against each retrieval ontology using retrievals in Table 6.

Due to the Fan Effect ontologies' small size (only 15 nodes compute activation), the overhead of the system corresponds to a significant portion of the wall-clock time for both the host and the device executions. For the host, this overhead corresponds to submitting tasks to the thread pool and waiting for values from the thread pool to become available to the HADM main thread (i.e. futures). For the device implementation, the principle overhead lies kernel setup and host-device copy operations. These overhead operations are responsible for the slightly non-linear behavior shown for smaller ontologies in Fig. 7.

For both the host and device, the performance of the system stabilizes as the network size grows. For both systems, candidate determination and activation calculations require the most time. Candidate determination requires $O(nk)$ time a serial execution, where n represents the number of nodes connected to the node with the smallest fan (see Section 5.1) and k represents the number of top-down retrieval constraints. Parallelization efforts (e.g. HADM's host and device implementations) reduce this complexity by a factor of m, where m represents the degree of hardware parallelization available. It is worth noting that saturation of device nodes is non-uniform during candidate determination. Saturation is dependent upon retrieval candidates, which cannot be predicted prior to receiving a retrieval request.

10 Conclusion

HADM is the first GPU-based implementation of declarative memory. It marks another contribution towards creating a hardware accelerated cognitive architecture within the LSCM initiative at AFRL. This paper has a simple conclusion: declarative memory's inherently parallel activation calculus drastically benefits from leveraging parallel hardware. HADM represents a complete and extremely parallel declarative retrieval system built from the ground-up.

Declarative memory has never been given such performance considerations, and the results show a highly capable declarative memory system is achievable and practical. soaDM was the fastest implementation of declarative memory to date before ADM's creation [1], [2], and now HADM supersedes ADM. The contributions of this work are summarized as: (1) an analysis of fan for large declarative memories and a proposal for potential solutions; (2) an improvement of host execution on a declarative using a thread pool; and (3) the first ever-implementation of ACT-R declarative retrievals on parallel hardware. The results show parallel hardware drastically accelerates declarative retrievals.

10.1 Future Work

The work presented here currently does not support a multi-GPU device retrieval. A multi-GPU implementation would enable larger ontology sources which extended beyond the available memory in a single GPGPU. Future work includes exploring other parallel or distributed computing platforms, such as OpenMPI. Additionally, HADM should be extended to support ACT-R's numeric retrieval requirements, partial matching, and blending.

Future work should also include a careful analysis and experimentation regarding the methods to mitigate or remove the disastrous effects of a negative spreading activation calculation when a declarative memory is allowed to grow indefinitely. The implications of effective fan or a variable $S_{\max}$ parameter must be analyzed by the cognitive modeling community. These ideas presented here are intended to spark a conversation among the cognitive modeling community about how declarative memory can guarantee positive spreading activation contributions with large, long-running DMs.

Other recent hardware-accelerated cognitive architectures include accelerating the knowledge mining of a cognitive domain ontology, presented in [11]. Combining these systems and implementing other hardware accelerated modules of the CECEP architecture would represent a cognitive architecture capable of supporting agents which utilize massive stores of knowledge.

Acknowledgments

Described research was partially supported by the Air Force Office of Sponsored Research (AFOSR) Repperger internship program and the Department of Energy Oak Ridge Institute for Science & Education (ORISE) program.

References

[1] M. Edmonds, T. Atahary, T. Taha, and S. A. Douglass, "High performance declarative memory systems through MapReduce," in *Proc. 16th IEEE/ACIS Int. Conf. Softw. Eng. Artif. Intell. Netw. Parallel/Distrib. Comput.*, 2015, pp. 1–8.

[2] S. A. Douglass and C. W. Myers, "Concurrent knowledge activation calculation in large declarative memories," in *Proc. 10th Int. Conf. Cognitive Model.*, 2010, pp. 55–60.

[3] S. Douglass, J. Ball, and S. Rodgers, "Large declarative memories in ACT-R," in *Proc. 9th Int. Conf. Cognitive Model.*, 2009, Art. 234.

[4] D. D. Salvucci, "Endowing a cognitive architecture with world knowledge," in *Proc. 36th Annu. Meeting Cognitive Sci. Soc.*, 2014, pp. 1353–1358.

[5] N. Derbinsky, J. E. Laird, and B. Smith, "Towards efficiently supporting large symbolic declarative memories," in *Proc. 10th Int. Conf. Cognitive Model.*, 2010, pp. 49–54.

[6] N. Derbinsky and J. E. Laird, "A functional analysis of historical memory retrieval bias in the word sense disambiguation task," *Ann Arbor*, vol. 1001, pp. 48 109–42 121, 2011.

[7] P. S. Rosenbloom, "Towards a 50 msec cognitive cycle in a graphical architecture," in *Proc. 11th Int. Conf. Cognitive Model.*, 2012, pp. 305–310.

[8] P. S. Rosenbloom, A. Demski, and V. Ustun, "Efficient message computation in Sigma's graphical architecture," *Biologically Inspired Cognitive Architectures*, vol. 11, pp. 1–9, 2015.

[9] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, "An integrated theory of the mind," *Psychological Rev.*, vol. 111, no. 4, 2004, Art. no. 1036.

[10] P. Langley, J. E. Laird, and S. Rogers, "Cognitive architectures: Research issues and challenges," *Cognitive Syst. Res.*, vol. 10, no. 2, pp. 141–160, 2009.

[11] T. Atahary, T. M. Taha, and S. Douglass, "Hardware accelerated cognitively enhanced complex event processing architecture," in *Proc. 14th ACIS Int. Conf. Softw. Eng. Artif. Intell. Netw. Parallel/Distrib. Comput.*, 2013, pp. 283–288.

[12] J. R. Anderson, "A spreading activation theory of memory," *J. Verbal Learning Verbal Behavior*, vol. 22, no. 3, pp. 261–295, 1983.

[13] J. R. Anderson, *Language, Memory, and Thought*, London, U.K.: Psychology Press, 1976.

[14] J. R. Anderson, *The Architecture of Cognition*. London, U.K.: Psychology Press, 1983.

[15] J. E. Laird, *The Soar Cognitive Architecture*. Cambridge, MA, USA: MIT Press, 2012.

[16] D. D. Salvucci and F. J. Lee, "Simple cognitive modeling in a complex cognitive architecture," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2003, pp. 265–272.
[17] S. J. Jones, A. R. Wandzel, and J. E. Laird, "Efficient computation of spreading activation using lazy evaluation," in *Proc. 14th Int. Conf. Cognitive Model.*, 2016, pp. 182–187.
[18] Y. Chen, M. Petrovic, and M. H. Clark, "SemMemDB: In-database knowledge activation," *Proc. 27th Int. Florida Artif. Intell. Res. Soc. Conf.*, 2014, pp. 18–23.
[19] M. A. Kelly, K. Kwok, and R. L. West, "Holographic declarative memory and the fan effect: A test case for a new memory module for ACT-R," *Can. J. Exp. Psychology*, vol. 69, pp. 365–365, 2015.
[20] A. Oltramari and C. Lebiere, "Extending cognitive architectures with semantic resources," in *Proc. Int. Conf. Artif. Gen. Intell.*, 2011, pp. 222–231.
[21] M. Grinberg, V. Haltakov, and H. Stefanov, "Approximate spreading activation for efficient knowledge retrieval from large datasets," in *Proc. 20th Italian Workshop Neural Nets.*, 2011, Art. no. 326.
[22] A. Nuxoll and J. E. Laird, "A cognitive model of episodic memory integrated with a general cognitive architecture," in *Proc. 4th Int. Conf. Cognitive Model.*, 2004, pp. 220–225.
[23] A. M. Nuxoll and J. E. Laird, "Enhancing intelligent agents with episodic memory," *Cognitive Syst. Res.*, vol. 17, pp. 34–48, 2012.
[24] F. Li, J. Frost, and B. J. Phillips, "An episodic memory retrieval algorithm for the soar cognitive architecture," in *Proc. Australasian Joint Conf. Artif. Intell.*, 2015, pp. 343–355.
[25] J. E. Laird, "Extending the Soar cognitive architecture," in *Proc. 1st AGI Conf. Artif. Gen. Intell.*, 2008, vol. 171, pp. 224–235.
[26] J. Frost, M. W. Numan, M. Liebelt, and B. J. Phillips, "A new computer for cognitive computing," in *Proc. IEEE 14th Int. Conf. Cognitive Informat. Cognitive Comput.*, 2015, pp. 33–38.
[27] P. S. Rosenbloom, "Combining procedural and declarative knowledge in a graphical architecture," in *Proc. 10th Int. Conf. Cognitive Model.*, 2010, pp. 205–210.
[28] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[29] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

Mark Edmonds received the BS degree from the University of Dayton, and the MS degree from the UCLA. He is working toward the PhD degree at the University of California, Los Angeles. His specializations are in artificial intelligence, parallel computing, causal learning, and robotics. He has worked on a wide range of projects from cognitive architectures, learning from demonstration, causality, and reinforcement learning.

Tanvir Atahary received the BS and MS degrees from the University of Dhaka, Bangladesh, in 2006 and 2008, respectively, and the PhD degree from the University of Dayton, in 2016. Before starting his PhD, he worked as full time faculty with the University of Liberal Arts (ULAB), Bangladesh from 2008 to 2011. He has been closely working with Wright Patterson Air Force Base (WPAFB) from his 1st year of doctoral study and spent consecutive five summers at WPAFB. After completing his PhD, he joined the University of Dayton as a Research Engineer and WPAFB a full time contractor. His research interests include high performance computing and cognitive computing architectures.

Scott A. Douglass received the PhD degree in cognitive psychology from Carnegie Mellon University, in 2007. He is a senior cognitive scientist with the 711/HPW Supervisory Control and Cognition Branch (RHCI), US Air Force Research Lab, Wright-Patterson Air Force Base, Ohio. Working with John R. Anderson at CMU, he acquired expertise in cognitive architectures and the modeling and simulation of complex situated cognitive processes. His research interests include cognitive computing, artificial intelligence, knowledge engineering, multi-formalism modeling, answer-set programming, and complex event processing.

Tarek Taha received the BSEE, MSEE, and PhD degrees in electrical engineering from the Georgia Institute of Technology. He is a professor of Electrical and Computer Engineering. His specializations are in high performance computing, neuromorphic computing, and artificial intelligence systems. He works closely with the Air Force Research Lab, the National Security Agency. He has several projects on the topics of neuromorphic processors, including the development of autonomous agents for these platforms, deep learning, neuromorphic architectures for cybersecurity, and memristor fabrication and circuit design. He has spent several summers at the NSA and AFRL. He is a recipient of the NSF CAREER Award.