CP for Bin Packing with Multi-Core and GPUs

Fabio Tardivo [1]
Department of Computer Science, New Mexico State University, Las Cruces, NM, USA

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

Enrico Pontelli
Department of Computer Science, New Mexico State University, Las Cruces, NM, USA

Abstract

The *BinPacking* constraint models the requirements of many logistics, resource allocation, and production scheduling applications. This paper explores new avenues based on the impressive computational power of modern GPUs to propagate the *BinPacking* constraint. This work showcases how the perspective of massive parallelization can lead to novel approaches, such as the use of a portfolio of lower bounds, to enhance the pruning of the *BinPacking* constraints. It delivers insights into the design choices and challenges presented by GPU platform for constraint propagation.

The paper evaluates a GPU-accelerated propagator against both sequential and parallel CPU versions, as well as state-of-the-art approaches. Comparisons across various benchmarks from the literature show strong performances with respect to both CPU versions and the standard pruning approach. When compared to techniques based on Linear Programming, our approach proves valuable for large instances or when spending extensive time to obtain the best possible bound is not convenient.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Massively parallel algorithms

Keywords and phrases Constraint Propagation, Bin Packing, Parallelism, GPU, Lower Bounds

Digital Object Identifier 10.4230/LIPIcs.CP.2024.28

Related Version *Previous Version*: `https://arxiv.org/abs/2402.14821`

Supplementary Material
Software: `https://bitbucket.org/constraint-programming/fzn-minicpp/src/gpu`
archived at `swh:1:dir:b7e3d8e8a2f41b87654cd2e2a81a2622099a8cf0`
Dataset: `https://bitbucket.org/constraint-programming/minicpp-benchmarks` [40]
archived at `swh:1:dir:1dfa371fc284829abddc0a89d1e60251c07d6c84`

Funding *Enrico Pontelli and Fabio Tardivo*: Supported by NSF grants 2151254, 1914635 and by a contract from Wallaroo Inc.
Laurent Michel: Partially supported by Synchrony.

Acknowledgements We would like to thank François Clautiaux, Maxence Delorme, Jürgen Rietz for their feedbacks, and Hadrien Cambazard for providing the Arc-Flow implementation.

---

[1] Corresponding author

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 28; pp. 28:1–28:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The *Bin Packing Problem (BPP)* consists of packing a set of items into the minimal number of bins, each with a fixed capacity. It has a fundamental role in logistics, resource allocation, and production scheduling applications. Because of its relevance, the Bin Packing Problem has been extensively studied over the last decades, both theoretically and practically. We refer interested readers to [10, 36] for a comprehensive review.

The BPP is NP-Hard in the strong sense [17] and it is challenging to solve even for a fixed number of bins [21] or a constant number of different item sizes [19]. Techniques based on Integer Linear Programming (ILP) are highly effective and represent the state-of-the-art for solving the BPP. When the BPP is a component of a larger problem, applying such techniques becomes challenging, and *Constraint Programming (CP)* emerges as a valuable alternative. There, the BPP often appears in its decision version, where the items must be packed into a fixed number of bins.

The decision variant is modeled in CP using the *BinPacking* constraint [38]. Its filtering algorithm employs an *approximated knapsack reasoning* to exclude or commit items to bins, and a *feasibility check* to prune the search if the remaining unpacked items cannot fit in the residual space. The check is performed using a lower bound on the number of bins necessary to pack the items.

The contributions of this paper are as follows:

1. describe a propagator architecture based on *parametric families* of lower bounds and their role in a portfolio setting;
2. demonstrate how the large number of bounds from those parametric families should be computed in parallel to derive the most value. In particular, the paper demonstrates that sampling bounds in a sequential or multi-core implementation is substantively weaker;
3. provide an implementation of a GPU-accelerated portfolio of lower bounds within a constraint propagator of a standard CP solver;
4. deliver an empirical evaluation comparing sequential, multi-threaded, and GPU-accelerated computation of those lower bounds, with other state-of-the-art approaches on different benchmarks.

The rest of the paper is organized as follows. Section 2 contains some general background about Constraint Satisfaction/Optimization Problems and General-Purpose computing on Graphics Processing Units (GPGPU). Section 3 summarizes related works on the *BinPacking* constraint, and on lower bounds for the Bin Packing Problem. Section 4 details the design process and implementation of the *BinPacking* propagator enhanced with the GPU-accelerated portfolio of lower bounds. Section 5 presents the results of our approach and the other techniques in the literature. Finally, Section 6 concludes the paper.

2 Background

2.1 Constraint Satisfaction/Optimization Problems

A Constraint Satisfaction Problem (CSP) is defined as a triplet $\langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of *variables*, $D = \{D_1, \ldots, D_n\}$ is a set of *domains*, and C is a set of *constraints*. Each domain $D_i \in D$ is a *finite* set of values. Each constraint $c \in C$ involves a subset of m variables $vars(c) = \{x_{i_1}, \ldots, x_{i_m}\} \subseteq X$, depending on its semantic. A constraint defines a relation $c \subseteq D_{i_1} \times \cdots \times D_{i_m}$. A *solution* is an assignment $\sigma : X \to \bigcup_{i=1}^{n} D_i$ such that $\sigma(x_i) \in D_i$ holds for every variable, and $\langle \sigma(x_{i_1}), \ldots, \sigma(x_{i_m})\rangle \in c$ holds for every constraint.

A Constraint Optimization Problem (COP) is a quadruplet $\langle X, D, C, f\rangle$ where $\langle X D, C\rangle$ is a CSP and $f : D_1 \times \cdots \times D_n \to \mathbb{R}$ is an *objective function* to be (w.l.o.g.) minimized. The goal is to find a solution σ^* that minimizes $f(\sigma(x_1), \cdots, \sigma(x_n))$.

A *constraint solver* searches for solutions of a CSP/COP by alternating non-deterministic choices and constraints propagation. The first is employed to choose the next variable and which value, from its current domain, to assign to it. The second is a method to filter the domain of the variables, removing values that are not part of any solution. Non-deterministic choices are typically implemented through backtracking and heuristic decisions that follow an ordering among variables and values. Constraint propagation is commonly implemented through a queue that tracks constraints that need to be re-evaluated. When a value is removed from a variable's domain, the constraints involving such variable are enqueued. The re-evaluation consists of extracting the constraint from the queue and applying the associated filtering algorithm or *propagator*. This iterative cycle continues until the queue is empty [23].

Filtering algorithms offer trade-offs between filtering power and computational complexity. Highly effective algorithms have been developed for *global constraints*. These constraints model a substantial portion of a CSP/COP and naturally arise in many problems.

2.2 General-Purpose Computing on Graphics Processing Units

Performance in modern hardware is the by-product of parallel computing resources in the form of multi-core central processing units (CPUs) and general purpose graphical processing units (GPUs). Modern commodity hardware features CPUs with up to 64 cores (e.g., AMD Ryzen Threadripper 7980X) and GPUs with up to 16384 cores per card (e.g., NVIDIA GeForce RTX 4090). Yet, the number of cores in CPUs and GPUs are orders of magnitude apart, the programming models are wildly different and GPUs impose restrictions on code to deliver performance.

The massive parallelism of GPUs is a golden opportunity. To harness such computing power, it is crucial to employ approaches and algorithms that align with the underlying architecture of the GPU. Recent studies indicate that GPUs can be used for computational logic, including applications like Satisfiability [8, 7], Answer Set Programming [12, 13], and Constraint Programming [41, 42].

GPU-accelerated applications rely on APIs that expose parallel computing primitives. The most prominent is *CUDA*, a C/C++ API, introduced by NVIDIA for its own GPUs [28]. In a typical GPU-accelerated application, the GPU handles only the most computationally demanding tasks. The CPU executes the main application logic and choreographs the GPU(s) activities such as data transfers as well as computing tasks known as *kernels*. The components of an NVIDIA GPU utilized for general-purpose computing are depicted in Figure 1. A current high-end GPU[2] is equipped with 128 *Streaming Multiprocessors (SM)*, each housing 128 computational units named *CUDA Cores*, and 128 KB of fast memory. This memory serves as *L1 cache* and/or scratchpad memory, in which case it is referred to as *shared memory*. In the middle and lower tiers of the memory hierarchy, there is an *L2 cache* of 72 MB and the *global memory* with a capacity of 24 GB.

The CUDA execution model is *Single-Instruction Multiple-Thread (SIMT)*, where a C/C++ function known as *kernel* is executed by many threads. Each thread utilizes its own unique index to identify the data to use or to modify its control flow. When different threads follow distinct control flows, it leads to *thread divergence*. In such scenarios, threads are

---

[2] NVIDIA GeForce RTX 4090

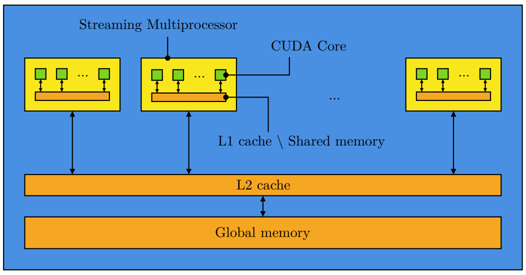


Figure 1 High level architecture of an NVIDIA GPU.

serialized, causing significant performance deterioration. Threads are organized into *blocks*, which are dispatched to the Streaming Multiprocessors. Each Streaming Multiprocessor executes the threads using its CUDA Cores, allowing efficient intra-block operations through shared memory. Communication between blocks is possible only through the use of global memory. To successfully leverage GPUs to accelerate expensive computations, it is essential to understand that they are designed to heavily trades raw execution speed for massive parallelization [20]. This often necessitates reformulating the problem to expose parallelism or exploiting shared memory to reduce costly global memory accesses.

In contrast, execution on multi-core CPUs relies on a *small number* of independent computing threads that execute fast, can have diverging behaviors with no performance penalties. Such an architecture can more readily adopt sequential code with the trade-off being the small number of threads (dozens rather than tens of thousands).

3 Bin Packing

Let $I = (c, W)$ be an instance of the Bin Packing Problem (BPP) with n items of weights $W = [w_1, \ldots, w_n]$, and bins of capacity c. The underling optimization problem can be formalized as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j=1}^{n} y_j \\
\text{subject to} \quad & \sum_{i=1}^{n} w_i x_{ij} \leq c y_j && j = 1, \ldots, n \\
& \sum_{j=1}^{n} x_{ij} = 1 && i = 1, \ldots, n \\
& y_j \in \{0, 1\} && j = 1, \ldots, n \\
& x_{ij} \in \{0, 1\} && i, j = 1, \ldots, n
\end{aligned}$$

where the variable y_j indicates whether the j^{th} bin is used and the variable x_{ij} indicates whether the i^{th} item is packed in the j^{th} bin.

One of the most effective approaches to solving the BPP involves adopting a graph-theoretical perspective. In the *Arc-Flow* method [9], a graph is constructed such that arcs represent items, and a path from the source s to the sink t represents a set of items that can be packed into a bin (see Figure 2). A solution corresponds to a minimum flow that utilizes one arc for each item $w \in W$. This flow problem is formulated and solved using an Integer Linear Programming (ILP) model with a robust linear relaxation, albeit with a pseudo-polynomial number of variables and constraints.

Algorithm 1 Simplified propagator for the *BinPacking* constraint.

```
Procedure: propagate$(c, W, k, X, L)$
for $j \leftarrow 1$ to $k$ do
    doLoadCoherence$(j, X, W, L)$
    doBasicLoadTightening$(j, X, W, L)$                        // Basic filtering
    for $i \in \{i \mid j \in x_i \wedge |x_i| > 1\}$ do
        doBasicItemEliminationCommitment$(i, j, X, W, L)$
for $j \leftarrow 1$ to $k$ do                                 // Knapsack filtering
    if $\neg$isBinPackable$(j, X, W, L)$ then Fail
    doKnapsackLoadTightening$(j, X, W, L)$
    for $i \in \{i \mid j \in x_i \wedge |x_i| > 1\}$ do
        doKnapsackItemEliminationCommitment$(i, j, X, W, L)$
$lb \leftarrow$ getLowerBound$(c, W, k, X)$                      // Feasibility check
if $lb > k$ then Fail
```

In CP, the decision version of the BPP, where the items must be packed in at most k bins, is modeled as:

$$\begin{aligned}
&x_i = \{1, \ldots, k\} \qquad i = 1, \ldots, n \\
&l_j = \{0, \ldots, c\} \qquad j = 1, \ldots, k \\
&\mathit{BinPacking}(W = [w_1, \ldots, w_n], X = [x_1, \ldots, x_n], L = [l_1, \ldots, l_k])
\end{aligned}$$

where the variable x_i represents the bins in which the i^{th} item can be packed, and the variable l_j represents the loads that the j^{th} bin can have. The *BinPacking* constraint was introduced in [38] and a simplified version of its filtering algorithm is listed in Algorithm 1. The following offers a brief description of each call in Algorithm 1:

doLoadCoherence Adjust the minimum/maximum load of a bin based on the total weight of the items and the load of the other bins.

doBasicLoadTightening Adjust the minimum/maximum load of a bin based on the sum of the items that are or can be packed in the bin.

doBasicItemEliminationCommitment An item is committed to a bin if it is needed to reach a valid load. An item is excluded from a bin if packing it would lead to an excessive load.

isBinPackable Checks whether a bin is packable based on an approximated knapsack reasoning to reach an admissible load.

doKnapsackLoadTightening Adjust the minimum/maximum load of a bin with an approximated knapsack reasoning.

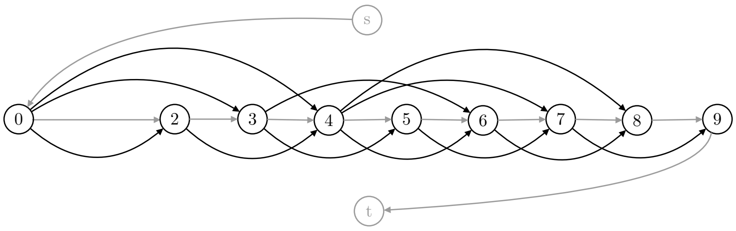
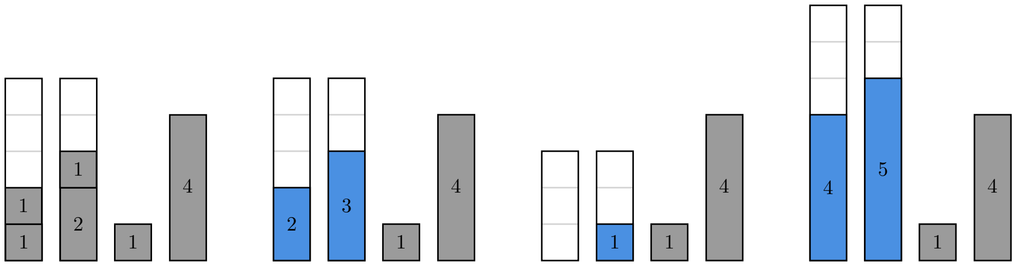


Figure 2 Graph underling Arc-Flow for an instance with $c = 9$ and $W = [4, 4, 3, 3, 2, 2]$.



(a) Partial Solution.
(b) Reduced Solution with R_0 (merge items).
(c) Reduced Solution with R_{Min} (Reduce bins by smallest virtual item).
(d) Reduced Solution with R_{Max} (Enlarge bins and virtual items).

Figure 3 Illustrations of a partial packing of the instance $I = (5, [4, 2, 1, 1, 1, 1])$, and reductions $R_0 = (5, [4, 3, 2, 1])$, $R_{Min} = (3, [4, 1, 1])$, $R_{Max} = (7, [5, 4, 4, 1])$. Virtual items are colored in blue.

getLowerBound A partial packing is considered feasible if a lower bound on the number of bins does not exceed the number of available bins. This lower bound, referred to as L_2 (see Section 3.2), is calculated on a *reduced* instance derived from the current partial packing (see Section 3.1).

The literature contains various enhancement of the *BinPacking* constraint. The authors of [35, 30, 11] introduced and refined a *cardinality reasoning*, well suited when there are constraints on the number of items in each bin or when the items have similar weights. In [3], it was employed a tight lower bound derived from the *linear relaxation* of the Arc-Flow model.

3.1 Reductions

Given a partial packing of an instance $I = (c, W)$, a reduction R provides an instance $I_R = (c_R, W_R)$ such that a lower bound for I_R is valid for the partial packing. Such partial packing is inferred from the variables X.

The standard reduction, knows as R_0, maintains the same capacity, all the unpacked items, and introduces *virtual items* representing the items packed in each bin (see Figure 3b). Other reductions similar to R_0 are possible. For instance, [14] introduced R_{Min} and R_{Max}. The first decreases the capacity of the bins and the virtual items by the size of the smallest virtual item (see Figure 3c). The second increases the capacity of the bins and the virtual items by a common quantity, so that two virtual items can not fit in the same bin. This is achieved when each virtual item is bigger than half of the bin capacity (see Figure 3d).

3.2 Lower bounds

Given an instance, $I = (c, W)$, a lower bound $L(I)$ estimates the minimum number of bins necessary to store the items. The simplest lower bound is referred to as L_1, and is calculated as follows:

$$L_1(I) = \left\lceil \frac{1}{c} \sum_{w \in W} w \right\rceil$$

where the total weight of the items is divided by the bin capacity, and the ceiling function is applied. This approach is equivalent to naively packing the items, cutting those that do not entirely fit.

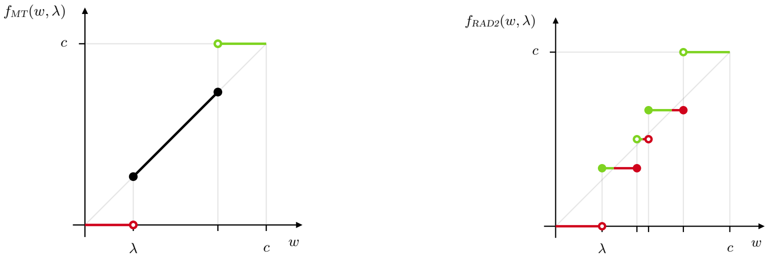


Figure 4 f_{MT} (left) and f_{RAD2} (right) for $\lambda = c\frac{4}{15}$. Weights that have been increased/decreased are shown in green/red.

An improvement of L_1, called L_2, was introduced in [25] and addresses the cases where big items cannot be packed together. It is defined as:

$$L_2(I) = \max_{0 \le \lambda \le \frac{c}{2}} L_2(I, \lambda)$$

where

$$L_2(I,\lambda) = |W_1| + |W_2| + max\left(0, \left\lceil \frac{1}{c}\left(\sum_{w \in W_3} w - \left(c\,|W_2| - \sum_{w \in W_2} w\right)\right)\right\rceil\right)$$
$$W_1 = \{w \mid w \in W \wedge c - \lambda < w\}$$
$$W_2 = \{w \mid w \in W \wedge \tfrac{c}{2} < w \le c - \lambda\}$$
$$W_3 = \{w \mid w \in W \wedge \lambda \le w \le \tfrac{c}{2}\}$$

The lower bound $L_2(I,\lambda)$ classifies the items as big (W_1), medium-big (W_2), medium-small (W_3), while it ignores the smallest items. Note how the definition of the sets are *parameterized* by λ. Then all the big and medium-big items are packed in different bins since they are all bigger than $\frac{c}{2}$. The medium-small items are packed as in L_1, using the available space in the bins where there is a medium-big item before considering other bins. Finally, the small items are just dropped. A direct implementation of L_2 is pseudo-polynomial, since $L_2(I,\lambda)$ has to be calculated exactly once for each $\lambda \in \left[0, \frac{c}{2}\right]$, i.e., $\Theta(c)$ times. A linear complexity can be achieved when the items are sorted in decreasing weight [24, 22]. Note how $L_2(I)$ defines a *family of lower bounds*, with one member for each $\lambda \in \left[0, \frac{c}{2}\right]$.

A general approach to enhance L_1, derived from duality theory, is based on *Dual Feasible Functions (DFFs)* [1]. Intuitively, a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ is dual feasible if, for every subset $W_S \subseteq W$, the following holds:

$$\sum_{w \in W_S} w \le c \quad \Rightarrow \quad \sum_{w \in W_S} f(w) \le f(c)$$

Consider the $f_{MT}(w,\lambda)$ definition below that keeps the same capacity, while defining new weights for items. It increases the weights of large items ($c - \lambda < w$) to c, decrease the weights of small items ($w < \lambda$) to 0, and leave the weights of medium items unchanged ($\lambda \le w \le c - \lambda$), i.e., they are w. Note that increasing the weight to c is equivalent to allocating an entire bin for the item, while decreasing the weight to 0 disregards the item. The function, shown in Figure 4, depends on an *integer* parameter λ:

$$\underset{0 \le \lambda \le \frac{c}{2}}{f_{MT}(w,\lambda)} = \begin{cases} c & \text{if } c - \lambda < w \\ w & \text{if } \lambda \le w \le c - \lambda \\ 0 & \text{if } w < \lambda \end{cases}$$

The lower bound obtained by combining L_1 with f_{MT} is:

$$L_{MT}(I) = \max_{0 \le \lambda \le \frac{c}{2}} \left\lceil \frac{1}{f_{MT}(c, \lambda)} \sum_{w \in W} f_{MT}(w, \lambda) \right\rceil \tag{1}$$

and it is equal to L_2 [16]. Other DFFs have been proposed, each with a different design for revising weights. For brevity, we report only some of them and refer interested readers to [6, 1] for a comprehensive review, and to [31, 32] for further insights.

$$\underset{\frac{c}{4} < \lambda \le \frac{c}{3}}{f_{RAD2}(w, \lambda)} = \begin{cases} 0 & \text{if } w < \lambda \\ \left\lfloor \frac{c}{3} \right\rfloor & \text{if } \lambda \le w \le c - 2\lambda \\ \left\lfloor \frac{c}{2} \right\rfloor & \text{if } c - 2\lambda < w < 2\lambda \\ c - f_{RAD2}(c - w, \lambda) & \text{if } 2\lambda \le w \end{cases}$$

$$\underset{1 \le \lambda \le 100}{f_{FS1}(w, \lambda)} = \begin{cases} w\lambda & \text{if } \frac{w(\lambda+1)}{c} \in \mathbb{N} \\ \left\lfloor \frac{w(\lambda+1)}{c} \right\rfloor c & \text{otherwise} \end{cases}$$

$$\underset{1 \le \lambda \le \frac{c}{2}}{f_{CCM1}(w, \lambda)} = \begin{cases} 2 \left\lfloor \frac{c}{\lambda} \right\rfloor - 2 \left\lfloor \frac{c-w}{\lambda} \right\rfloor & \text{if } w > \frac{c}{2} \\ \left\lfloor \frac{c}{\lambda} \right\rfloor & \text{if } w = \frac{c}{2} \\ 2 \left\lfloor \frac{w}{\lambda} \right\rfloor & \text{if } w < \frac{c}{2} \end{cases}$$

$$\underset{2 \le \lambda \le c}{f_{VB2}(w, \lambda)} = \begin{cases} 2 \max\left(0, \left\lceil \frac{c\lambda}{c} \right\rceil - 1\right) - 2 \max\left(0, \left\lceil \frac{(c-w)\lambda}{c} \right\rceil - 1\right) & \text{if } w > \frac{c}{2} \\ \max\left(0, \left\lceil \frac{c\lambda}{c} \right\rceil - 1\right) & \text{if } w = \frac{c}{2} \\ 2 \max\left(0, \left\lceil \frac{w\lambda}{c} \right\rceil - 1\right) & \text{if } w < \frac{c}{2} \end{cases}$$

$$\underset{1 \le \lambda \le c}{f_{BJ1}(w, \lambda)} = \begin{cases} \left\lfloor \frac{w}{\lambda} \right\rfloor (\lambda - c \bmod \lambda) & \text{if } w \bmod \lambda \le c \bmod \lambda \\ \left\lfloor \frac{w}{\lambda} \right\rfloor (\lambda - c \bmod \lambda) + w \bmod \lambda - c \bmod \lambda & \text{otherwise} \end{cases}$$

Interestingly, these five definitions are all parametric in λ and define 5 additional families, most with $\Theta(c)$ members (except f_{FS1}). To get the best possible bound, one would need to compute the bounds for each family and across all parameter values in that family. To reduce the sequential computational burden, one could resort to only computing some families, or computing only a subset of different λ values in each admissible range. Alternatively, one can adopt parallel techniques as *all* families and *all* λ values can be computed independently. The next section studies this tradeoff.

4 Design and Implementation

To determine the most convenient DFF to use, we examined the lower bounds derived from various DFFs on the Falkenauer and Scholl instances (see Section 5). The results in Table 1 confirm f_{CCM1} as the best overall function [6], while the generally weak f_{RAD2} proves effective

Table 1 Statistics for different DFF-based lower bounds on the Falkenauer and Scholl instances.

DFF	Only Opt	Total Opt	Only Best	Total Best	Sum
f_{MT}	2	1151	0	55	120184
f_{RAD2}	10	189	0	36	105345
f_{FS1}	2	742	0	45	119504
f_{CCM1}	40	1219	1	60	120270
f_{VB2}	1	973	0	40	119786
f_{BJ1}	47	1101	0	50	120039

Algorithm 2 Sequential DFFs-based *getLowerBound* function.

```
    Function: $getLowerBound(c, W, k, X) \rightarrow lb$
 1  $lb \leftarrow 0$
 2  for $R \in \{R_0, R_{Min}, R_{Max}\}$ do
 3      $(c_R, W_R) \leftarrow R(c, W, X)$
 4      for $f \in \{f_{CCM1}, f_{MT}, f_{BJ1}, f_{VB2}, f_{FS1}, f_{RAD2}\}$ do
 5          $L_f \leftarrow 0$
 6          $(\underline{\lambda}, \overline{\lambda}, \delta) \leftarrow getParametersMinMaxStep(f, c_R, 256)$
 7          for $\lambda \leftarrow \underline{\lambda}$ to $\overline{\lambda}$ by $\delta$ do
 8              $sum \leftarrow \sum_{w_R \in W_R} f(w_R, \lambda)$
 9              $L_f \leftarrow max(L_f, \lceil \frac{sum}{f(c_R, \lambda)} \rceil)$
10          $lb \leftarrow max(lb, L_f)$
11          if $lb > k$ then return $lb$          // Infeasibility detected, early return
12  return $lb$
```

when stronger functions are suboptimal [31]. Since no DFF family dominates, it is apparent that restricting ourselves to *choosing* a single family is not productive. Instead, a *portfolio* of independent DFFs should be computed with parallel resources to deliver stronger pruning at virtually no cost (in term of wall-clock time). Recall that the calculation of a single family of lower bound is still pseudo-polynomial and can be costly for large c values. Ideally, one would consider only a minimal subset of parameters guaranteed to lead to the tightest bound, but this is only possible for L_{MT} [16]. In practice, for the CPU implementations, we consider a sampling of 256 equispaced λ values for each family as it proved empirically adequate for obtaining effective bounds.

Similar design considerations were done about the reduction(s) to employ. The analysis in [14] suggest using both R_{Min} and R_{Max}. However, preliminary experiments showed that R_0 is beneficial in some instances, so we considered all of them.

4.1 Sequential CPU Implementation

A sequential DFFs-based implementation of the function *getLowerBound* (see Algorithm 1) is listed in Algorithm 2. It has a nested loop structure where the loop at line 2 consider the three reductions presented in Section 3.1, the loop at line 4 consider the six DFFs in the portfolio, and the loop at line 7 samples the rage of parameters. That results in computation that sequentially calculates $3 * 6 * 256 = 4608$ lower bounds.

4.2 Parallel CPU Implementation

The nested loop structure of Algorithm 2 is easily parallelizable since all iterations are independent. The only data that need to be atomically updated is the maximum lower bound at line 10.

The outermost 2 loops execute the main body of the function (lines 5–10) 18 times (i.e., 3 reductions and 6 DFFs). To easily run on commodity CPU with about 10 cores, it is appropriate to use one thread per DFF to executes the main body sequentially for all 3 reductions. This approach uses 6 threads, each calculating $3 * 256 = 768$ lower bounds. It provides a *sublinear* speedup of 2x when compared to the fully sequential implementation (see Section 5). While it is possible to also parallelize all 3 reductions on a machine with at least 18 cores, it did not seem to be a promising avenue.

The parallel implementation is obtained using OpenMP [29], a C/C++ API that enable transparent multi-threading by simply adding annotations, or *directives*, to the loops. We use the `omp parallel for num_threads(6)` to parallelize the DFF loop, and the directive `reduction(max:lowerbound)` to correctly update the maximum lower bound.

4.3 Considerations for a GPU architecture

To successfully leverage GPUs it is fundamental to understand the weakness and strengths of their architecture. The efficiency of a CPU stems from its *low latency*, which indicates the time required to execute individual operations. Mechanisms such as branch prediction, multiple levels of fast cache, and high clock speeds all contribute to making each of the "few" CPU threads extremely fast. In contrast, the efficiency of a GPU is grounded in its *high throughput*, which represents the number of operations executed per unit of time. The vast number of threads, coupled with rapid context switching, makes the GPU highly effective in performing extensive workloads, compensating for its high latency.

There are various approaches to accelerate propagation algorithms with GPUs. One approach is to parallelize the most prominent algorithm(s). While this seems appealing, it is hard to accomplish for two reasons. First, such algorithms are often designed with a sequential model in mind, making them challenging to parallelize. Data dependencies between iterations as well as the need to synchronize for data structure updates are at the heart of the problem. The second reason is the GPUs high latency, mainly due to the "simple" memory hierarchy where a L1 cache miss results in costly off-chip memory access, as well as the time required to move data and control to and from the GPUs. The optimal point to offload a computation to the GPU changes based on several factors, including hardware characteristics. It is often the case that data transfer negates the benefits of parallelization. This overhead disappears once the GPU workload is large enough. Empirically, it is generally not helpful to offload the propagation of algorithms with a time complexity of $O(n^2)$ or lower.

Another strategy involves utilizing the GPU to reduce the computational cost of strong filtering algorithms [42]. This idea can be applied to the *BinPacking* constraint by employing the GPU to perform a *complete knapsack reasoning* instead of an approximated one. Using the Dynamic Programming (DP) approach presented in [43] it is possible to obtain a stronger filtering that replace all the basic and knapsack filtering in Algorithm 1. We developed a GPU-accelerated implementation of this *pseudo-polynomial* method, leveraging bitwise operations and processing each bin in parallel. Empirical results revealed no significant gains in terms of explored nodes (within the time limits) compared to the approximated reasoning. Scalability tests further indicate that the GPU-accelerated implementation becomes faster than an optimized implementation of the approximated filtering when the number of bins is in the order of *hundreds*. This evidence indicates that theoretically interesting implementations may encounter overheads that outweigh the computational benefits. Ultimately, the disappointing results pushed this second strategy aside.

GPUs can also enhance *pruning*. In the case of the *BinPacking* constraints, this translates into improving the feasibility check to obtain the best possible lower bounds at a reduced computational cost. The tightest available lower bound is derived from the linear relaxation of the Arc-Flow model (see Section 3), which involves solving a sparse linear system. Since this task is notoriously challenging to effectively accelerate with GPUs [20], we explored the next option: considering *all the parameters* and all the DFF families.

Algorithm 3 GPU-accelerated DFFs-based *getLowerBound* function.

```
Function: $getLowerBound(c, W, k, X) \rightarrow lb$
$[I_{R_0}, I_{R_{Min}}, I_{R_{Max}}] \leftarrow calcReductions([R_0, R_{Min}, R_{Max}], (c, W, X))$
$lb \leftarrow 0$
$cudaMemcpyCpuToGpu([lb, I_{R_0}, I_{R_{Min}}, I_{R_{Max}}])$                    // Asynchronous API
for $(c_R, W_R) \in \{I_{R_0}, I_{R_{Min}}, I_{R_{Max}}\}$ do
    for $f \in \{f_{CCM1}, f_{MT}, f_{BJ1}, f_{VB2}, f_{FS1}, f_{RAD}\}$ do
        $(\underline{\lambda}, \overline{\lambda}) \leftarrow getParametersMinMax(f, c_R)$
        $nThreads \leftarrow \overline{\lambda} - \underline{\lambda} + 1$
        $cudaLaunchKernel(calcDffLowerBound, nThreads, [f, c_R, W_R, \ldots])$   // Async API
$cudaMemcpyGpuToCpu(lb)$                                                        // Asynchronous API
$waitGpu()$                                                                      // Synchronous API
return $lb$
```

4.4 GPU Implementation

We handled each combination reduction-DFF with a separate kernel, and each of the $\overline{\lambda} - \underline{\lambda} + 1$ parameter with a different thread (see Figure 5). The GPU-accelerated implementation of the method *getLowerBound* is outlined in Algorithm 3. The first operation copies the reduced instances and the initial lower bound into the GPU's global memory. The amount of transferred data is minimal, and encoded as an array of integers. After that, 18 kernels are launched, each with the appropriate number of threads and several arguments, including the DFF and reduction that they must consider. Finally, the highest lower bound is copied back from the GPU and returned.

The heart of the parallelization is the kernel *calcDffLowerBound*, listed in Algorithm 4. The line 2 shows how each thread uses its index to identify the parameter it works on. The barrier at line 4 ensures the initialization of L_f, and prevents race conditions on its value. Lines 5–6 calculates the value of $L_f(c_R, W_R)$. Finally, the barrier at line 7 guarantees that all parameters are considered before updating the best lower bound.

The pseudocode abstracts out some implementation details that are worth mentioning. From Section 2.2, we recall that the threads of a kernel are organized into *blocks*, each executing in a Streaming Multiprocessor with its own on-chip *shared memory*. This fast memory reduces accesses to the slower global memory in two ways. First, it caches c_R and W_R, ensuring fast access for subsequent lower bound calculations. Second, it maintains L_f enabling faster atomic *max* operations (line 6) that run concurrently between blocks. However, the final atomic *max* operation (line 8) must be performed on global memory, as it is the only means of communication among blocks and kernels.

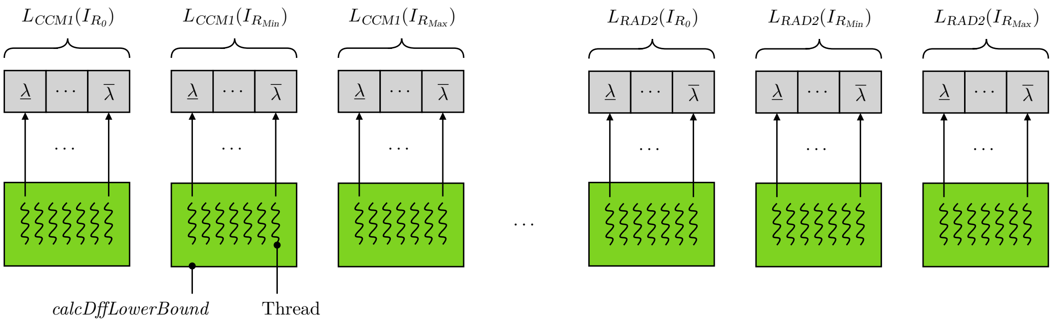


Figure 5 Parallelism of the GPU accelerated *getLowerBound*.

Algorithm 4 Pseudocode of the *calcDffLowerBound* kernel.

```
Procedure: $calcDffLowerBound(f, c_R, W_R, k, lb, \underline{\lambda}, \overline{\lambda})$
if $lb \leq k$ then
    $\lambda \leftarrow \underline{\lambda} + getThreadIdx()$
    $L_f \leftarrow 0$                                                    // Only one thread
    $threadsBarrier()$
    $sum \leftarrow \sum_{w_R \in W_R} f(w_R, \lambda)$
    $L_f \leftarrow max(L_f, \lceil \frac{sum}{f(c_R, \lambda)} \rceil)$                         // Atomic operation
    $threadsBarrier()$
    $lb \leftarrow max(lb, L_f)$                            // Only one thread, atomic operation
```

4.5 Solver integration

There are no limitations that prevent the GPU-accelerated *getLowerBound* to be used in the *BinPacking* propagator of a standard CP solver. However, there are a couple of aspects that facilitates such task. Unsurprisingly, it is easiest to integrate in solvers written in C/C++ since CUDA is a C/C++ API, and no wrappers or bindings are needed. Moreover, kernels can be compiled with(in) the solver, without the need to compile them separately and load them at runtime. From the usability prospective, it would be convenient that the solver is compatible with the high-level constraint modelling language *MiniZinc* [27]. By using its annotation mechanism, it is possible to communicate to the solver which implementation of *getLowerBound* to use. For example, when a *BinPacking* constraint is added, it can be annotated with `::parallel` to use the CPU parallel version, or with `::gpu` to use the GPU-accelerated implementation.

We implemented the different versions of *getLowerBound*, along with the relative annotations, within a solver compatible with MiniZinc [39]. Such solver is based on MiniCPP [18], a C++ implementation of MiniCP [26]. We choose MiniCP(P) because it is open-source, well documented, and reasonably simple to modify.

5 Experiments

This section presents a comparison between propagators that use different lower bounds for the feasibility check. We evaluate our linear time complexity implementation of L_2 (i.e, `L2`), our sequential (i.e., `DFFs-CPU-Seq`), parallel (i.e., `DFFs-CPU-Par`), and GPU (i.e., `DFFs-GPU`) DFFs-based implementations, and the implementation from [3] which uses the Arc-Flow based lower bound (i.e., `Arc-Flow`). We select two BPP benchmarks from the literature [15, 37], and generate new instances similar to the ones proposed in [5] and [4]. This results in a total of 2072 instances [40] organized as follows:

Falkenauer This benchmark has two classes of 80 instances each. The ‘U’ instances have items with weights uniformly distributed in $[20, 100]$, $n \in \{120, 250, 500, 1000\}$ and $c = 150$. The ‘T’ instances are characterized by triplets of items that must be packed in the same bin in any optimal solution. For this class $n \in \{60,\ 120,\ 249,\ 501\}$ and $c = 1000$.

Scholl These instances are divided into three sets of 720, 480, and 10 instances. The instances in Set 1 have weights uniformly distributed to expect a number of items per bin not larger than three, $n \in \{50,\ 100,\ 200,\ 500\}$, $c \in \{100, 120, 150\}$. For the instances in Set 2 the number of expected items per bin is between three and nine items, $n \in \{50, 100, 200, 500\}$, $c = 1000$. Set 3 contains big instances with weights uniformly distributed in the range $[20000, 35000]$, $n = 200$ and $c = 100000$.

- **Weibull** These instances are based on the Weibull probability distribution. It can model various distributions found in different problem domains by adjusting the shape parameter $k > 0$ and the scale parameter $\lambda > 0$. Similarly to [5], we generated 92 sets of weights W with the parameters $n \in \{100, 200\}$, $k \in \{0.5, 0.6, \ldots, 5.0\}$, and $\lambda = 1000$. For each set W, we generate 6 instances (c, W) with $c = \sigma \cdot \max(W)$ for $\sigma \in \{1.0, 1.2, \ldots, 2.0\}$. The total number of instances is 552, with capacity ranging between 1300 and 92500.
- **Scaled Non-IRUP** These instances are derived from instances which *do not* satisfy the *Integer Round-Up Property (IRUP)*. Intuitively, an instance is IRUP if the roundup value of the (strongest) linear relaxation yields to the optimal number of bins. We considered 50 of the instances in [4]. For each instance (c, W) and $s \in \{3, 4, 5\}$, we derived (c_s, W_s) such that $c_s = s * c$ and W_s is the list containing s times the set $\{s * w \mid w \in W\}$. The total number of instances is 150, with $n \in \{45, 60, 75\}$ and c in the range $[921, 5240]$.

The model and search heuristic are the same as in previous works [38, 3], where a minimum number of bins is established and an attempt to find a solution is made. If such a solution does not exist, the number of bins is increased, and a new attempt is made. All implementations use the *decreasing best fit* search heuristic. In this strategy, the items are considered in descending order of weight and assigned to the first bin within their domain that has the smallest residual capacity sufficient to accommodate the item. Additionally, two symmetry-breaking rules are applied on backtracking: first, the bin is removed from the domain of all items of the same size, and second, all the bins with the same load are removed from the domains of these items. Finally, a dominance rule is applied before a choice point: if an item completely fills the remaining capacity of a bin, it is assigned to that bin.

The implementations `L2`, `DFFs-CPU-Seq`, `DFFs-CPU-Par`, and `DFFs-GPU` include a couple of additional techniques. First, another dominance rule is applied before a choice point: if among the set of candidate items that can be packed in a bin, only one can be packed, then the heaviest item is assigned to the bin [34]. Second, the symmetry breaking described in [33] is enforced with an additional constraint. Cardinality reasoning was considered but set aside in preliminary experiments, as it did not yield notable differences in terms of explored nodes while adding some overhead. This can be attributed to the combined effects of strong pruning and the absence of cardinality constraints in our benchmarks

The experiments are performed with 10 minutes timeout to ensure a reasonable benchmark time. The test system features an Intel Core i7-10700K (8 Cores), 32 GB of RAM, and an NVIDIA GeForce RTX 3080 (8704 CUDA Cores). The system operates on Ubuntu Linux 22.04 LTS and uses CUDA 11.8 and GCC 11.4 for our implementations, along with OpenJDK 11.0 and CPLEX 22.1 for `Arc-Flow`.

Results and Analysis

The analysis focuses on instances solved within the 10 minutes time limit. Table 2 reports, for each approach and benchmark, the number of solved instances, the average time per instance, the total solving time, and the total number of visited nodes. Instances that time out are not contributing anything to the total time, average time or nodes column.

Global Analysis. Falkenauer T instances highlight the contrast between fast and slow pruning. `DFF-GPU` quickly solved 73% of the instances, while `Arc-Flow` solved 85% of them taking, on average, 3x more time. The other DFFs-based approaches fall in the middle, and `L2` is last.

Table 2 Statistics for the solved instances of different lower bound methods.

Benchmark (Instances)	Lower Bound	Solved	Avg Time [s]	Time [s]	Nodes
Falkenauer T (80)	`L2`	38	19	733	623305
	`DFFs-CPU-Seq`	46	44	2045	105856
	`DFFs-CPU-Par`	47	28	1315	154440
	`DFFs-GPU`	58	11	650	448780
	`Arc-Flow`	**68**	31	2120	5235
Falkenauer U (80)	`L2`	30	17	496	481984
	`DFFs-CPU-Seq`	56	43	2382	106646
	`DFFs-CPU-Par`	57	39	2198	122521
	`DFFs-GPU`	60	31	1888	357108
	`Arc-Flow`	**79**	16	1303	16012
Scholl 1 (720)	`L2`	637	6	4057	7126695
	`DFFs-CPU-Seq`	696	6	3961	593028
	`DFFs-CPU-Par`	698	5	3398	1361855
	`DFFs-GPU`	703	3	1952	3997093
	`Arc-Flow`	**717**	6	4097	116135
Scholl 2 (480)	`L2`	332	2	771	2777677
	`DFFs-CPU-Seq`	391	8	3035	273011
	`DFFs-CPU-Par`	391	4	1421	273011
	`DFFs-GPU`	**440**	2	827	1235237
	`Arc-Flow`	423	69	29287	278014
Scholl 3 (10)	`L2`	–	–	–	–
	`DFFs-CPU-Seq`	–	–	–	–
	`DFFs-CPU-Par`	–	–	–	–
	`DFFs-GPU`	**3**	1	4	4322
	`Arc-Flow`	–	–	–	–
Weibull (552)	`L2`	371	6	2350	13082358
	`DFFs-CPU-Seq`	395	6	2381	342116
	`DFFs-CPU-Par`	397	4	1782	669801
	`DFFs-GPU`	**417**	6	2636	18149205
	`Arc-Flow`	286	105	30046	11103
Scaled Non-IRUP (150)	`L2`	82	52	4303	63979492
	`DFFs-CPU-Seq`	82	71	5836	4214685
	`DFFs-CPU-Par`	90	61	5520	8329009
	`DFFs-GPU`	**116**	52	6071	41873760
	`Arc-Flow`	108	7	1866	6388

In the Falkenauer U instances, `Arc-Flow` demonstrates a good balance between speed and strength, solving almost all instances in a short amount of time. The DFFs-based approaches have similar performance, suggesting that the computation of lower bounds is negligible. This happens when failures occur earlier in the propagation, during the knapsack reasoning.

In the Scholl 1 instances, the gap between `Arc-Flow` and the DFF-based approaches diminishes notably. `DFFs-GPU` outpaces the CPU approaches by a factor of 2x and 1.7x on average. Notably, while achieving tighter bounds, `DFFs-GPU` explores, on average, 14x and 5x more nodes per second compared to the CPU implementations. It has the lowest runtime per instance and completes 703 instances in half the time of all other contenders.

On Scholl 2, `DFFs-GPU` clearly dominates the field. It solves the most instances (440 out of 480), completes 35 times faster than the second best (Arc-Flow) and clearly improves on its parallel and sequential brethren (2x to 4x faster).

Scholl 3 instances are characterized by huge capacities and highlight the benefits of the GPU approach. It was the only method able to solve any instance leveraging tighter bounds than `L2`, `DFFs-CPU-Seq`, and `DFFs-CPU-Par`, while also being faster than `Arc-Flow`.

The Weibull instances, whose capacities range from medium to large, favor faster computation over strong pruning. In terms of instances solved, `DFFs-GPU` comes first, followed by the other DFFs-based approaches, then `L2`, and `Arc-Flow` last. While `DFFs-CPU-Par` ekes out a win on time per instance, it solves 10% fewer instances than its GPU version. Such instances account for the higher `DFFs-GPU` average solving time. Considering the exploration speed, `DFFs-GPU` visited, on average 48x more nodes than the sequential version and 18x more nodes than the multi-core version.

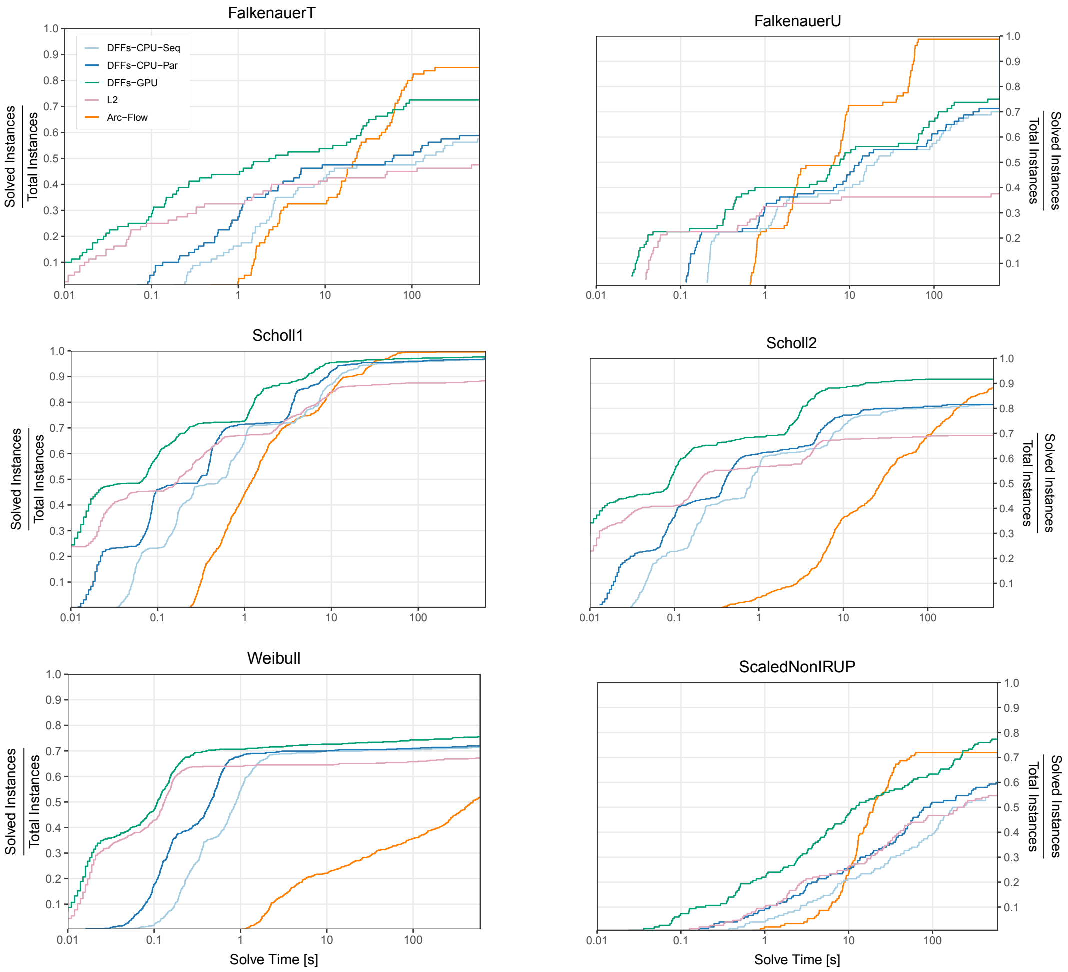


Figure 6 Plots of the empirical cumulative distribution for the benchmarks.

The Scaled Non-IRUP instances *stress-tests* the lower bound capabilities of solvers. While `DFFs-GPU` solves the most instance and is followed by `Arc-Flow`, observe that the hardness is not a function of the number of items. Indeed, `Arc-Flow` times-out on small instance with 45 items, but takes the crown on instances with 60 or 75 items. The remaining contenders are much weaker as an additional 22% of the instance solved by `DFFs-GPU` remain out of reach for `L2` and the sequential DFF implementation, further highlighting the value of a GPU.

Cumulative Analysis. Instance hardness in each benchmark suite is far from uniform. All methods can quickly solve some instances, yet they sharply diverge on others. Cumulative plots for six benchmark classes appear in Figure 6 (School 3 is omitted as `DFF-GPUs` alone could solve instances). The logarithmic horizontal axis is the solving time, while the vertical axis indicates the percentage of instances solved in that time. The `DFFs-GPU` is the green curve and it is readily apparent that it is the north-most, left-most curve in the plots. Indeed, it generally solves more instances significantly faster. The Scaled Non-IRUP instances exhibit an interesting behavior where `DFFs-GPU` and `Arc-Flow` switch roles *twice* as the most effective technique. `DFF-GPUs` is the top-most curve for most values along the x axis.

Table 3 Statistics for `DFFs-GPU` without optimizations.

Version	Solved	Time [s]	Nodes
`DFFs-GPU`	1571	1359	1582808
`DFFs-GPU-NoDom`	1544	2457	83559849
`DFFs-GPU-NoSymBrk`	1526	3970	143491180

Ablation Analysis. An ablation study was conducted on instances solved by `DFFs-GPU` in less than 60 seconds (see Table 3). The most effective technique is the symmetry breaking constraint derived from [33], which is quite general as it applies to variations of the BPP.

6 Conclusions and Future works

This paper revisits the *BinPacking* constraint from a parallel prospective and demonstrates how a parallel mindset leads to novel approaches. It presents a feasibility check based on a portfolio of lower bounds derived from Dual Feasible Functions (DFFs). Sequential, multi-threaded, and GPU-accelerated implementations are described and compared.

The results highlight the role of GPUs and how to achieve an effective balance between computational cost and pruning strength. It allows to handle large instances or situations where it is not practical to spend excessive time at nodes of the search tree. From an analytical standpoint, it would be interesting to identify DFFs that lead to tight bounds in cases where the current ones fall short. Practically, a valuable extension is to explore the effectiveness of multidimensional DFFs [2] on 2D, 3D and Vector Packing Problems.

References

1 Cláudio Alves, Francois Clautiaux, José Valério de Carvalho, and Jürgen Rietz. *Dual-feasible functions for integer programming and combinatorial optimization.* EURO Advanced Tutorials on Operational Research. Springer International Publishing, Basel, Switzerland, 2016.

2 Cláudio Alves, José M. Valério de Carvalho, François Clautiaux, and Jürgen Rietz. Multidimensional dual-feasible functions and fast lower bounds for the vector packing problem. *Eur. J. Oper. Res.*, 233(1):43–63, 2014. `doi:10.1016/J.EJOR.2013.08.011`.

3 Hadrien Cambazard and Barry O'Sullivan. Propagating the bin packing constraint using linear programming. In David Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, pages 129–136. Springer, 2010. `doi:10.1007/978-3-642-15396-9_13`.

4 Alberto Caprara, Mauro Dell'Amico, José Carlos Díaz Díaz, Manuel Iori, and Romeo Rizzi. Friendly bin packing instances without integer round-up property. *Math. Program.*, 150(1):5–17, 2015. `doi:10.1007/S10107-014-0791-Z`.

5 Ignacio Castiñeiras, Milan De Cauwer, and Barry O'Sullivan. Weibull-based benchmarks for bin packing. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2012. `doi:10.1007/978-3-642-33558-7_17`.

6 François Clautiaux, Cláudio Alves, and José M. Valério de Carvalho. A survey of dual-feasible and superadditive functions. *Ann. Oper. Res.*, 179(1):317–342, 2010. `doi:10.1007/S10479-008-0453-8`.

7 Michele Collevati, Agostino Dovier, and Andrea Formisano. GPU parallelism for SAT solving heuristics. In Roberta Calegari, Giovanni Ciatto, and Andrea Omicini, editors, *Proceedings of the CILC'22*, volume 3204 of *CEUR Workshop Proceedings*, pages 17–31. CEUR-WS.org, 2022.

8 Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUD@SAT: SAT solving on GPUs. *J. Exp. Theor. Artif. Intell.*, 27(3):293–316, 2015. `doi:10.1080/0952813X.2014.954274`.

9 José M. Valério de Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Ann. Oper. Res.*, 86:629–659, 1999. `doi:10.1023/A\%3A1018952112615`.

10 Maxence Delorme, Manuel Iori, and Silvano Martello. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*, 255(1):1–20, November 2016. `doi:10.1016/j.ejor.2016.04.030`.

11 Guillaume Derval, Jean-Charles Régin, and Pierre Schaus. Improved filtering for the bin-packing with cardinality constraint. *Constraints An Int. J.*, 23(3):251–271, 2018. `doi:10.1007/S10601-017-9278-X`.

12 Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Parallel answer set programming. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 237–282. Springer, 2018. `doi:10.1007/978-3-319-63516-3_7`.

13 Agostino Dovier, Andrea Formisano, and Flavio Vella. GPU-Based Parallelism for ASP-Solving. In Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen, and Dietmar Seipel, editors, *Declarative Programming and Knowledge Management*, volume 12057 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2019. `doi:10.1007/978-3-030-46714-2_1`.

14 Julien Dupuis, Pierre Schaus, and Yves Deville. Consistency check for the bin packing constraint revisited. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 117–122. Springer Berlin Heidelberg, 2010. `doi:10.1007/978-3-642-13520-0_15`.

15 Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *J. Heuristics*, 2(1):5–30, 1996. `doi:10.1007/BF00226291`.

16 Sándor P. Fekete and Jörg Schepers. New classes of fast lower bounds for bin packing problems. *Math. Program.*, 91(1):11–31, 2001. `doi:10.1007/S101070100243`.

17 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

18 Rebecca Gentzel, Laurent Michel, and W.-J. van Hoeve. HADDOCK: A language and architecture for decision diagram compilation. In *Lecture Notes in Computer Science*, pages 531–547. Springer International Publishing, Cham, 2020. `doi:10.1007/978-3-030-58475-7_31`.

19 Michel X. Goemans and Thomas Rothvoss. Polynomiality for bin packing with a constant number of item types. *J. ACM*, 67(6):38:1–38:21, 2020. `doi:10.1145/3421750`.

20 Wen-Mei W Hwu, David B Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022. URL: `https://shop.elsevier.com/books/programming-massively-parallel-processors/hwu/978-0-323-91231-0`.

21 Klaus Jansen, Stefan Kratsch, Dániel Marx, and Ildikó Schlotter. Bin packing with fixed number of bins revisited. *J. Comput. Syst. Sci.*, 79(1):39–49, 2013. `doi:10.1016/j.jcss.2012.04.004`.

22 Richard E. Korf. A new algorithm for optimal bin packing. In Rina Dechter and Michael J. Kearns andRichard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 731–736. AAAI Press / The MIT Press, 2002. URL: `http://www.aaai.org/Library/AAAI/2002/aaai02-110.php`.

23 Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977. `doi:10.1016/0004-3702(77)90007-8`.

24 Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

25 Silvano Martello and Paolo Toth. Lower bounds and reduction procedures for the bin packing problem. *Discret. Appl. Math.*, 28:59–70, 1990. `doi:10.1016/0166-218X(90)90094-S`.

26 L. Michel, P. Schaus, and P. Van Hentenryck. MiniCP: a lightweight solver for constraint programming. *Mathematical Programming Computation*, pages 133–184, 2021. `doi:10.1007/s12532-020-00190-7`.

27 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. `doi:10.1007/978-3-540-74970-7_38`.

28 Nvidia Team. CUDA. URL: `https://developer.nvidia.com/cuda-toolkit`.

29 OpenMP Team. OpenMP. URL: `https://www.openmp.org/`.

30 François Pelsser, Pierre Schaus, and Jean-Charles Régin. Revisiting the cardinality reasoning for binpacking constraint. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 578–586. Springer, 2013. `doi:10.1007/978-3-642-40627-0_43`.

31 Jürgen Rietz, Cláudio Alves, and José M. Valério de Carvalho. Theoretical investigations on maximal dual feasible functions. *Oper. Res. Lett.*, 38(3):174–178, 2010. `doi:10.1016/J.ORL.2010.01.002`.

32 Jürgen Rietz, Cláudio Alves, and José M. Valério de Carvalho. Worst-case analysis of maximal dual feasible functions. *Optim. Lett.*, 6(8):1687–1705, 2012. `doi:10.1007/S11590-011-0359-2`.

33 Khadija Hadj Salem and Yann Kieffer. An experimental study on symmetry breaking constraints impact for the one dimensional bin-packing problem. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2020 Federated Conference on Computer Science and Information Systems, FedCSIS 2020, Sofia, Bulgaria, September 6-9, 2020*, volume 21 of *Annals of Computer Science and Information Systems*, pages 317–326, 2020. `doi:10.15439/2020F19`.

34 Pierre Schaus. *Solving Balancing and Bin-Packing problems with Constraint Programming.* PhD thesis, University of Louvain, 2009. URL: `http://cp2013.a4cp.org/sites/default/files/pierre_schaus_-_mr.pdf`.

35 Pierre Schaus, Jean-Charles Régin, Rowan Van Schaeren, Wout Dullaert, and Birger Raa. Cardinality reasoning for bin-packing constraint: Application to a tank allocation problem. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 815–822. Springer, 2012. `doi:10.1007/978-3-642-33558-7_58`.

36 Guntram Scheithauer. *Introduction to Cutting and Packing Optimization.* Springer International Publishing, 2018. `doi:10.1007/978-3-319-64403-5`.

37 Armin Scholl, Robert Klein, and Christian Jürgens. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Comput. Oper. Res.*, 24(7):627–645, 1997. `doi:10.1016/S0305-0548(96)00082-2`.

38 Paul Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming – CP 2004*, pages 648–662. Springer Berlin Heidelberg, 2004. `doi:10.1007/978-3-540-30201-8_47`.

39 Fabio Tardivo. Fzn-MiniCPP. URL: `https://bitbucket.org/constraint-programming/fzn-minicpp`.

40 Fabio Tardivo. MiniCPP-Benchmarks. URL: `https://bitbucket.org/constraint-programming/minicpp-benchmarks`.

41 Fabio Tardivo, Agostino Dovier, Andrea Formisano, Laurent Michel, and Enrico Pontelli. Constraint propagation on GPU: A case study for the AllDifferent constraint. *Journal of Logic and Computation*, page exad033, June 2023. `doi:10.1093/logcom/exad033`.

42 Fabio Tardivo, Agostino Dovier, Andrea Formisano, Laurent Michel, and Enrico Pontelli. Constraint propagation on GPU: A case study for the cumulative constraint. In André A. Ciré, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 20th International Conference, CPAIOR 2023, Nice, France, May 29 - June 1, 2023, Proceedings*, volume 13884 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2023. `doi:10.1007/978-3-031-33271-5_22`.

43 Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Ann. Oper. Res.*, 118(1-4):73–84, 2003. `doi:10.1023/A:1021801522545`.