GPU-Accelerated Relaxed Decision Diagrams for Branch-and-Bound Optimization

Fabio Tardivo
University of Connecticut, School of Computing, Storrs, CT, USA

Laurent Michel
Synchrony Chair in Cybersecurity, U. of Connecticut, School of Computing, Storrs, CT, USA

Willem-Jan van Hoeve
Carnegie Mellon University, Tepper School, Pittsburgh, PA, USA

Abstract

Branch-and-bound methods for combinatorial optimization rely critically on the efficient computation of strong bounds during search. Decision diagram–based optimization provides such bounds via restricted and relaxed multi-valued decision diagrams (MDDs), but compiling relaxed diagrams can become a computational bottleneck for existing solvers. We present a GPU-accelerated implementation of decision diagram–based branch-and-bound using a decoupled architecture. It separates the compilation of relaxed and restricted diagrams and coordinates them through two queues of search states. This design enables heterogeneous parallelization: restricted diagrams are compiled concurrently on CPU threads while relaxed diagrams are constructed in parallel on a GPU. The GPU implementation exploits the layered structure of decision diagrams by expanding states in parallel and performing successor generation, dominance filtering, and state merging on the GPU. Computational experiments on knapsack, maximum independent set, and Golomb ruler benchmarks demonstrate substantial performance improvements over CPU-based decision diagram solvers, including speedups of up to an order of magnitude on hard instances and the ability to solve Golomb ruler instances up to size 16.

2012 ACM Subject Classification Computing methodologies → Combinatorial algorithms; Computing methodologies → Massively parallel algorithms; Computing methodologies → Discrete space search

Keywords and phrases Decision Diagrams, GPU Computing, Dynamic Programming, Combinatorial Optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2026.61

Supplementary Material *Software*: `https://github.com/ldmbouge/CODD/releases/tag/CP26`

Funding *Laurent Michel*: Supported by Synchrony.

1 Introduction

Branch-and-bound methods remain one of the most general approaches for solving combinatorial optimization problems. Their effectiveness depends critically on the quality and computational cost of the bounds used to prune the search tree. Decision diagram–based techniques have recently emerged as a powerful mechanism for computing such bounds through relaxed and restricted multi-valued decision diagrams (MDDs) [1, 11]. In this framework, restricted diagrams provide feasible solutions while relaxed diagrams yield dual bounds through state merging, enabling an exact branch-and-bound procedure.

The strength of this approach depends on the width of the decision diagrams. Larger widths typically yield tighter bounds and more effective pruning, but the cost of constructing the diagrams increases rapidly. Existing implementations rely primarily on sequential or CPU-parallel methods, which limit the achievable diagram width and constrain the scalability

of the approach [2, 4, 11, 10]. Specifically, the current (CPU) parallel approaches parallelize the search process [2, 4].

A key structural property of decision diagram compilation is that diagrams are constructed layer by layer, where the expansion of states within a layer is largely independent. This structure exposes substantial data parallelism. In particular, successor generation, dominance filtering, duplicate elimination, and state merging can be applied independently across large sets of states. These operations match well with the execution model of modern GPUs, which are designed to perform uniform computations over large arrays of data.

In this paper, we investigate the use of GPUs to accelerate the construction of relaxed decision diagrams within a branch-and-bound solver. Our approach relies on a decoupled design of the search process that separates the compilation of relaxed and restricted diagrams. The overall search control remains on the CPU while the computationally intensive relaxation step is offloaded to the GPU. The GPU kernels implement state expansion, including successor generation, filtering using bounding and dominance rules, and merging operations required to enforce width limits in relaxed diagrams. Within this architecture, CPU threads compile restricted diagrams for pruning while the GPU constructs relaxed diagrams to compute bounds, allowing both components to operate concurrently during the search.

We evaluate the proposed approach on several combinatorial optimization benchmarks, including knapsack, maximum independent set, and Golomb ruler instances. The experiments show that GPU acceleration significantly reduces the runtime of relaxed diagram construction and yields speedups of up to an order of magnitude compared to state-of-the-art sequential implementations, enabling the solver to process larger diagrams within the same time budget.

The paper is organized as follows. Section 2 briefly recalls the terminology for branch & bound frameworks using decision diagrams. Section 3 outlines a decoupled architecture for such a solver while section 4 focuses on its implementation on a GPU architecture. Section 5 presents empirical results, and section 6 concludes the paper.

2 Decision Diagram-Based Branch-and-Bound

We first provide a brief background on decision diagram-based branch-and-bound [1], using the terminology of the CODD solver [10]. In this approach, a combinatorial optimization problem is modeled as a dynamic program defined by a set of states $\mathcal{S}$, an initial state $r \in \mathcal{S}$, and transitions between states generated by decision labels from a set U. The label generation function $\lambda(s) \subseteq U$ specifies the labels applicable at state s. The transition function $\tau : \mathcal{S} \times U \to \mathcal{S}$ maps a state and label to a successor state, and the transition cost function $c : \mathcal{S} \times U \to \mathbb{R}$ defines the cost of applying a label. A feasible solution corresponds to a sequence of transitions from the initial state to a terminal state.

The dynamic program induces a decision diagram, a directed acyclic graph whose nodes correspond to states and whose arcs represent transitions defined by τ. The diagram is constructed layer by layer, where each layer contains states reachable after the same number of decisions. Expanding a layer generates successor states using the transition function. Doing so, each state s will have an associated (implicit) cost function $g(s)$ representing the value of a shortest r–s path (assuming minimization). In addition, the function $h(s)$ represents a local bound at state s, which the user may specify. These are combined in the function $f(s) = g(s) + h(s)$ representing the scoring function of s (an over-approximation of the true cost of completing a solution through s).

Because the state space may grow exponentially, CODD constructs restricted and relaxed diagrams of bounded size by limiting the width of any layer to some maximum value w.

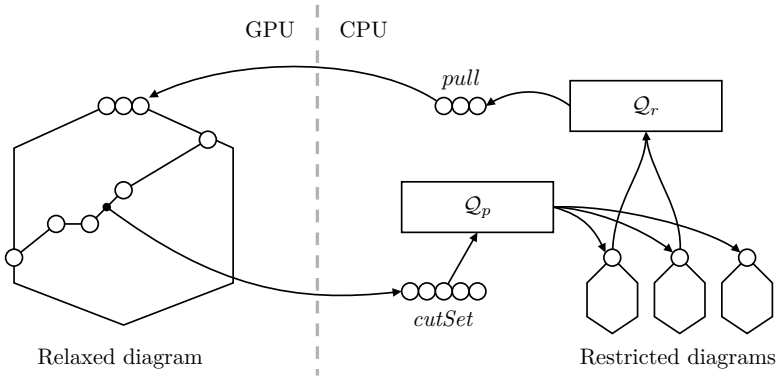

Figure 1 A schematic overview of the decoupled CODD architecture that separates the compilation of relaxed and restricted diagrams on different queues. While this architecture can, in principle, be implemented sequentially on a single CPU, our approach compiles each relaxed diagram in parallel on a GPU and the restricted diagrams in parallel on a multi-threaded CPU.

Restricted diagrams retain a subset of states and therefore represent a subset of feasible solutions, yielding primal bounds. For relaxed diagrams, a state merging operator $m : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ is specified, whose purpose is to merge non-equivalent states and return a new state whose definition encompasses all feasible completions that are obtainable from the merged states individually, and potentially more. During compilation, when a layer exceeds the maximum width, the merge operator is applied until the maximum width is respected. The resulting diagram represents a superset of all feasible solutions, and optimizing over it yields a dual bound [1].

These diagrams are used within a branch-and-bound procedure that maintains a queue of states to explore. At each step, the solver selects a state from the queue and compiles both a relaxed and a restricted decision diagram rooted at that state. The restricted diagram attempts to produce a feasible solution (primal bound), while the relaxed diagram computes a dual bound and identifies an exact cut set of states (separating the exact states from the relaxed states) that are inserted back into the search queue, if they can improve the incumbent solution. CODD may also apply dominance rules during compilation to discard states that cannot lead to improving solutions.

Prior work on parallelizing decision diagram–based optimization focused mainly on parallelizing the search process rather than the construction of the diagrams themselves [2]. In particular, Ddo distributes branch-and-bound subproblems across multiple CPU workers, enabling parallel exploration of the search tree while the compilation of individual decision diagrams remains sequential [4]. Related work in the domain-independent dynamic programming (DIDP) framework follows a different state-based search paradigm, implementing algorithms such as parallel Complete Anytime Beam Search (CABS) [12] using multiple CPU threads [8]. These approaches exploit parallelism at the search level. Still, the core operation of expanding states during decision diagram compilation remains largely sequential, despite the inherent layer-by-layer structure that exposes substantial parallelism.

3 Decoupled Architecture of the Branch-and-Bound Search

We start by presenting a decoupled architecture for the search process that reflects the high-level structure of our approach. We provide a visualization in Fig. 1. Our main goal is to separate the compilation of relaxed and restricted decision diagrams, coordinating them via

Function: DecBnB$(r, p, w_x, w_r) \to s^*$
1 $s^* \leftarrow \bot$ // No solution found yet
2 $\mathcal{Q}_r \leftarrow \varnothing$
3 $\mathcal{Q}_p \leftarrow \{r\}$ // To gain an initial primal bound
4 **while** $\mathcal{Q}_r \neq \varnothing \wedge \mathcal{Q}_p \neq \varnothing$ **do**
5 $\quad \langle s^*, \mathcal{Q}_r, \mathcal{Q}_p \rangle \leftarrow prune(s^*, \mathcal{Q}_r, \mathcal{Q}_p, w_r)$ // Processes and empties $\mathcal{Q}_p$
6 $\quad \langle s^*, \mathcal{Q}_r, \mathcal{Q}_p \rangle \leftarrow refine(s^*, Q_r, p, w_x)$ // Processes and empties $\mathcal{Q}_r$
return: s^*

Algorithm 1 Sequential decoupled branch-and-bound search (high-level specification).

two queues of search states, in a manner amenable to parallel GPU and CPU computations. This is done by maintaining a queue of 'ready' search states $\mathcal{Q}_r$, from which subsets of states are pulled for joint compilation as a relaxed diagram. We will ensure that the pulled nodes are all from the same layer (the distance from the root), which allows potential simultaneous parallel compilation of a single relaxed diagram from this set. Upon completion, we identify the exact cut set and place it in a 'pending' queue $\mathcal{Q}_p$. All states in the pending queue are used to compile restricted diagrams. Only those pending states for which the restriction does not yield an exact diagram are returned to the queue of ready nodes for further processing.

3.1 High-Level Specification

We formalize this process in Algorithm 1, where we assume w.l.o.g. that the problem is a minimization problem. The algorithm presents a sequential specification of the decoupled branch-and-bound algorithm. Its input is a root state r, and parameters p, w_x, and w_r bounding, respectively, the size of the set to be pulled from $\mathcal{Q}_r$ and the maximum widths for the relaxed and restricted diagrams. Its main data structures are two queues, $\mathcal{Q}_p$ and $\mathcal{Q}_r$, that hold pending and ready states, respectively. The pending queue is initialized with the root state r while the ready queue is initially empty. The while loop (lines 4-8) first calls the function *prune* that processes all pending states $s \in \mathcal{Q}_p$ by building a restricted diagram of maximum width w_r rooted at s. It returns a potentially new incumbent solution, as well as the subset of states $\mathcal{Q}_p$ that can potentially improve the best solution, and are added to Q_r. The details of *prune* are given in Section 3.2.

The states in the ready queue $\mathcal{Q}_r$ are processed by the function *refine* to build relaxed diagrams. The compilation will consider a subset of states of size at most p. The details of this function are given in Section 3.3.

The computation ends when both queues are empty, indicating that all states have been processed (either explicitly or implicitly via pruning). Observe how starting with $\mathcal{Q}_p = \{r\}$ gives a chance to the restricted diagram in *prune* to produce a first primal bound that can benefit even the very first run of the relaxed diagram.

3.2 Pruning the Pending Queue With Restricted Diagrams

Algorithm 2 specifies the *prune* function, which processes states in the pending queue with a restricted diagram. Its purpose is to filter the pending queue to safely discard states that cannot improve upon the incumbent s^*. Its output is an augmentation of the ready queue, namely the set $\mathcal{Q}_r \cup K$, where $K \subseteq \mathcal{Q}_p$. Lines 2-8 loop through the states in $\mathcal{Q}_p$. For each state n, it constructs a restricted diagram rooted at n of width at most w_r. This diagram delivers a solution σ (if one is found) alongside a Boolean flag *exact* indicating whether any

Function: prune(s^*, $\mathcal{Q}_r$, $\mathcal{Q}_p$: pending states, $w_r : \mathbb{N}$) $\to \langle s^*, \mathcal{Q}_r, \mathcal{Q}_p \rangle$

1 $K = \varnothing$ // Subset of states in $\mathcal{Q}_p$
2 **for** $n \in \mathcal{Q}_p$ **do**
3 $\quad \langle \sigma, exact \rangle \leftarrow restrict(n, w_r)$
4 $\quad$ **if** $\sigma \neq \bot \wedge f(\sigma) < f(s^*)$ **then**
5 $\quad\quad s^* \leftarrow \sigma$
6 $\quad$ **if** $\neg exact$ **then**
7 $\quad\quad K \leftarrow K \cup \{n\}$
8 $\quad \mathcal{Q}_p \leftarrow \mathcal{Q}_p \setminus \{n\}$

return: $\langle s^*, \mathcal{Q}_r \cup K, \mathcal{Q}_p \rangle$

Algorithm 2 The prune function (high-level sequential specification). Here, σ represents the best exact solution (if one is found).

Function: refine(s, Q_r, p, w_x) $\to \langle s^*, Rem, Cut \rangle$

1 $Rem \leftarrow \varnothing$ // Remaining nodes in Q_r to be returned
2 $Cut \leftarrow \varnothing$ // Cut set to be returned
3 **if** $\mathcal{Q}_r \neq \varnothing$ **then**
4 $\quad n_b \leftarrow \text{argmin}_{n \in Q_r} f(n)$ // Best node in queue
5 $\quad pull \leftarrow \{n_1 \cdots n_k\}$ s.t. $n_i \in Q_r \wedge f(n_i) = f(n_b) \wedge \mathcal{L}(n_i) = \mathcal{L}(n_b) \wedge k \leq p$
6 $\quad Rem \leftarrow \mathcal{Q}_r \setminus pull$
7 $\quad \langle cutSet, \tilde{\sigma}, \sigma \rangle \leftarrow \text{relaxed}(pull, w_x)$
8 $\quad$ **if** $\sigma \neq \bot \wedge f(\sigma) < f(s^*)$ **then**
9 $\quad\quad s^* \leftarrow \sigma$
10 $\quad$ **if** $\tilde{\sigma} \neq \bot$ **then**
11 $\quad\quad$ **if** $f(\tilde{\sigma}) \leq f(s^*) \wedge \tilde{\sigma} \neq \sigma$ **then**
12 $\quad\quad\quad Cut \leftarrow \{n \in cutSet \mid f(n) < f(s^*)\}$

return: $\langle s^*, Rem, Cut \rangle$

Algorithm 3 The refine function (high-level sequential specification). Here, $\tilde{\sigma}$ represents the best found solution (relaxed or exact), and σ represents the best exact solution (if one is found).

diagram states were ever discarded. If σ exists and improves the incumbent, line 5 updates the incumbent. In case the restricted diagram is not exact, state n is considered *ready* and added to the set K of states to be considered for further refinement via the function *refine* in Algorithm 1. We note that the function *restrict* follows the standard specification of the CODD library, and we therefore omit the details of that function in this paper.

3.3 Refining the Ready Queue With Relaxed Diagrams

Algorithm 3 specifies the *refine* function, which processes states in the ready queue by compiling relaxed diagrams. This function deviates from standard compilation methods for relaxed decision diagrams by processing multiple states from the same layer simultaneously to build a single relaxed diagram.

Specifically, line 3 in Algorithm 3 first selects a node in the queue with the current best objective value. Then, in line 4, we construct a set *pull* of nodes from $\mathcal{Q}_r$ with the same objective value and from the same layer (the layer of node n is indicated by $\mathcal{L}(n)$). In addition, for computational purposes, we limit the size of this set to be at most p. When

```
   Function: ParDecBnB(r: state, p : ℕ,w_x: ℕ,w_r: ℕ) → s*
1  s* ← ⊥                                  // Shared and accessed in mutual exclusion
2  Q_r ← {r}                               // Shared and accessed in mutual exclusion
3  Q_p ← ∅                                 // Shared and accessed in mutual exclusion
4  while Q_r ≠ ∅ ∧ Q_p ≠ ∅ do
5      do in parallel                                              // t CPU threads
6          parall t ∈ {0, 1, ..., T − 2} do
7              ⟨s*, Q_r, Q_p⟩ ← prune(s*, Q_r, Q_p, w_r);
8          ⟨s*, Q_r, Q_p⟩ ← refine(s*, Q_r, p, w_x)
   return: s*
```

Algorithm 4 Parallel decoupled branch-and-bound search (high-level specification).

$p = 1$, the condition simplifies to selecting the best state from $\mathcal{Q}_r$ according to f. This is the policy implemented in solvers like DDO [3] and CODD [10]. When $p > 1$, the relaxed diagram is not rooted at a single state, but at a collection of states all sharing the same objective score f and distance from the root.

The relaxed diagram creates, for every state s in the parent layer, the set of children reachable via the transition function $\tau(s, \ell)$ for all valid labels $\ell \in \lambda(s)$. If the number of valid children (unique and not dominated) exceeds the width restriction w_x, the merging operator is applied until the maximum width w_x is reached. The limit w_x is *shared* by all the descendants of the (up to) p roots in the layer. Naturally, one would expect p to be much smaller than w_x, as an increase in p weakens the width allowance. Observe how this is *dynamic*, namely, the roots of the diagrams are not guaranteed to be apportioned the same fraction of the width $\frac{w_x}{p}$. Instead, the merging of states within a layer attempts to preserve the globally best (according to g) $w_x - 1$ states in the layer and merge all remaining ones into the last open slot of the layer.[1]

The sequential version of the *relaxed* function uses the standard specification of the CODD library, except that it is initialized with multiple root nodes instead of one. We present a GPU-parallel version of the *relaxed* function as Algorithm 5 in the next section.

4 Parallelizing the Decoupled Branch-and-Bound Search

We next introduce the parallel version of the decoupled branch-and-bound search, which involves parallelization on the CPU (for compiling the restricted diagrams) and massive parallelization on a GPU (for compiling the relaxed diagrams).

4.1 Parallelizing the Top-Level on the CPU

The parallel implementation is obtained by adapting Algorithm 1 and is presented as Algorithm 4. The first difference is that the core data structures, s^*, $\mathcal{Q}_r$, and $\mathcal{Q}_p$, are shared across t threads on the CPU. Further, the ready and pending queues $\mathcal{Q}_r$ and $\mathcal{Q}_p$ are now producer-consumer queues dynamically updated throughout the search process. Lines 5-8 define t independent threads that synchronize through those shared structures. Lines 6-7 rely on $t - 1$ threads to concurrently consume pending states from $\mathcal{Q}_p$, process them by compiling

[1] This is unrelated to the priority queues $\mathcal{Q}_r$ and $\mathcal{Q}_p$ and purely focuses on influencing which states get merged within a layer.

restricted diagrams, and add the surviving ready states directly into $\mathcal{Q}_r$. Line 8 runs the last thread that consumes nodes from the relaxed queue $\mathcal{Q}_r$ and refines them. The refinement, as before, can update s^* and produce new pending states into $\mathcal{Q}_p$. Specifically, those states are from the cut set and still have the potential to improve the incumbent.

All the parallelism in Algorithm 4 occurs at the CPU level, and can be tuned with the t parameter. While *prune* is essential, its parallelization is intended to maximize GPU utilization. Reducing t too much would likely *starve* the refinement thread from work as it would block in case $\mathcal{Q}_r = \varnothing$. Likewise, too many refinement workers would suffer from high contention on $\mathcal{Q}_p$ without much (if any) impact unless the width w_x is scaled up accordingly.

4.2 Programming a GPU

Before presenting the GPU implementation, it is opportune to provide some background on GPU programming. As this cannot be a comprehensive treatment, we refer the interested reader to [6] for further details.

While often associated with graphics rendering and vectorized operations, modern GPUs are, in fact, general-purpose parallel processors. At a high level, a GPU can be seen as a collection of multi-core processors, each paired with dedicated memory, that excels at parallelizing independent loop iterations. Loops are central to many computations, including matrix and vector operations as well as state expansion. Computations that rely on non-trivial data structures or are not naturally loop-based must be reformulated, which is sometimes non-trivial and not always possible.

This work leverages CUDA, the de facto standard that allows developers to easily express parallel computations as kernels (i.e., C/C++ functions). Each thread executes the same kernel and uses its unique index to identify the data it operates on. While allowing massive parallelization, this paradigm comes with its own challenges:

Memory latency Memory access is generally much slower than on the CPU, making pointer-chasing and pointer-based data structures impractical. Access patterns should exploit locality to maximize cache usage, which motivates the array-based design adopted throughout.

Thread synchronization Concurrent access to shared data must be carefully managed. For example, parallel insertion into a shared vector requires mutual exclusion, which is achieved with a lock-free mechanism based on atomic operations.

Memory allocation The GPU cannot efficiently access system RAM, making it necessary to pre-allocate all required memory. The size of the temporary arrays is determined by an upper bound that accounts for the width w_x and the maximum number of labeled transition edges leaving a state. For moderately sized states, a GPU with 16GB of memory can support widths on the order of millions of states.

Overhead There is a *non-negligible* overhead for setting up kernels and copying data back and forth. GPUs can only shine if they are tasked to accelerate enough work to overcome this cost. Empirically, workloads that can be completed sequentially in a handful of seconds are not worth doing on a GPU. The benefits only arise once the load requires a significant amount of time sequentially.

4.3 Parallelizing Relaxed Diagram Compilation on the GPU

The main component of the refine function is the compilation of relaxed diagrams through the function *relaxed*. Algorithm 5 shows the GPU-parallelized version of the *relaxed* function. It is invoked on a *set* of states (*pull*) and with a desired maximum width w_x. The construction

Function: relaxed($pull$, w_x) $\rightarrow \langle cutSet, \tilde{\sigma}, \sigma \rangle$

1 $cutSet \leftarrow [\,]$
2 $\tilde{\sigma} \leftarrow \bot$
3 $\sigma \leftarrow \bot$
4 $P \leftarrow [\,]$ // Parents list
5 $S \leftarrow pull$ // Successors list
6 $A \leftarrow [\bot, \ldots, \bot]$ // Ancestor in cut set
7 **while** $S \neq \varnothing \wedge \tilde{\sigma} = \bot$ **do**
8 $(P, S) \leftarrow (S, P)$ // Swap parent and children
9 $S \leftarrow \text{calcSuccessorsGPU}(P, S)$ // Asynchronous calls
10 $S \leftarrow \text{sortByGPU}(S, g(\cdot))$
11 $(A, cutSet) \leftarrow \text{updateCutsetGPU}(P, S, A, w_x, cutSet)$
12 $S \leftarrow \text{mergeSuffixGPU}(S, w_x)$ // Reduce S to w_x states
13 $\tilde{\sigma} \leftarrow \text{findIfGPU}(S, \text{isTarget}(\cdot))$ // Returns $\bot$ if no targets
14 waitGPU() // Synchronization
15 $(\tilde{\sigma}, \sigma) \leftarrow \text{findBestIfGPU}(S, g(\cdot), \text{isTarget}(\cdot))$
16 $cutSet \leftarrow \text{updateBoundsGPU}(cutSet, g(\tilde{\sigma}))$ // $g(\tilde{\sigma})$ is a valid relaxation
17 waitGPU()
return: $\langle cutSet, \tilde{\sigma}, \sigma \rangle$

Algorithm 5 GPU-accelerated relaxed diagram compilation (high-level specification)

of the relaxed diagram proceeds in layers, generating the successors of states in P (the parents) into a new layer S. Since the code does not retain all layers or edges, the cut set must be constructed on the fly as the parents of relaxed states are identified. The A vector is used to report, for each state in P, whether the state itself or one of its ancestors is already part of the cut set. The initialization in line 6 is equivalent to setting $cutSet$ to $\varnothing$. The main loop starts by swapping the two buffers, with the children of iteration i becoming the parents (P) of iteration $i+1$. It then invokes a sequence of kernels on the GPU. Specifically, line 9 generates the successors of states from P in vector S. Line 10 sorts the states in S by g (the exact cost of the path from the root to the state). Line 11 updates the vector A and greedily adds parents of nodes to be relaxed into $cutSet$. Line 12 *collapses* S in parallel (and on the GPU) by retaining the best $w_x - 1$ states untouched and merging everything else in a catch-all state that will occupy the last slot in the vector S. Finally, line 13 identifies the relaxed state (if any) in layer S that is a parent of the sink (a target). Line 14 is a barrier that forces the CPU to wait for all GPU kernels to finish before assessing the termination condition to either exit or proceed to the next layer. When the computation ends, line 15 populates $\tilde{\sigma}$ as well as σ from the penultimate layer. Note that σ may be undefined in case *all states* were not exact. Line 16 attempts to tighten the dual bound of each state $s \in cutSet$, considering $g(\tilde{\sigma}) - g(s)$. The absence of edges precludes an exact bound calculation from the sink. The code ends with a final barrier before returning the accumulated $cutSet$, the best relaxed and possibly the best exact final state.

Successor generation on GPU

Algorithm 6 presents a high-level specification of the core kernel in charge of generating the direct successors of states in the parent layer P. Recall from Section 4.2 that the code manipulates vectors whose sizes are upper bounded based on the maximum branching factor

```
   Function: calcSuccessorsGPU(P: parents, S: successors) → (S: successors)
1  S ← []                                                  // Initialize successor array
2  T_0 ← []                                                // Feasible successors of P
3  parall s ∈ P do
4      parall ℓ ∈ λ(s) do
5          T_0 ← T_0 ‖ [τ(s, ℓ)]                          // Array concatenation (thread-safe)
6  T_1 ← []                                                // Filter sub-optimal states
7  parall s ∈ T_0 do
8      if f(s) < g(s*) then
9          T_1 ← T_1 ‖ [s]
10 H ← parSortBy(T_1, h ∘ π)                               // Array of states sorted by hash
11 R ← [⊥, ..., ⊥]                                         // State dominance flags
12 parall i ∈ {0, ..., |H| − 1} do
13     forall j > i s.t. hash(π(H[i])) = hash(π(H[j])) do
14         if H[i] = H[j] ∨ dom(H[i], H[j]) then
15             R[j] ← ⊤                                    // State H[j] is dominated
16         else if dom(H[j], H[i]) then
17             R[i] ← ⊤                                    // State H[i] is dominated
18 parall i ∈ {0, ..., |H| − 1} do
19     if R[i] = ⊥ then
20         S ← S ‖ [H[i]]
   return: S
```

Algorithm 6 GPU-based successor generation and pruning

of a state. Note how the nested parallel loops in lines 3-5 generate all the successors with the generic function τ (for taking the transition out of a state under a label ℓ) and λ (to generate the set of labels leaving the source state). The $\|$ operator is a thread-safe addition to the end of the vector. Lines 7-9 filter (in parallel) the states that still have the potential of improving upon the incumbent in a new vector T_1, which is sorted on line 10 by hash value to identify identical states (for de-duplication) and dominance checking. For computational benefits, we use a projection operator $\pi(s)$ that returns only the state attributes of s that are relevant to the dominance rule. Lines 11-17 populate a vector R with flags, each indicating whether the corresponding state is dominated. Each entry of H is processed by a GPU thread that scans the sequence of states sharing the same hash of the projection (by π) on the attributes required to be in the same equality/dominance class. Lines 18-20 cover the last parallel loop that pulls the states not flagged in R into the resulting successors T_2 (again, via a thread-safe vector add).

Cut set computation on the GPU

The last code fragment worth detailing appears in Algorithm 7 and its task is to update *cutSet* from the set of parents P when one identifies the states in S that are going to be merged. This computation also occurs in parallel on the GPU and is conditioned on having an actual width that exceeds the limit w_x (i.e., $|S| > w_x$). The parallel loop spanning lines 3-6 identifies the parent of a state i to be merged (whose index is larger than or equal to $w_x - 1$) and flags that parent for addition into the *cutSet* provided that it is not a descendant of a node already in *cutSet*. The loop spanning lines 7-9 adds the flagged parents to *cutSet*. Finally, lines 11-14 compute a revised flag vector T_1 that includes whoever was already

Function: updateCutsetGPU(P: parents, S: successors, A: ancestor in cut set, w_x: $\mathbb{N}$, *cutSet*: cut set) $\rightarrow$ (A: ancestor in cut set, *cutSet*: cut set)

1 **if** $|S| > w_x$ **then**
2 $T_0 \leftarrow [\bot, \ldots, \bot]$ `// Temporary of size` w_x
3 **parall** $i \in \{w_x - 1, \ldots, |S| - 1\}$ **do** `// For the states going to be merged`
4 $j \leftarrow$ indexOfParent($S[i]$)
5 **if** $A[j] = \bot$ **then**
6 $T_0[j] \leftarrow \top$ `// Flag parent to be saved`
7 **parall** $i \in \{0, \ldots, |P| - 1\}$ **do**
8 **if** $T_0[i] = \top$ **then**
9 $cutSet \leftarrow cutSet \cup \{P[i]\}$ `// Lock-free implementation`
10 $T_1 \leftarrow [\bot, \ldots, \bot]$ `// Temporary of size` w_x
11 **parall** $i \in \{0, \ldots, w_x - 1\}$ **do** `// For the states not merged`
12 $j \leftarrow$ indexOfParent($S[i]$)
13 **if** $A[j] = \top \vee T_0[j] = \top$ **then**
14 $T_1[i] \leftarrow \top$ `// Flag ancestor has been saved`
15 $A \leftarrow T_1$

return: (A, *cutSet*)

Algorithm 7 GPU-based update of the cut set

flagged in the input A or that just got flagged in T_0. Atomic operations must be used to update *cutSet*.

5 Experimental Results

We compare our approach against state-of-the-art solvers for dynamic programming models on standard benchmarks from the literature. The solvers considered are:

CODD a sequential decision diagram-based solver [10].

DDO a decision diagram-based solver offering both a sequential and parallel search [4].

DIDP-RS a state-based solver offering sequential A*, CABS [7], and parallel CABS [8][2]. RPID [9] does not provide a parallel solver.

GPU-DD-BnB the approach presented in this paper.

Both CODD and parallel DDO were tested with widths $32, 64, \ldots, 1024$, and we report the best result for each instance. Parallel DDO and DIDP-RS were configured to use all available cores (96). The parameters of GPU-DD-BnB are $w_x = 200{,}000$, $w_r = 5000$, $p = 5000$ and $t = 32$, which is the number of "pruners" thinning out $\mathcal{Q}_p$.

We used three benchmarks from the literature: Knapsack, Maximum Independent Set (MISP [5]), and Golomb Ruler. Each experiment pursues a different objective:

Knapsack was chosen as a sanity check. All benchmarks can be solved sequentially quite fast (in the order of a second or two), provided that enough width/memory is available. The goal is to determine how much overhead one should expect when starting the GPU machinery.

MISP has models available across all solvers. One important note is that DDO offers a model with a strong global variable selection heuristic, which significantly affects runtime. We turned it off to fairly assess the benefits of parallelization.

[2] See the `*_dypdl.rs` files in https://github.com/Kurorororo/didp-rust-models.

Instance	CODD-BnB			DIDP-RS-A*			DIDP-RS-CABS			GPU-DD-BnB		
	T	N	M	T	N	M	T	N	M	T	N	M
knapPI_1_10000_1000_1	6.1	0.0	11.1	–	–	–	137.3	17.6	0.1	9.4	0.0	0.1
knapPI_1_5000_1000_1	2.2	0.0	2.7	29.7	1.5	0.2	12.2	4.1	0.0	5.4	0.0	0.1
knapPI_2_10000_1000_1	8.1	0.0	14.4	132.8	4.7	0.5	94.0	14.7	0.0	9.4	0.0	0.1
knapPI_2_5000_1000_1	2.9	0.0	4.0	7.6	0.9	0.1	7.5	3.2	0.0	5.4	0.0	0.1
knapPI_3_10000_1000_1	4.0	0.0	7.5	–	–	–	294.3	31.0	0.1	9.5	0.0	0.1
knapPI_3_2000_1000_1	39.8	0.0	8.9	1.2	0.5	0.1	1.6	2.2	0.0	2.5	0.0	0.1
knapPI_3_5000_1000_1	103.2	0.0	41.1	45.4	2.3	0.2	30.4	9.6	0.0	5.4	0.0	0.1

Table 1 Results summary for Knapsack.

Golomb Ruler is a challenging benchmark whose cost grows quite quickly with instance size. Traditional (and sequential) CP solvers struggle with instances of size 13, CODD reaches 14, and the GPU implementation reaches 16.

The experiments were conducted on an AMD Ryzen Threadripper PRO 7995WX[3], 768 GB of RAM, and a NVIDIA RTX 6000 Ada Generation[4]. The system runs Ubuntu 24.04 LTS and uses CUDA 12.5, GCC 13.3, and Rust 1.90. To ensure a reasonable benchmark time, we used a 5-minute timeout.

5.1 Knapsack

To ensure a compact and insightful presentation, Table 1 includes only solved instances for which at least one solver needed more than five seconds. For each solver, we report the search time T in seconds, the explored nodes N in millions, and the peak CPU memory usage M in GB. With the default parameters, the total GPU memory requirement is 2.49 GB. Note that this modest GPU memory footprint indicates one could use consumer-grade NVIDIA cards (a "gaming" card like the RTX 5070 Ti features 8 to 16GB of VRAM). What is apparent is that knapsack is reasonably easy for sequential solvers (i.e., CODD). Indeed, the GPU implementation can hardly compete. The run times of 9.4s and 5.4s are directly correlated with the number of items in the knapsack (5000 or 10,000). Clearly, the GPU closes all the instances at the root node, and the time is directly related to the number of items (layers). The only clear wins for the GPU are on `knapPI_3_xxxx`, namely the hard correlated instances for which the additional width offered by the GPU is beneficial. It is particularly encouraging to note that the overhead cost on an "easy" problem is quite low when one compares the best run times from any solver against those of the GPU implementation. They never exceed a 2x slowdown and reach a 19x speedup at best.

5.2 MISP

Table 2 conveys the results for maximum independent set (again, focusing on significant instances). With the default parameters, the total GPU memory requirement is 903 MB. Dashes in the table indicate a timeout. The GPU implementation never takes less than a second to complete due to setup overhead. Nonetheless, for hard instances, the speedup can be significant, and more instances can be closed within the time limit (e.g., `san1000` can be

[3] 96 physical Cores at 5.1 GHz, 384 MB of L3 Cache
[4] 18176 CUDA Cores at 2.5 GHz, 96 MB of L2 Cache, 48 GB of VRAM.

Instance	CODD-BnB			DDO-Par			DIDP-RS-Par			GPU-DD-BnB		
	T	N	M	T	N	M	T	N	M	T	N	M
brock200_1	–	–	–	60.4	1.6	5.7	–	–	–	21.8	2.1	1
brock200_2	0.9	0.0	0	0.5	0.0	0.5	1.4	11.9	0	2.0	0.0	0
brock200_3	7.5	0.0	0	1.4	0.0	1.8	7.3	80.4	0	3.0	0.0	0
brock200_4	23.3	0.1	0	2.6	0.1	2.5	33.0	382.7	3	5.0	0.1	0
hamming10-2	0.2	0.0	0	5.8	0.0	0.5	–	–	–	14.2	0.0	4
hamming8-2	0.0	0.0	0	0.4	0.0	0.1	–	–	–	4.1	0.0	1
hamming8-4	5.0	0.0	0	1.3	0.0	2.4	82.1	936.1	9	3.0	0.0	0
keller4	7.6	0.0	0	0.7	0.0	1.1	4.1	41.5	0	2.0	0.0	0
p_hat1000-1	–	–	–	18.4	0.2	21.8	98.9	1,039.8	5	15.0	0.2	0
p_hat1500-1	–	–	–	116.1	1.0	50.0	–	–	–	79.1	1.3	1
p_hat300-2	95.7	0.3	0	8.9	0.1	4.6	128.7	1,429.4	6	9.1	0.3	0
p_hat500-1	6.7	0.0	0	2.0	0.0	1.7	3.5	31.1	0	4.0	0.0	0
p_hat700-1	31.1	0.0	0	5.5	0.0	6.4	16.3	164.5	1	6.0	0.0	0
san1000	–	–	–	–	–	–	–	–	–	89.0	0.5	1
san200_0.7_1	–	–	–	12.1	0.1	3.6	49.1	519.9	3	3.0	0.0	0
san200_0.7_2	–	–	–	90.8	1.6	5.6	32.7	338.2	2	33.0	0.5	1
san400_0.5_1	28.4	0.0	1	2.7	0.0	2.6	3.3	29.3	0	4.0	0.0	0
sanr200_0.7	88.9	0.4	0	9.4	0.3	3.4	104.2	1,187.1	9	7.0	0.4	0
sanr400_0.5	125.3	0.4	1	8.8	0.2	6.9	71.5	816.8	5	10.0	0.2	0

Table 2 Results summary for MISP.

Instance	CODD-BnB			DDO-Par			DIDP-RS-Par			GPU-DD-BnB		
	T	N	M	T	N	M	T	N	M	T	N	M
10	0.3	0.0	0.0	1.5	0.0	3.1	11.1	5.0	0.4	1.6	0.0	0.2
11	6.0	0.1	0.1	9.2	0.1	6.9	125.1	84.9	8.2	1.5	0.0	0.3
12	27.4	0.2	0.3	54.1	1.2	10.6	956.5	584.2	44.7	1.5	0.2	0.3
13	317.6	3.6	1.9	1,586.8	8.4	18.3	–	–	–	9.5	1.6	1.4
14	2,420.3	19.3	11.3	–	–	–	–	–	–	41.6	7.4	2.8
15	–	–	–	–	–	–	–	–	–	467.2	0.4	31.9
16	–	–	–	–	–	–	–	–	–	3,152.4	1.6	75.0

Table 3 Solving results for Golomb Ruler.

closed). The harder the instance is for sequential solvers, the greater the speedup on the GPU, ranging from 1x to 12x compared to the best sequential.

The comparison against parallel solvers (that use all 96 physical CPU cores) is equally informative. For instances that can be closed, the speedup is, of course, more modest (1x to 3x). But the GPU implementation can close everything in this set under 300 seconds of runtime. One should note that CUDA-capable GPUs with 2-4 GB of VRAM are considered commodity hardware and widely available. In contrast, CPUs with hundreds of cores (like the CPU used here) are server-class machines (a commodity desktop or laptop has of the order of 16 cores). Interestingly, using a large width on easy instances is counterproductive, as in `hamming10-2`, whose runtime can be reduced to 2s by using $w_r = 2000$ (see Section 5.4).

5.3 Golomb Ruler

Table 3 reports the results for instances from sizes 10 to 16. Due to the hardness of the benchmark, we used a 1-hour timeout. With the default parameters, the total GPU memory requirement is 20.5 GB. Recall that DIDP-RS increases width automatically via doubling.

Golomb(10) is "easy" for the fastest sequential solver (CODD in 0.3s), whereas the GPU version takes 1.6s to complete the same task. This illustrates the limit of GPUs. The setup overhead can easily reach seconds of runtime and negate all benefits. Evidence of this can be seen in 3 instances (Golomb(10), Golomb(11), Golomb(12)) where the setup entirely dominates the runtime. Yet, starting with instance Golomb(11), benefits are apparent and only grow with instance size. The best sequential solver does not get past Golomb(14) (it times out on 15 and 16). The amount of memory reported in the table is in gigabytes of RAM on the system, and it remains reasonable. To push beyond size 14, instances 15 and 16 were run with $w_x = 350{,}000$, $w_r = 25{,}000$, $pull = 1000$, and $t = 96$. With these parameters, the total GPU memory requirement is 35.93 GB. This change *alone* enables the resolution (with proof) of Golomb(15) in 467 seconds and Golomb(16) in 3152 seconds.

5.4 Sensitivity of the Parameters

The GPU implementation relies on 3 parameters: w_x, the width of the relaxed diagrams on the GPU, w_r, the width of the restricted diagrams on the CPU, and p, the size of the pull set when loading work onto the GPU. The baseline for the following analysis is $w_x = 500{,}000$, $w_r = 4096$, $p = 500$ and $t = 16$ workers are used to prune the pending queue $\mathcal{Q}_p$.

Table 4 reports on the sensitivity of all three parameters when the other two are held to their baseline values. One representative instance was selected (`brock200_1`) because its run times are long enough to capture behavioral changes.

Sensitivity to w_x. Widths are based on a geometric progression starting at 128 and doubling until just north of 2,000,000 states. The earliest delivery of the optimal solution occurs at $w_x = 524{,}288$ in 4.92 seconds and also delivers the shortest optimality proof in 33 seconds. Observe how the proof time (as well as the time to best) first decreases as the width increases until a threshold, beyond which additional width only introduces overhead with no further benefits. Namely, it corresponds to computing more states in all the layers that are inferior to the incumbent and do not provide any assistance in the resolution. The memory column indicates the amount of GPU memory necessary to process the restricted diagrams at the corresponding width.

Sensitivity to w_r. The CPU width used for the restricted diagrams built by *prune* also follows a geometric progression (starting at 32, doubling, and ending at 262,144). While this component is essential to deliver primal solutions that enable optimality pruning, the effects of width increases are more subdued (still a twofold change between worst and best run times). Memory usage here is on the CPU as the restricted diagrams run there (recall that we are using $t = 16$ workers to carry out the pruning in parallel). The sweet spot around 4096 justifies the choice of default value for that parameter.

Sensitivity to p. Once again, p follows a geometric progression starting at 1 and doubling all the way to 262,144. This time, it is readily apparent that p matters. Runtimes are improving steadily up to $p = 512$, where they appear to provide a runtime of 33s for the optimality proof. It is clear that loading enough work onto the GPU in one "go" is critical to performance, as it reduces communication between the CPU and the GPU that requires loading and processing states one at a time.

Increasing p also leads to an increase in the number of branch and bound nodes being explored. The behavior is consistent with weakening the relaxation as the fixed width w_x is now shared by a larger number of roots and induces more merging, leading to more relaxed states entering the branch-and-bound queue. Further increases in p would lead to an increase in the number of nodes in the branch-and-bound queue, potentially overshadowing the gain

Parameter	Size	Opt Time [s]	Proof Time [s]	Nodes	Memory
w_x $(GPU - Relaxed)$	128	14.07	81.01	25,242	584.00 KB
	256	20.03	68.01	18,172	1.14 MB
	512	13.87	62.01	15,508	2.34 MB
	1024	11.32	54.00	11,151	4.65 MB
	2048	11.01	51.00	9,268	9.28 MB
	4096	12.74	49.00	7,946	18.53 MB
	8192	9.32	46.02	6,684	37.05 MB
	16384	11.72	45.01	6,074	74.07 MB
	32768	10.10	42.00	5,132	148.11 MB
	65536	10.87	40.00	4,820	296.19 MB
	131072	11.88	38.00	4,352	592.36 MB
	262144	7.27	37.00	3,357	1.16 GB
	524288	4.92	33.00	2,736	2.31 GB
	1048576	7.21	35.00	2,154	4.63 GB
	2097152	12.94	38.00	1,754	9.26 GB
w_r $(CPU - Restricted)$	32	4.86	70.01	63,448	3.69 MB
	64	4.69	70.01	63,484	5.09 MB
	128	4.83	69.01	61,832	7.91 MB
	256	4.77	64.01	48,361	13.53 MB
	512	4.61	48.00	26,070	24.78 MB
	1024	4.53	36.00	12,978	47.28 MB
	2048	4.76	33.00	6,265	92.28 MB
	4096	4.65	33.00	2,777	182.28 MB
	8192	4.81	35.02	1,136	362.28 MB
	16384	5.06	38.00	516	722.28 MB
	32768	5.79	46.01	184	1.41 GB
	65536	6.58	56.01	63	2.81 GB
	131072	8.54	73.01	36	5.63 GB
	262144	12.56	94.01	11	11.25 GB
p $(PullSize)$	1	4.83	301.03	1,452	–
	2	4.68	301.03	2,102	–
	4	4.59	177.02	2,102	–
	8	4.78	108.01	2,109	–
	16	4.79	72.01	2,202	–
	32	4.78	53.01	2,244	–
	64	4.83	42.00	2,310	–
	128	4.77	36.00	2,379	–
	256	4.66	33.00	2,519	–
	512	4.61	33.00	2,778	–
	1024	4.62	33.00	2,743	–
	2048	4.61	33.00	2,743	–
	4096	4.76	33.00	2,777	–
	8192	4.75	33.00	2,743	–
	16384	4.77	33.00	2,819	–
	32768	4.82	33.00	2,745	–
	65536	4.71	33.00	2,852	–
	131072	4.76	33.00	2,775	–
	262144	4.72	33.00	2,852	–

Table 4 Sensitivity study on `brock200_1`.

in raw power from fully loading the GPU. This last observation is problem-dependent, as the weakening induced by merging depends on the state definition and the merge operator, both of which are problem-specific. While MISP instances appear to be handled adequately with modest p values, Golomb, for instance, benefits from larger and more aggressive values.

5.5 Performance profiles against the relaxed width w_x

Figure 2 plots the optimality gap (over time) on the same `brock200_1` instance. It reports the change for 4 different values of w_x. The purpose is to show how the primal (below the optimum) and the dual (above the optimum) values converge as the width increases.

At $w_x = 4096$ (narrow width for relaxed), the dual starts high and drops slowly, inherently hampering the algorithm's ability to effectively perform optimality pruning. At $w_x = 65{,}536$,

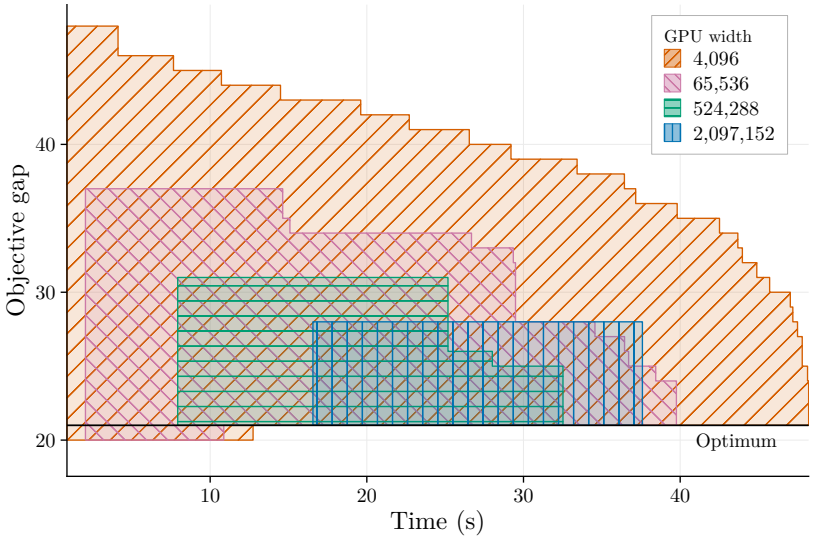

Figure 2 Plot of the optimality gap during search in on `brock200_1`.

the initial dual (obtained fairly early) is stronger and drops faster. That yields a behavior quite similar to the first one, simply on an accelerated timescale (the proof closes the computation just under 40s). At $w_x = 524{,}288$, the behavior changes noticeably. The horizontally striped curve (green) shows that the initial dual is not obtained at the very start but only after 8 seconds. It takes a moment to secure it. Yet, it comes in even stronger. It changes less often as well, but overall drops fast to close the gap and the instance in 33s. By the time $w_x = 2{,}097{,}152$, this behavior is amplified, and it takes quite a while (17s) to deliver an initial (but excellent) dual (blue vertical stripes). Yet, the overhead is such that this does not translate into a net gain, and the dual remains until the instance closes at 38 seconds.

In all cases, note how the primal bound was tight (the part of the surfaces below the optimum line) from the start, and the time delay between finding the optimal and proving it.

Unsurprisingly, this analysis demonstrates that raw power alone is insufficient and that developing a sense of the performance profile is worthwhile for selecting parameters that yield the fastest resolution times.

6 Conclusion

We presented a GPU-accelerated method for compiling relaxed multi-valued decision diagrams within a branch-and-bound framework. This is enabled by a new search architecture separating restricted and relaxed diagram construction, allowing CPU threads to perform pruning while the GPU executes massively parallel diagram compilation. Experimental results on knapsack, maximum independent set, and Golomb ruler benchmarks showed that this approach can significantly reduce the computational cost and enable larger diagram widths than CPU-only approaches. This translates into notable speedups of up to an order of magnitude on difficult instances. We conclude that GPU acceleration of state-based search delivers substantial computational benefits at no additional modeling cost, as users need only formulate their problem as a dynamic programming model.

References

1 D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. *Decision diagrams for optimization*. Springer, 2016. doi:10.1007/978-3-319-42849-9.

2 David Bergman, André A. Ciré, Ashish Sabharwal, Horst Samulowitz, Vijay A. Saraswat, and Willem Jan van Hoeve. Parallel combinatorial optimization with decision diagrams. In Helmut Simonis, editor, *Proceedings of CPAIOR*, volume 8451 of *Lecture Notes in Computer Science*, pages 351–367. Springer, 2014. doi:10.1007/978-3-319-07046-9_25.

3 Vianney Coppé, Xavier Gillard, and Pierre Schaus. Modeling and exploiting dominance rules for discrete optimization with decision diagrams. In Bistra Dilkina, editor, *Proceedings of CPAIOR*, volume 14742 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2024. doi:10.1007/978-3-031-60597-0_15.

4 Xavier Gillard, Pierre Schaus, and Vianney Coppé. Ddo, a generic and efficient framework for mdd-based optimization. In Christian Bessiere, editor, *Proceedings of IJCAI*, pages 5243–5245. ijcai.org, 2020. doi:10.24963/IJCAI.2020/757.

5 Demian Hespe, Sebastian Lamm, and Christian Schorr. Targeted Branching for the Maximum Independent Set Problem. In David Coudert and Emanuele Natale, editors, *Procceedings of SEA 2021*, volume 190 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:21, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2021.17.

6 Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 4 edition, 2022. doi:10.1016/C2020-0-02969-5.

7 Ryo Kuroiwa and J. Christopher Beck. Domain-independent dynamic programming. *CoRR*, abs/2401.13883, 2024. doi:10.48550/ARXIV.2401.13883.

8 Ryo Kuroiwa and J. Christopher Beck. Parallel beam search algorithms for domain-independent dynamic programming. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Procceedings of AAAI*, pages 20743–20750. AAAI Press, 2024. doi:10.1609/AAAI.V38I18.30062.

9 Ryo Kuroiwa and J. Christopher Beck. RPID: rust programmable interface for domain-independent dynamic programming. In Maria Garcia de la Banda, editor, *Proceedings of CP*, volume 340 of *LIPIcs*, pages 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.1609/AAAI.V38I18.30062.

10 Laurent Michel and Willem-Jan van Hoeve. CODD: A decision diagram-based solver for combinatorial optimization. In *Proceedings of ECAI*, volume 392, pages 4240–4247. IOS Press, 2024. doi:10.3233/FAIA240997.

11 Isaac Rudich, Quentin Cappart, and Louis-Martin Rousseau. Improved peel-and-bound: Methods for generating dual bounds with multivalued decision diagrams. *J. Artif. Intell. Res.*, 77:1489–1538, 2023. doi:10.1613/JAIR.1.14607.

12 Weixiong Zhang. Complete anytime beam search. In Jack Mostow and Chuck Rich, editors, *Proceedings AAAI*, pages 425–430. AAAI Press / The MIT Press, 1998. URL: http://www.aaai.org/Library/AAAI/1998/aaai98-060.php.