Parallelizing Classical Planning Critical Path Heuristics on the GPU

Hypergraph convolutions for computing h^m

Markus Fritzsche David Speck Daniel Gnad Simon Ståhlberg

ICAPS 2026

Classical planning leaves hardware idle

Observation

Search is mostly sequential. GPUs are not.

- Open/closed lists are hard to parallelize
- Cheap heuristics do not amortize transfer overhead
- Expensive heuristics can

Candidate

h^m: admissible, informative, many independent updates.

Bottleneck

One table entry for every atom set of size $\leq m$.

The h^m recurrence

Let s be a set of atoms obtained by regression.

$$h^m(s) = \begin{cases} 0 & s \subseteq I, \\ \min\limits_{(s,a,s') \in R(P)} \left(h^m(s') + \mathrm{cost}(a) \right) & |s| \leq m, \\ \max\limits_{s' \in \mathcal{P}_{\leq m}(s)} h^m(s') & \text{otherwise.} \end{cases}$$

Three cases

1. true initially: 0
2. small set: regress
3. too large: hardest small subset

For search state s

Initialize $d^0(v) = 0$ if $v \subseteq s$, else ∞.

One h^2 update loop

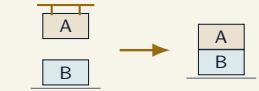
`stack(A,B)`

pre: H_A, C_B add: O_{AB}, C_A, E

$H_A = \text{holding}(A)$,
$C_X = \text{clear}(X)$
$O_{XY} = \text{on}(X, Y)$,
$E = \text{handempty}$.

One h^2 update loop

`stack(A,B)`



pre: H_A, C_B add: O_{AB}, C_A, E

$H_A = \text{holding}(A)$,
$C_X = \text{clear}(X)$
$O_{XY} = \text{on}(X, Y)$,
$E = \text{handempty}$.

Max+add kernel

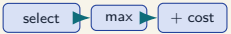
One work item per edge:

$$u_e = \max_{x \in T_e} d(x) + \text{cost}(e)$$

`stack(A,B)`, $c = 1$

tail tuple	initial
$\emptyset$	0
$\{H_A\}$	0
$\{C_B\}$	0
$\{H_A, C_B\}$	0

$\max(0, 0, 0, 0) + 1 = 1$



One h^2 update loop

`stack(A,B)`

pre: H_A, C_B add: O_{AB}, C_A, E

$H_A = \text{holding}(A)$,
$C_X = \text{clear}(X)$
$O_{XY} = \text{on}(X, Y)$,
$E = \text{handempty}$.

Max+add kernel

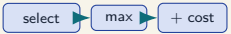
One work item per edge:

$$u_e = \max_{x \in T_e} d(x) + \text{cost}(e)$$

`stack(A,B)`, $c = 1$

tail tuple	initial
$\emptyset$	0
$\{H_A\}$	0
$\{C_B\}$	0
$\{H_A, C_B\}$	0

$$\max(0, 0, 0, 0) + 1 = 1$$



Grouped min-update loop

Generated heads decrease:

$$d(h) \leftarrow \min(d(h), \min_{e \to h} u_e)$$

tuple	old	after 1
$\{H_A\}$	0	0
$\{C_B\}$	0	0
$\{O_{AB}\}$	∞	1
$\{C_A\}$	∞	1
$\{E\}$	∞	1
$\{H_B\}$	∞	∞
$\{O_{BA}\}$	∞	∞
$\{O_{AB}, C_A\}$	∞	1
$\{O_{AB}, E\}$	∞	1
$\{C_A, E\}$	∞	1
...	...	...
$\{O_{AB}, O_{BA}\}$	∞	∞

Hypergraph representation

Vertices

$$X = \mathcal{P}_{\leq m}(V)$$

One vertex per table entry.

Hyperedges

Regression transition (s, a, s'):

$$T = \mathcal{P}_{\leq m}(s'), \qquad H = \{v\}, \quad v \in \mathcal{P}_{\leq m}(s),$$

with weight $\mathrm{cost}(a)$.

Hypergraph representation

Vertices

$$X = \mathcal{P}_{\leq m}(V)$$

One vertex per table entry.

Hyperedges

Regression transition (s, a, s'):

$$T = \mathcal{P}_{\leq m}(s'), \qquad H = \{v\}, \quad v \in \mathcal{P}_{\leq m}(s),$$

with weight $\mathrm{cost}(a)$.

GPU view

The expensive part becomes two reductions over fixed arrays:

$$\text{tail values} \xrightarrow{\max} \text{edge proposals} \xrightarrow{\min} \text{head values}$$

- tails and heads are index tensors
- action costs are edge weights
- atom-set costs are vertex labels

One convolution round

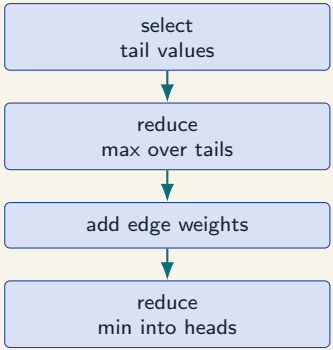
For each hyperedge $e = (T, \{v\})$:

$$u_e^t = \max_{x \in T} d^t(x) + \mathrm{cost}(e).$$

Update by cheapest incoming proposal:

$$d^{t+1}(v) = \min\left(d^t(v), \min_{e=(T,\{v\}) \in E} u_e^t\right).$$

Repeat to fixed point.



PyTorch: select, reduce-max, reduce-min.

Batching and cost functions

Batching

Evaluate all successors in parallel.

- Same graph
- Many label columns
- Higher GPU occupancy

Batching and cost functions

Batching

Evaluate all successors in parallel.

- Same graph
- Many label columns
- Higher GPU occupancy

Cost partitioning

Use cost functions $c_1, \ldots, c_n$ satisfying

$$c_1(a) + \cdots + c_n(a) \leq \mathrm{cost}(a).$$

- The graph topology is shared
- Edge weights differ per cost function
- Additive heuristic stays admissible
- Dominance pruning: stricter test

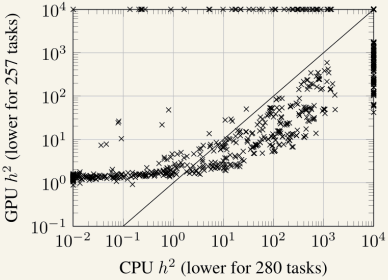
Experimental setup

- IPC optimal-track benchmarks
- A* search
- 30 min, 8 GiB CPU memory
- NVIDIA A100, 40 GB

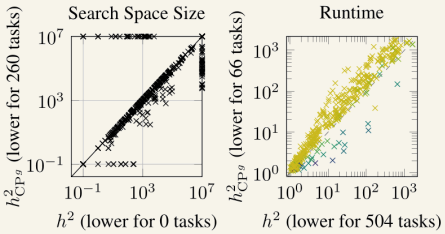
Compared configurations

- CPU h^2 baseline
- GPU h^2
- Batched GPU B-h^2
- Cost partitioning variants
- LM-cut baseline

Runtime and cost partitioning



Small tasks: overhead.
Large tasks: up to 100× speedup.



Less search, but often more runtime/memory.

Coverage

domain	n	**CPU**	LM	**GPU**	B-**GPU**	GPU-CPr	GPU-CPg	$\Delta(B\text{-GPU}, \text{CPU})$
elevators-08	30	9	22	16	19	15	15	+10
elevators-11	20	7	18	13	16	12	12	+9
sokoban-08	30	13	30	22	22	19	20	+9
organic-split	20	14	17	2	2	2	1	−12
tidybot-11	20	8	14	0	0	0	0	−8
mystery	30	17	17	11	11	11	9	−6
others (equal coverage)	454	99	150	99	99	99	99	0
others (remaining)	1223	409	705	449	473	413	413	+64
Sum	**1827**	**576**	**973**	**612**	**642**	**571**	**569**	**+66**

domain color: average parallelism — low — medium — high — out of memory

Conclusion

- h^m: fixed hypergraph, repeated convolution
- GPU tensor operations compute the update
- Batching makes heuristic evaluation practical
- Search stays sequential; the heuristic work becomes parallel

Code

downward-h2

markus.fritzsche@liu.se

Backup: Contribution

- Fixed hypergraph for the h^m dynamic program
- Bellman-Ford iteration as hypergraph convolution
- PyTorch/LibTorch tensor primitives
- Batching and multiple cost functions
- Fast Downward evaluation on IPC benchmarks

Backup: Implementation strategy and GPU work

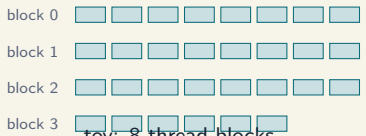
- Fast Downward integration
- CPU builds hypergraph once
- LibTorch 2.1.2 + CUDA 12.1
- Delayed convergence checks
- CUDA graphs for repeated launches

CUDA launch shape

Fixed-size blocks:

$$\texttt{threads} = 256, \qquad \texttt{blocks} = \left\lceil \frac{\texttt{work}}{256} \right\rceil .$$

Example: $n_{\mathrm{cp}} = 2$, $B = 3$, $|E| = 5$
$2 \cdot 3 \cdot 5 = 30$ edge work items



toy: 8-thread blocks
implementation: 256-thread blocks

What changes the thread count?

Logical work: $n_{\mathrm{cp}} B|E|$ for max+add and $n_{\mathrm{cp}} B|H_{\mathrm{grp}}|$ for grouped min. Tail size affects work per thread.

Backup: Tensor implementation from the paper

```
1: all_vals = index_select(
     input=vertices, dim=0,
     index=value_indices)
2: max_vals = index_reduce(
     input=all_vals, dim=0,
     index=max_indices, source=all_vals,
     reduce="amax", include_self=False)
3: max_vals = max_vals + edge_weights
4: vertices = index_reduce(
     input=vertices, dim=0,
     index=min_indices, source=max_vals,
     reduce="amin", include_self=True)
```

Select tails, reduce max, add weights, reduce min into heads.

Backup: Computing a heuristic value

Given a search state s:

$$d^0(v) = \begin{cases} 0 & \text{if } v \subseteq s, \\ \infty & \text{otherwise.} \end{cases}$$

Let d^* be the converged labeling. The heuristic is

$$h^m(s) = \max_{g \in \mathcal{P}_{\leq m}(G)} d^*(g).$$

Why this is correct

At convergence, $d^*(v) = h^m(v)$ for every vertex v.

GPU-friendly structure

Same parallel update each round.

Backup: Dominance pruning

Theorem

Let $e_1 = (T_1, H)$ and $e_2 = (T_2, H)$ be hyperedges with the same head. If $T_1 \subseteq T_2$ and $\mathrm{cost}(e_1) \leq \mathrm{cost}(e_2)$, then removing e_2 does not change the computed vertex values.

Reason

Larger tail: no smaller max. Larger weight: no compensation.

Cost partitioning caveat

With several cost functions, dominance must hold for every cost function, so the pruning condition becomes stricter.