A Parallel CPU-GPU Framework for Batching Heuristic Operations in Depth-First Heuristic Search

Ehsan Futuhi[1], **Nathan R. Sturtevant**[1,2]
[1]University of Alberta
[2]Alberta Machine Intelligence Institute (Amii)
{futuhi, nathanst}@ualberta.ca

Abstract

The rapid advancement of GPU technology has unlocked powerful parallel processing capabilities, creating new opportunities to enhance classic search algorithms. This hardware has been exploited in best-first search algorithms with neural network-based heuristics by creating batched versions of A* and Weighted A* that delay heuristic evaluation until sufficiently many states can be evaluated in parallel on the GPU. But, research has not addressed how depth-first algorithms like IDA* or Budgeted Tree Search (BTS) can have their heuristic computations batched. This is more complicated in a tree search, because progress in the search tree is blocked until heuristic evaluations are complete. In this paper we show that GPU parallelization of heuristics can be effectively performed when the tree search is parallelized on the CPU while heuristic evaluations are parallelized on the GPU. We develop a parallelized cost-bounded depth-first search (CB-DFS) framework that can be applied to both IDA* and BTS, significantly improving their performance. We demonstrate the strength of the approach on the 3x3 Rubik's Cube and the 4x4 sliding tile puzzle (STP) with both classifier-based and regression-based heuristics.

Introduction

There has been significant recent growth in computational resources, particularly in GPUs (Dally, Keckler, and Kirk 2021; Rotem et al. 2022). GPUs have become indispensable for computation-intensive tasks due to their massive parallelism, capable of performing millions of operations simultaneously. Modern CPUs also continue to evolve with enhanced parallel processing capabilities, enabling faster execution of complex algorithms. This advancement in both CPU and GPU technologies has been well exploited in many fields of Artificial Intelligence, especially deep learning (Schrittwieser et al. 2020; Yao et al. 2024).

In classical search algorithms, several approaches (Li et al. 2022; Zhou and Zeng 2015; Agostinelli et al. 2019, 2024) have been developed to enhance search using the parallel processing capabilities of modern GPUs. One potential use of deep learning on GPUs is to learn heuristics to guide search. For instance, Li et al. (2022) introduced admissible neural network heuristics that compresses a large pattern database (PDB) heuristic with less information loss than standard compression techniques (Felner et al. 2007; Helmert, Sturtevant, and Felner 2017).

A* (Hart, Nilsson, and Raphael 1968) and Weighted A* (Pohl 1970) have had batch versions developed which can use GPU-based heuristics more efficiently for optimal (Li et al. 2022) and sub-optimal search (Agostinelli et al. 2019). The batched versions collect states into batches to evaluate their heuristics in a single parallel neural network lookup using the GPU. This technique, called *batch heuristic evaluations*, utilizes GPU parallelism and significantly improves performance over performing individual neural network lookups for each state.

Algorithms like IDA* (Korf 1985) have not yet had batched variants built. The depth-first nature of IDA* search complicates the batching process, because in a depth-first search there are often only a few states available at one time, while batching is most efficient when hundreds of states are available for computing heuristic values in parallel. The same issue applies to algorithms such as Budgeted Tree Search (BTS) (Helmert et al. 2019; Sturtevant and Helmert 2020), as both IDA* and BTS are built upon performing repeated cost-bounded depth-first searches (CB-DFS).

This paper addresses the issue by observing that approaches used for CPU parallelization of CB-DFS make many more states available for batching, and thus enable efficient use of GPU-based heuristics. By re-designing CB-DFS for parallel search, we can then effectively build batch versions of IDA* and BTS. The effectiveness of the batched versions of IDA* and BTS is shown using both regression-based and classifier-based heuristics on the Rubik's Cube and 15-puzzle domains. Batching is highly effective in improving the performance of these algorithms, resulting in over a $40\times$ improvement in search speed. This improvement in search performance opens the door for further research on improving heuristic quality, and for designing new search algorithms to handle the inadmissible values that we expect to find in large neural network-based heuristics.

Background and Related Work

In *heuristic search*, the broad task is to find a path in a graph $\{G = \{V, E\}, s, g, c, h\}$ from a start state $s \in V$ to a goal state $g \in V$, where $c : E \rightarrow \mathbb{R}^+$ is a cost function associated with the edges between states. The heuristic function $h(v)$

provides an estimate of the distance from a state v to the goal g. The heuristic is considered *admissible* if, for all states v, $h(v)$ does not exceed the true shortest distance $h^*(v)$ to the goal. It is *consistent* if, for any two states a and b, the heuristic satisfies $h(a) \leq c(a,b)+h(b)$. In large state spaces, the graph G is represented implicitly, meaning it is generated online by expanding states and exploring their neighbors. A* and IDA* are guaranteed to find optimal solutions when the heuristic is admissible (Felner et al. 2011).

Heuristics

Pattern Database (PDB) heuristics (Culberson and Schaeffer 1998) and the related merge and shrink framework (Helmert et al. 2007) are widely utilized, particularly in problems that exhibit exponential growth (Gnad, Sievers, and Torralba 2023). These heuristics abstract the original graph V into a reduced state space $\phi(V)$. In this abstract space, edges between vertices in the original graph are preserved in the abstracted graph, meaning if an edge exists between v_1 and v_2 in V, a corresponding edge will exist between $\phi(v_1)$ and $\phi(v_2)$ in $\phi(V)$. As a result, abstract distances are admissible estimates of distances in V. PDBs can reduce the size of the state space exponentially with only a small loss in heuristic accuracy (Felner, Sturtevant, and Schaeffer 2009).

Standard PDB compression methods (Felner et al. 2007; Helmert, Sturtevant, and Felner 2017) primarily treat the PDB as a table of numbers. These methods group entries to reduce the PDB's size and replace each entry in a group with the smallest value in that group to ensure the heuristic remains admissible. One common method is *DIV*, where k adjacent entries are grouped by dividing the index by k. Another method is *MOD*, which combines entries offset by $\frac{m}{k}$ in a PDB with m total entries using the modulo operator.

Neural networks have been used to compress PDB heuristics. ADP (Samadi et al. 2008) used a range of techniques to ensure admissibility. These included a unique loss function to penalize overestimation, a decision tree to partition states, and employing ANNs only for the resulting subsets of states. Any states with inadmissible heuristics were then stored in a hash table. ADP was developed prior to current hardware and is orthogonal to work in this paper on designing more efficient algorithms that use these heuristics.

Li et al. (2022) also studied approaches for learning admissible heuristics. They treated the learning of heuristics as a classification problem rather than using regression, because in NP-complete problems the solution length is polynomial, meaning a small number of heuristic values (classes), while the state space is exponential. Admissibility is guaranteed in two ways. First, because the heuristic classes are ordered, the classification quantile used for the predicted class can be adjusted to ensure admissibility. Second, an ensemble of neural networks can be trained, with the minimum value from the ensemble used as the prediction. This technique leverages the diversity of the ensemble to produce an admissible heuristic.

Search Algorithms

A variety of search algorithms can be used for solving shortest path problems.

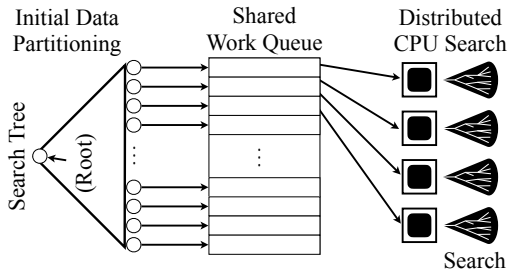

Figure 1: Structure of AIDA*.

IDA* combines a cost-bounded depth-first search (CB-DFS) with iterative deepening to find optimal solutions. IDA* performs a series of depth-first searches, each with an increasing cost threshold, which is determined by the current path cost and the heuristic estimate to the goal. IDA* increases the cost threshold conservatively so it can terminate once the goal is found. IDA* is memory-efficient, as it only requires storage for the current path. However, if the number of nodes in each iteration does not grow fast enough, the sum of costs of the iterations may outweigh the cost of the final iteration, increasing the total expansions to $O(N^2)$.

BTS (Helmert et al. 2019) is identical to IDA* when the iterations grow by a suitable constant factor, but reduces the worst-case overhead to $O(N \log C^*)$, where C^* is the optimal solution cost. As in IDA*, BTS repeatedly invokes CB-DFS, just with more aggressive thresholds and additional node expansion limits. Thus, both IDA* and BTS can be improved through GPU parallelization of CB-DFS.

AIDA* is designed to parallelize the CB-DFS portion of IDA*. While some forms of depth-first search cannot be parallelized well (Reif 1985), AIDA* exploits the structure of the problem to create independent subproblems. AIDA* has three phases for CPU parallelization of the CB-DFS portion of IDA*, as shown in Figure 1:

Initial Data Partitioning In this phase a single search is performed from the root of the search tree where the leaves of the search have cost greater than the current cost threshold. The threshold is increased until there are sufficiently many leaves above the cost threshold to efficiently perform the following algorithmic steps. At the end of this phase, duplicate nodes are eliminated from the frontier nodes.

Distributed Node Assignment Next, the leaf nodes from the data partitioning are put into a shared work queue for distributed assignment of work. Path information is maintained to prevent the search from returning to the parents.

Distributed CPU Search In this phase, processors independently search the *subtrees* found below the states in the shared work queue, maintaining the lowest unexpanded f-cost in the search. In each iteration either a solution is found and the search terminates, or no solution is found, and the search is repeated at the next cost threshold. Since the work queue is shared among all threads, idle threads dynamically retrieve unprocessed work from the queue as they complete their current work.

GPU Architecture

GPU parallelization is fundamentally different than CPU parallelization. While in CPU parallelization we need independent subtasks, GPU parallelization works best with correlated tasks that are solved with the same instructions but different data. GPUs use a hierarchical execution model that enhances parallel processing efficiency. At the heart of this model is the *kernel*, which executes across multiple thread blocks. Each block contains warps, groups of threads that execute the same instructions in parallel. Threads within a block can share data through on-chip memory, but blocks themselves work independently.

The architecture of GPUs is designed to support this model, with each GPU featuring multiple *Streaming Multiprocessors* (SMs). These SMs include on-chip memory, *shader cores*, and *warp schedulers*. Shader cores handle arithmetic and logic operations, while warp schedulers manage the execution of warps, selecting which ones are ready to execute in each cycle. GPUs can be connected to systems either via the PCI-E bus, as in Ubuntu servers, or integrated on the same processor package as the CPU, like in Apple's M1 or M2 chips. When connected through PCI-E, GPUs typically have dedicated memory, necessitating explicit data transfers between CPU and GPU memory.

GPU Parallelization of Algorithms

In recent years, several parallel versions of the IDA* algorithm have been introduced. AIDA* (Reinefeld and Schnecke 1994), a highly parallel iterative-deepening search algorithm, was designed for large-scale asynchronous Multiple Instruction, Multiple Data (MIMD) systems. The algorithm partitions the search space and processes it asynchronously across multiple CPU processors. Taking a different approach, Horie and Fukunaga (2017) investigate the parallelization of IDA* on GPUs using a block-based approach. The proposed Block-Parallel IDA* (BPIDA*) assigns subtrees to blocks of threads that execute on the same *streaming multiprocessors (SMs)*. BPIDA* takes advantage of local shared memory of within a SM to reduce warp divergence and improve load balancing.

GPU parallelism has also been effectively utilized in other search algorithms. Zhou and Zeng (2015) introduced the first parallel variant of the A* search algorithm called GA* that leverages the computational power of GPUs. GA* uses multiple parallel priority queues to manage the Open list, enabling the simultaneous extraction and expansion of nodes across GPU threads. The heuristic computations are also parallelized across the GPU cores to optimize performance further. GA* is up to 45x faster than a traditional CPU-based A* implementations in large and complex search spaces. Q* search (Agostinelli et al. 2024) employs deep Q-networks to calculate combined transition costs and heuristic values for child nodes in a single forward pass, thereby eliminating the need to explicitly generate them.

Edelkamp and Sulewski (2009) investigate bitvector-based search algorithms, where the GPU's parallel processing capabilities allow for the efficient handling of state expansion and duplicate detection. The GPU is also employed for ranking and unranking permutations, computing hash functions, and managing the search frontier, all of which are parallelized to exploit the GPU's architecture. Meanwhile, other approaches have formulated A* and Weighted A* algorithms as differentiable and end-to-end trainable neural network planners (Yonetani et al. 2021; Archetti, Cannici, and Matteucci 2021). These data-driven approaches, rather than learning the heuristic function, take a raw image as an input and convert it to a guidance map by an encoder. GPUs have also been instrumental in learning heuristics. For example, Li et al. (2022) developed an admissible heuristic for the sliding tile puzzle (STP) and TopSpin using an ensemble of neural networks and classifier quantiles. Other approaches (Agostinelli et al. 2021; Arfaee, Zilles, and Holte 2011; Thayer, Dionne, and Ruml 2011; Pándy et al. 2022) have also focused on heuristic learning, though without guaranteeing admissibility.

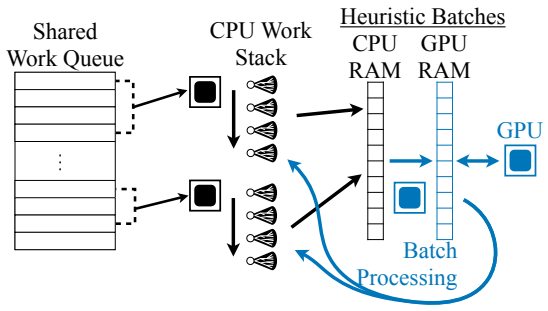

Figure 2: The structure of SingleGPU CB-DFS.

CB-DFS for Neural Heuristics

In a heuristic search problem, the heuristic, h, can come from any source. This paper studies the *neural heuristic search* problem. In this problem, h is a neural heuristic. This means that $h \in H_{NN}$, where H_{NN} is the set of all heuristics that are computed by a neural network. In recent work, heuristics in H_{NN} have been learned from PDB heuristics (Li et al. 2022), and general techniques for heuristic learning have been described (Khandelwal, Sheth, and Agostinelli 2024). The aim of this paper is to improve algorithms that make use of such heuristics. Experimental results evaluate the quality of heuristics we currently have access to, but we are working under the assumption that neural network heuristics will continue to improve, and thus improved algorithms will be broadly beneficial.

We now introduce the SingleGPU Batch CB-DFS algorithm, which can be used with both IDA* to create Batch IDA* and BTS to create Batch BTS. We provide pseudocode and prove theoretical properties. Then, we discuss the Multi-GPU CB-DFS algorithm.

SingleGPU Batch CB-DFS

Batch heuristic lookups in best-first search algorithms like A* are relatively easy because once the search gets started, there are many states waiting in the open list. But, in a cost-bounded depth-first tree, the depth-first search cannot continue until the f-cost of all children is known. Thus, to get

Algorithm 1: Batch IDA*

```
1: Input: h_M, start s, goal g, d_init
2: works, history, batch ← {}
3: bound ← h_M(s, g)
4: foundSolution ← false
5: GenerateWork(s,d_init,history)
6: start a batch-processing CPU thread executing ProcessBatch()
7: while not foundSolution do
8:     start search CPU threads executing CB-DFS(bound)
9:     wait for CB-DFS threads to end
10:    UpdateThreshold(bound)
11: end while
```

Algorithm 2: Parallel CB-DFS

```
1: function CB-DFS(bound)
2:     Initiate stack[workNum]
3:     Initialize terminated ← [0, 0, . . . , 0] of length workNum
4:     counter ← 0
5:     miss ← 0
6:     for i = 1 to workNum do
7:         stack[i] ← works.pop()
8:     end for
9:     while miss < workNum do                ▷ wait for all works to
10:        if stack[counter] is done then          be done
11:            if works is not empty then
12:                stack[counter] ← works.pop()
13:            else
14:                miss ←miss+1
15:                terminated[counter] ← 1
16:            end if
17:        end if
18:        if not terminated[counter] then
19:            DoIteration(stack[counter],bound)
20:        end if
21:        counter ← counter+1
22:    end while
23: end function
```

Algorithm 3: Subtree expansion

```
1: function DoIteration(work, bound)
2:     newStatesFound ← false
3:     while not newStatesFound do
4:         s ← work.GetTop()
5:         if h_M(s) is not ready then
6:             return
7:         end if
8:         if h_M(s,goal)<bound then
9:             newStatesFound←true
10:        end if
11:    end while
12:    actions ← env.GetActions(s)
13:    for each action in actions do
14:        s_next ← env.ApplyAction(s, action)
15:        work.add(s_next)
16:        batch.add(TensorRepresentation((s_next)))
17:    end for
18: end function
```

sufficient states for batching heuristic lookups, we must either have a large enough branching factor to permit efficient batching, search speculatively beyond the cost limit, or run several CB-DFS searches in parallel so we can batch states across subtrees. This paper explores the last approach.

The overall structure of *SingleGPU CB-DFS* is illustrated in Figure 2. It begins similarly to AIDA* by generating a *Shared Work Queue* with subtrees that can be searched independently. However, instead of having each CPU take a single subtree, the CPUs now fill a work stack with multiple subtrees. A single expansion is performed on the first subtree, which results in a set of children that need their heuristics evaluated before the search can continue. These states are placed into a shared *heuristic batch queue* on the CPU, and then work continues on the next subtree in the work stack, generating more states until the heuristic batch queue is full. If multiple CPUs are available, they can execute this process in parallel.

Once the heuristic batch queue is full, a dedicated *batch-processing* thread manages the batch evaluation of states. This involves (1) copying the data to GPU memory, (2) evaluating the batch on the GPU using the model h_M, and then (3) returning the resulting heuristics to each of the CPUs. The batch is then cleared, allowing the next set of generated states to be processed. More than one batch is maintained in RAM so the CPUs can continue to search in parallel to the batch evaluation. The CB-DFS completes when the shared work queue is exhausted and all threads have completed any searches remaining in their work stacks. This completes the high-level CB-DFS.

The pseudo-code showing batch CB-DFS and how it is integrated into SingleGPU Batch IDA* is outlined in Algorithm 1. The `GenerateWork` function (line 5) is called once before the main search loop begins, instructing an initial search tree to depth d_{init}. BatchIDA* then invokes `CB-DFS` in parallel across CPU threads given the cost threshold of the high-level search (line 8). The cost threshold is updated (`UpdateThreshold` function) after the CB-DFS with the current threshold is completed (line 10).

The `DoIteration` function (Algorithm 3) expands subtrees one at a time. While executing the `DoIteration` function, if a subtree cannot progress further—meaning the heuristic evaluation is not yet available for the top node (line 5) – the thread switches to the next subtree in its stack. Upon revisiting the same subtree, the thread resumes from the same node if its heuristic evaluation is ready; otherwise, it switches again to the next subtree.

The batch processing is done via the `ProcessBatch` function, which is not shown. The most important detail of this algorithm is that it has a timeout after which a batch is evalauted even if it is not completely full. This is important when very little work is available at the end of an iteration, and impacts the integration with BTS.

Overall, the performance of Batch IDA* hinges on the balance between two key events: *filling the batch* and *processing the batch*. The time required to fill the batch depends on the speed of the CPU CB-DFS threads, while processing time is determined by the GPU's efficiency and the transfer time via the PCI-E bus. If batch-filling is faster, CB-DFS threads may remain idle, either waiting to add new nodes or for heuristic evaluations. Conversely, if batch processing is

faster, the GPU is underutilized. Therefore, these two processes must be balanced to ensure optimal resource utilization during the CB-DFS.

Batch CB-DFS with Multiple GPUs

In most servers, multiple GPU devices are available, although the design shown thus far only incorporates a single GPU. If there are many fast CPUs available, they can end up waiting on the GPU for heuristic evaluations.

We scale Batch CB-DFS to multiple GPUs by loading h_M onto each GPU, and running one batch-processing thread per GPU. CB-DFS threads are then uniformly assigned to the available GPUs, with each GPU processing the batch of nodes generated by its respective CB-DFS threads. This setup maximizes CPU utilization by increasing the number of batches while ensuring sufficient GPU resources are available to process them concurrently.

Correctness of Batch CB-DFS

In this section, IDA*, BTS, and AIDA* are all implemented with a CB-DFS sub-routine. Thus, as long as Batch CB-DFS searches the same tree as a classic CB-DFS search, we can substitute Batch CB-DFS without impacting the correctness of these algorithms. The batching operation does not impact the fact that the cost limit in the search is bounded, because the cost bound is still respected in the subtree expansions. Detailed proofs are found in an extended version of this paper (Futuhi and Sturtevant 2025).

Batch BTS Enhancements

One important feature of the BTS algorithm is that it avoids the worst case where the IDA* tree grows too slowly, which can happen when there are many unique h-costs (e.g. as a result of using a regression-based heuristic). BTS guarantees that in each iteration the number of nodes will grow exponentially. It does this by establishing a limit of how fast the tree should grow in the next iteration (measured by node expansions), and then does a search over the cost bounds to find the cost bound that achieves this growth. Thus, in Batch BTS we must impose an additional node expansion limit in the CB-DFS. For efficiency, we do not check this after every expansion, but often enough to ensure that the search will terminate within a constant bound of the limit.

Next, we note that using classic heuristics, there is little cost for running many small iterations at the very beginning of search. But, with neural heuristics, doing so is very slow, because the search never grows large enough to exploit large batch sizes. Thus, a very small iteration of BTS can be a thousand times more expensive than a larger iteration.

To mitigate this, we make two additional modifications in Batch BTS. First, instead of setting the initial node limit to a small constant, such as 1, we use a much larger constant – in our case 10,000 expansions. This allows the initial searches to be significantly larger. But, setting the node limit on its own is not sufficient, we also need to raise the cost bound of the first search. Thus, instead of using $h(\text{root})$, we use $h(\text{root}) + 1$ instead. Neither of these changes impact the asymptotic behavior of the algorithm. Because the overheads are just large constants, on sufficiently large problems these modifications would not be necessary. But, in practice, they significantly improve performance.

Memory Cost Analysis

The memory cost of IDA* is $\mathcal{O}(d)$, where d is the solution depth (Korf 1985). For AIDA*, which utilizes n CB-DFS threads, the memory cost increases to $\mathcal{O}(dn)$ as it concurrently expands n subtrees. In Batch IDA* and Batch BTS's CB-DFS, each thread manages k subtrees in a stack, resulting in a memory cost of $\mathcal{O}(dnk)$. We know that n and k are related to hardware speed on any domain. So, these parameters are expected to be constant as hardware is constant. Additionally, while the heuristic evaluations for unexpanded nodes are stored temporarily, memory is freed after their expansion, keeping the total memory cost at $\mathcal{O}(dnk)$.

AIDA*, Batch IDA*, and Batch BTS require loading their respective heuristics into memory. If AIDA* uses a PDB heuristic h and Batch IDA* uses a learned model h_M, the total memory cost for AIDA* is $\mathcal{O}(nd + size(h))$ and for Batch IDA* it is $\mathcal{O}(dnk + size(h_M))$. While PDB heuristics typically fit into memory and thus their cost is constant, large PDBs, such as the 12-edge Rubik's cube PDB (500 GB), exceed memory capacity, making them impractical for AIDA*. This limitation underscores the importance of further work on compressing PDB heuristics using deep learning and designing search algorithms, such as the CB-DFS used by Batch IDA* and Batch BTS, to use them efficiently.

Experimental Results

The primary purpose of the experimental results is to evaluate whether or not the batched version of CB-DFS can effectively reduce the overhead of neural heuristics through GPU parallelism. We evaluate this in both the Batch IDA* and Batch BTS algorithms with neural heuristics built on both classifiers and regression, as regression-based heuristics are expected to fail with IDA*, but not BTS. We further evaluate the impact of GPU count and neural heuristic size on performance. Additional experiments examining the effects of hardware and algorithmic hyperparameters are included in the extended version (Futuhi and Sturtevant 2025).

Experimental Setup

We conduct our experiments across two domains: the 3x3 Rubik's cube and the 4x4 sliding tile puzzle (STP). We use relatively modest computational resources in our experiments. Specifically, we utilize a server equipped with 32 AMD Ryzen Threadripper 2950X CPU cores and two NVIDIA GeForce RTX 2080 Ti GPUs running CUDA version 12.4. The Batch algorithms are implemented in C++, and Libtorch is employed for the deep learning components. Any CPU parallelization uses one thread per CPU core. For batched algorithms, we use a 4-millisecond timeout to trigger batch processing when an insufficient number of states are available to fully populate the batch. For the standard algorithms, IDA* and AIDA*, we use h_{PDB} which is the 8-corners PDB for Rubik's cube, and the sum of 1-7 and 8-15 tile additive PDBs for STP. For the STP, we use standard

Average Performance Over 50 Instances							
Algorithm	Batch Size	3x3 Rubik's Cube			4x4 STP		
		Time (s)	Expanded	Generated	Time (s)	Expanded	Generated
SingleGPU Batch IDA*	1	> 100	-	-	16.80	104,869	225,542
SingleGPU Batch IDA*	8	> 100	-	-	3.97	93,610	201,217
SingleGPU Batch IDA*	80	58.07	902,026	11,966,615	0.94	93,575	201,145
SingleGPU Batch IDA*	800	5.46	908,522	12,054,452	0.54	104,945	225,704
SingleGPU Batch IDA*	8,000	3.46	900,725	13,734,532	0.71	115,190	248,108
2GPU Batch IDA*	800	2.78	982,864	13,044,563	0.44	104,833	225,468
2GPU Batch IDA*	8,000	2.51	936,426	13,172,161	0.64	115,876	249,264
Batch A*	1,000	23.58	779,495	14,030,893	0.18	17,892	41,465

Table 1: Summary results comparing Batch IDA* and Batch A* using the same PDB heuristic.

benchmark instances (Korf 1985). For the Rubik's Cube domain, unless otherwise stated, we generated instances of solution length 11 using random walks starting from the goal state. PDB heuristics and baselines are from HOG2[1].

The heuristic h_M for the STP domain is the model introduced by Li et al. (2022). It is constructed by combining two ensemble models: the first is trained on the difference between the 1–7 tile PDB and the Manhattan distance heuristic, while the second is similarly trained using the 8–15 tile PDB. For the Rubik's Cube domain, we trained two admissible heuristics h_M, both based on the 8-corner PDB, with a size of 1.9 MB. The first model is a classifier that was trained extensively to achieve both admissibility and the same average heuristic as in the input PDB. The second model is a regression network trained using mean squared error (MSE) loss. For admissibility, we subtracted 2.2 from all predicted heuristic values. However, this adjustment reduced the average heuristic by 2.3 compared to the perfect heuristic.

Fixed-Tree Evaluation

Our first experiment is designed to evaluate the effectiveness of batch operations in Batch IDA* with a classifier-based heuristic. We designed this experiment to isolate as many variables as possible in the experiment and to isolate the quality of neural heuristic from the mechanics of Batch IDA*. We isolate the impact of the neural heuristic training by fixing the tree to only contain states expanded by a baseline PDB heuristic (h_{PDB}). The search evaluates the neural heuristic at each state, but uses h_{PDB} for pruning.

Under this methodology all algorithms are searching exactly the same tree. This allows us to initialize the neural heuristic randomly and quickly experiment with different network topologies without having to re-train the neural heuristics. This approach can measure the effectiveness of parallelism, but does not measure the quality of any learned heuristics, which is not the primary concern of this paper.

Results of this experiment are presented in Table 1. Each algorithm is given 5000s to solve all problems in the problem set, or an average of 100s per problem. Batch IDA* with a batch size of 1 or 8 fails to solve all of the Rubik's Cube instances within the time limit. But, increasing the batch size to 8000 reduces the average time to 3.46s. Using two GPUs and a batch size of 8000 further reduces this to 2.51s, which is almost ten times faster than Batch A*. Note that Batch A* is only parallel on the GPU, not the CPU. Adding CPU parallelization to Batch A* is an open challenge.

We use the first 50 instances from Korf's benchmarks for the STP domain. On this domain, going from batch size of 1 to 800 reduces the running time from 16.80s to 0.54s, a $30\times$ improvement. Using two GPUs and batch size 800 reduces this further to 0.44s. The variance in nodes expanded is due to a deep initial phase in Batch IDA* aimed at producing sufficient work pieces, which results in batches containing extra nodes that are not necessary to solve the problem (f-cost greater than the threshold). Although Batch IDA* is faster than Batch A* with respect to time per node expansion, Batch A* expands fewer nodes because it detects duplicates, which Batch IDA* does not.

[1]https://github.com/nathansttt/hog2/tree/PDB-refactor

Average Performance Over 50 RC Instances			
Algorithm	Batch Size	Time ± SE (s)	Expanded
Solution Length 8			
Batch BTS	10	33.41 ± 5.80	537,763
Batch BTS	100	4.26 ± 0.54	569,475
Batch BTS	1,000	0.97 ± 0.13	571,179
Batch BTS	2,000	0.83 ± 0.11	583,135
Batch IDA*	1,000	163.20 ± 54.63	366,530,531
Batch IDA*	2,000	144.91 ± 37.63	360,097,779
Solution Length 9			
Batch BTS	10	169.73 ± 65.58	1,211,327
Batch BTS	100	20.25 ± 6.58	1,217,398
Batch BTS	1,000	3.64 ± 0.94	1,294,972
Batch BTS	2,000	3.01 ± 0.94	1,294,329
Batch IDA*	2,000	-	-

Table 2: Performance on 3x3 Rubik's Cube instances with the regression model.

Regression-Based Neural Heuristics

In Table 2 we evaluate the impact of using a real-valued heuristic on Batch IDA* and Batch BTS. In this experiment both algorithms are using an identical neural heuristic, and

Average Time (s) Over 50 Instances				
Model size (MB)	SingleGPU Batch IDA*		Batch A*	
	RC	STP	RC	STP
0.2	3.46	0.52	11.16	0.18
2.3	4.75	0.78	16.71	0.29
13.6	10.23	1.45	20.66	0.38

Table 3: Impact of model size on Batch IDA* speed.

Average Performance Over 50 Instances				
Algorithm	RC		STP	
	Time (s)	Generated	Time (s)	Generated
4GPU Batch IDA*	2.56	12,027,388	0.57	238,440
2GPU Batch IDA*	2.97	11,322,876	0.58	239,492
SingleGPU Batch IDA*	6.11	12,498,111	0.99	231,958
AIDA*	2.26	14,793,061	0.02	101,020

Table 4: Impact of GPU counts on Batch IDA* performance.

we are measuring the cost of searching the resulting tree with that algorithm. Because the heuristic is learned using regression, which produces real-valued heuristics, we the IDA* iterations to grow very slowly. This is confirmed by the experimental results.

Due to the poor performance of Batch IDA*, we only report results where the algorithm is able to solve all instances within the 4-hour time limit. Batch IDA* expands orders of magnitude more states than Batch BTS, reinforcing that Batch BTS is better suited for regression-based heuristics. On instances with solution length 9, Batch IDA* expands billions of nodes per instance and fails complete within the time limit, while Batch BTS successfully solves all instances across all batch sizes. Notably, Batch BTS achieves a $56\times$ speedup when the batch size is increased from 10 to 2000.

Model Size

Our third experiment looks at the impact of model size on CB-DFS performance. We run Batch IDA* on the same domains with a single GPU and the best parameters from Table 1. We vary the model size by fixing the number of layers, but changing the number of fully connected neurons in each layer. The results of this experiment are found in Table 3. These results show that increasing the size of the model does not linearly increase the time required to evaluate the model, although larger models are more expensive to evaluate. Thus, there will be trade-offs between model quality and speed when deploying neural heuristics.

The results also show that Batch A* is less affected by model size than Batch IDA*. The main reason for this difference is that the batches in Batch IDA* are not always full, meaning the GPU isn't being used as efficiently. This can be caused by node generations above the cost thresholds and the uneven distribution of work between subtrees. When the model requires more time to evaluate, these issues are accentuated. See the extended version (Futuhi and Sturtevant 2025) for further analysis.

Average Performance Over 50 Instances				
Algorithm	4*4 STP			
	batch size	Time (s)	Expanded	Generated
SingleGPU Batch IDA*	8000	13.11	777,652	1,598,045
2GPU Batch IDA*	8000	11.78	770,341	1,428,177
AIDA*	-	1.21	10,513,092	30,254,284
IDA*	-	4.23	8,180,928	17,639,025
Batch A*	1000	2.26	88,977	273,505

Table 5: STP results using a learned admissible heuristic.

Average Performance Over 50 Instances				
Algorithm	3*3 Rubik's cube			
	batch size	Time (s)	Expanded	Generated
2GPU Batch IDA*	8000	5.11	1,001,733	13,028,957
SingleGPU Batch IDA*	8000	8.06	1,005,874	13,082,769
AIDA*	-	2.74	2,218,215	29,401,255
IDA*	-	5.84	2,106,615	27,900,766
Batch A*	1000	42.07	779,491	14,030,761

Table 6: RC results using a learned admissible heuristic.

GPU Count

To evaluate the impact of GPU count, we present the performance of Batch IDA* with different numbers of GPUs in Table 4, using other servers for this experiment. In both domains, performance improves with additional GPUs, with the 4-GPU version matching AIDA*'s performance on the Rubik's Cube. However, the performance gains diminish with increasing GPU counts, due to fewer threads being available to populate batches. We assess Batch IDA* using various hardware configurations, with results provided in the extended version (Futuhi and Sturtevant 2025).

PDB Comparison

We conclude by evaluating neural heuristics in Batch IDA* against AIDA*. Batch algorithms rely on admissible neural network heuristic h_M, while standard algorithms use a compressed PDB heuristic h_{DIV} of equal size. Results for STP and Rubik's Cube are presented in Table 5 and Table 6, respectively. The slower performance of learning algorithms compared to the results in Table 1 is due to the need for additional tensor operations per node to ensure heuristic admissibility. Standard algorithms expand more nodes in both domains since h_{DIV} is a weaker heuristic than h_M. AIDA* performs significantly better than all other algorithms, while among the batch algorithms, Batch IDA* remains faster than Batch A* in terms of constant time per node. An analysis of this performance gap between AIDA* and Batch IDA* is in the extended version (Futuhi and Sturtevant 2025).

Discussion and Conclusions

This paper has shown how to parallelize CB-DFS on the CPU and GPU for efficiently using neural heuristics during search. CB-DFS is explored with both Batch IDA and Batch BTS. The batching approach provides significant gains when batch sizes reach a sufficiently large size. Thus, this work provides the foundation for future research on building larger and stronger neural heuristics.

Acknowledgements

This work was supported by the National Science and Engineering Research Council of Canada Discovery Grant Program and the Canada CIFAR AI Chairs Program. We extend deepest appreciation to our colleague who contributed significant feedback to improving this work.

References

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8): 356–363.

Agostinelli, F.; McAleer, S.; Shmakov, A.; Fox, R.; Valtorta, M.; Srivastava, B.; and Baldi, P. 2021. Obtaining approximately admissible heuristic functions through deep reinforcement learning and A* search. In *Bridging the Gap between AI Planning and Reinforcement Learning workshop at ICAPS*.

Agostinelli, F.; Shperberg, S. S.; Shmakov, A.; McAleer, S.; Fox, R.; and Baldi, P. 2024. Q* Search: Heuristic Search with Deep Q-Networks.

Archetti, A.; Cannici, M.; and Matteucci, M. 2021. Neural Weighted A*: Learning graph costs and heuristics with differentiable anytime A*. In *International Conference on Machine Learning, Optimization, and Data Science*, 596–610. Springer.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17): 2075–2098.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence*, 14(3): 318–334.

Dally, W. J.; Keckler, S. W.; and Kirk, D. B. 2021. Evolution of the graphics processing unit (GPU). *IEEE Micro*, 41(6): 42–51.

Edelkamp, S.; and Sulewski, D. 2009. Parallel state space search on the GPU. In *Symposium on Combinatorial Search (SoCS)*.

Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. C. 2007. Compressed pattern databases. *Journal of Artificial Intelligence Research*, 30: 213–247.

Felner, A.; Sturtevant, N.; and Schaeffer, J. 2009. Abstraction-based heuristics with true distance computations. *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 9.

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artificial Intelligence*, 175(9-10): 1570–1603.

Futuhi, E.; and Sturtevant, N. R. 2025. A Parallel CPU-GPU Framework for Batching Heuristic Operations in Depth-First Heuristic Search. *arXiv preprint arXiv:2507.11916*.

Gnad, D.; Sievers, S.; and Torralba, Á. 2023. Efficient Evaluation of Large Abstractions for Decoupled Search: Merge-and-Shrink and Symbolic Pattern Databases. *International Conference on Automated Planning and Scheduling (ICAPS)*, 33(1): 138–147.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Helmert, M.; Haslum, P.; Hoffmann, J.; et al. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 176–183.

Helmert, M.; Lattimore, T.; Lelis, L. H. S.; Orseau, L.; and Sturtevant, N. R. 2019. Iterative Budgeted Exponential Search. In *International Joint Conference on Artificial Intelligence*, 1249–1257.

Helmert, M.; Sturtevant, N.; and Felner, A. 2017. On variable dependencies and compressed pattern databases. In *Symposium on Combinatorial Search (SoCS)*, volume 8, 129–133.

Horie, S.; and Fukunaga, A. 2017. Block-parallel IDA* for GPUs. In *Symposium on Combinatorial Search (SoCS)*, volume 8, 134–138.

Khandelwal, V.; Sheth, A.; and Agostinelli, F. 2024. Towards Learning Foundation Models for Heuristic Functions to Solve Pathfinding Problems. *arXiv preprint arXiv:2406.02598*.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1): 97–109.

Li, T.; Chen, R.; Mavrin, B.; Sturtevant, N. R.; Nadav, D.; and Felner, A. 2022. Optimal search with neural networks: Challenges and approaches. In *Symposium on Combinatorial Search (SoCS)*, volume 15, 109–117.

Pándy, M.; Qiu, W.; Corso, G.; Veličković, P.; Ying, Z.; Leskovec, J.; and Liò, P. 2022. Learning graph search heuristics. In *Learning on Graphs Conference*, 10–1. PMLR.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4): 193–204.

Reif, J. H. 1985. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5): 229–234.

Reinefeld, A.; and Schnecke, V. 1994. AIDA*-Asynchronous Parallel IDA*. In *Canadian Society for Computational Studies of Intelligence*, 295–302.

Rotem, E.; Yoaz, A.; Rappoport, L.; Robinson, S. J.; Mandelblat, J. Y.; Gihon, A.; Weissmann, E.; Chabukswar, R.; Basin, V.; Fenger, R.; et al. 2022. Intel alder lake CPU architectures. *IEEE Micro*, 42(3): 13–19.

Samadi, M.; Siabani, M.; Felner, A.; and Holte, R. 2008. Compressing pattern databases with learning. In *ECAI 2008*, 495–499. IOS Press.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839): 604–609.

Sturtevant, N.; and Helmert, M. 2020. A Guide to Budgeted Tree Search. In *Symposium on Combinatorial Search (SoCS)*.

Thayer, J.; Dionne, A.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 21, 250–257.

Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; and Narasimhan, K. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Yonetani, R.; Taniai, T.; Barekatain, M.; Nishimura, M.; and Kanezaki, A. 2021. Path planning using neural A* search. In *International Conference on Machine Learning*, 12029–12039. PMLR.

Zhou, Y.; and Zeng, J. 2015. Massively parallel A* search on a GPU. In *AAAI Conference on Artificial Intelligence*.