CP for Bin Packing with Multi-core and GPUs

Fabio Tardivo[1] Laurent Michel[2] Enrico Pontelli[1]

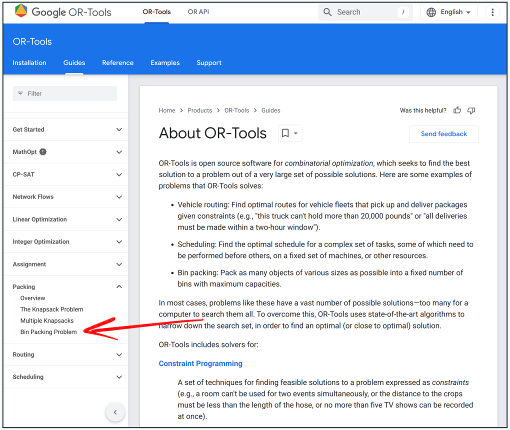
[1] New Mexico State University

[2] University of Connecticut

Motivation

Motivation – Bin Packing Problem

Packing items is a crucial problem in many applications.



About OR-Tools

OR-Tools is open source software for *combinatorial optimization*, which seeks to find the best solution to a problem out of a very large set of possible solutions. Here are some examples of problems that OR-Tools solves:

- Vehicle routing: Find optimal routes for vehicle fleets that pick up and deliver packages given constraints (e.g., "this truck can't hold more than 20,000 pounds" or "all deliveries must be made within a two-hour window").
- Scheduling: Find the optimal schedule for a complex set of tasks, some of which need to be performed before others, on a fixed set of machines, or other resources.
- Bin packing: Pack as many objects of various sizes as possible into a fixed number of bins with maximum capacities.

In most cases, problems like these have a vast number of possible solutions—too many for a computer to search them all. To overcome this, OR-Tools uses state-of-the-art algorithms to narrow down the search set, in order to find an optimal (or close to optimal) solution.

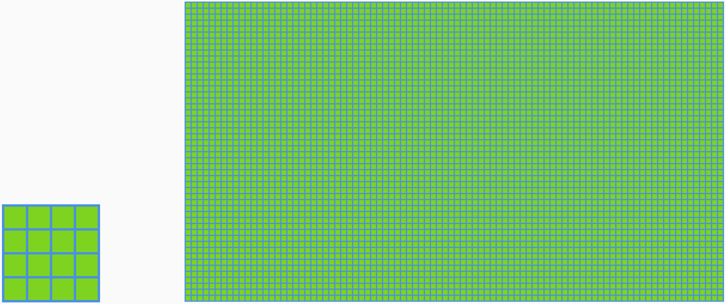
OR-Tools includes solvers for:

Constraint Programming

A set of techniques for finding feasible solutions to a problem expressed as *constraints* (e.g., a room can't be used for two events simultaneously, or the distance to the crops must be less than the length of the hose, or no more than five TV shows can be recorded at once).

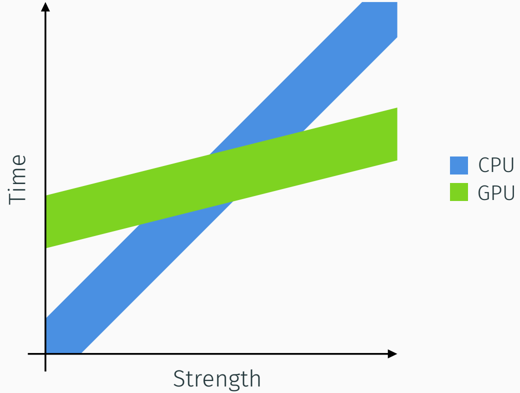
Motivation – GPU

Cores difference between CPU and GPU.



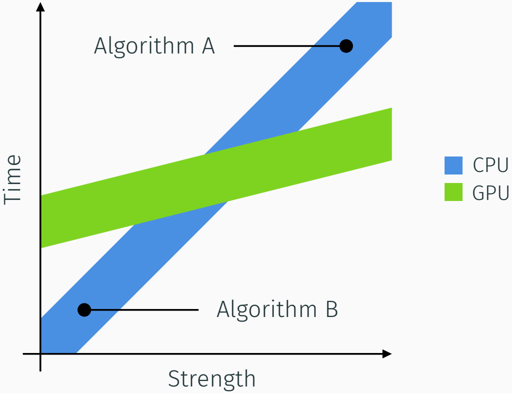
Motivation – GPU

Different trade-off propagation strength vs propagation time.



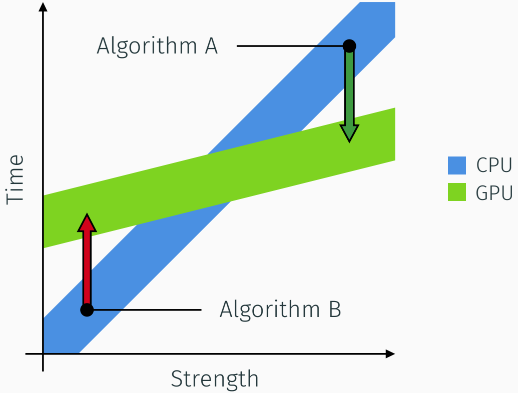
Motivation – GPU

Different trade-off propagation strength vs propagation time.



Motivation – GPU

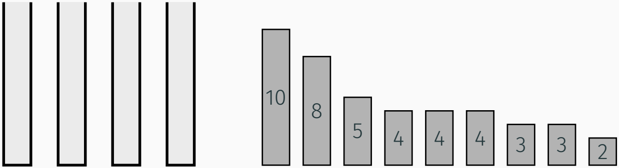
Different trade-off propagation strength vs propagation time.



Introduction

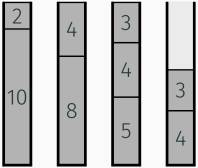
Introduction – Bin Packing Problem

The Bin Packing Problem (BPP) involves packing n items with weights $W = [w_1, w_2, \ldots, w_n]$ into the fewest bins of capacity c.



Introduction – Bin Packing Problem

The Bin Packing Problem (BPP) involves packing n items with weights $W = [w_1, w_2, \ldots, w_n]$ into the fewest bins of capacity c.



Introduction – BinPacking Constraint

An instance with n items of weights $W = [w_1, w_2, \ldots, w_n]$, and k bins of capacity c is modeled as:

$$\begin{aligned} x_i &= \{1, \ldots, k\} \qquad & i = 1, \ldots, n \\ l_j &= \{0, \ldots, c\} \qquad & j = 1, \ldots, k \end{aligned}$$

$$BinPacking(W = [w_1, \ldots, w_n], X = [x_1, \ldots, x_n], L = [l_1, \ldots, l_k])$$

A Constraint for Bin Packing, Shaw, 2004

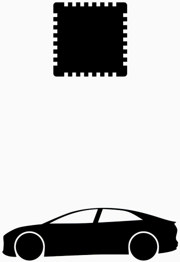
Introduction – BinPacking Constraint

Simplified propagation algorithm for the BinPacking constraint:

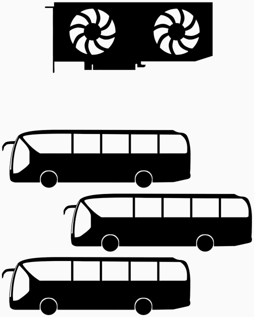
```
Procedure: $propagate(c, W, k, X, L)$
for $j \leftarrow 1$ to $k$ do
    $doLoadCoherence(j, X, W, L)$
    $doLoadTightening(j, X, W, L)$
    for $i \in \{i \mid j \in x_i \land |x_i| > 1\}$ do
        $doItemEliminationCommitment(i, j, X, W, L)$
if $getLowerBound(c, W, k, X) > k$ then $Fail$
```

A Constraint for Bin Packing, Shaw, 2004

Introduction – GPU Programming



- Fast.
- Few people/tasks.



- Slow.
- Many people/tasks.

Introduction – GPU Programming

GPU execution model is known as Single-Instruction Multiple-Thread:

- Each thread executes the same C/C++ function, called a kernel.
- Each thread has a unique index used for data access and control flow.

Programming Massively Parallel Processors: A Hands-on Approach, Wen-Mei et al., 2022

Introduction – GPU Programming

GPU execution model is known as Single-Instruction Multiple-Thread:

- Each thread executes the same C/C++ function, called a kernel.
- Each thread has a unique index used for data access and control flow.

Sequential combination of the arrays *A* and *B* into *C*:

```
Procedure Main(Args)
    $n \leftarrow \ldots$
    $A \leftarrow \ldots$
    ... // Initialization

    combineArrays(n, A, B, C)
```

```
Procedure combineArrays(n, A, B, C)
    for $i \leftarrow 0$ to $n - 1$ do
        if $i \bmod 2 = 0$ then
            $C[i] \leftarrow C[i] + f(A[i], B[i])$
        else
            $C[i] \leftarrow C[i] + g(A[i], B[i])$
```

Programming Massively Parallel Processors: A Hands-on Approach, Wen-Mei et al., 2022

Introduction – GPU Programming

GPU execution model is known as Single-Instruction Multiple-Thread:

- Each thread executes the same C/C++ function, called a kernel.
- Each thread has a unique index used for data access and control flow.

Parallel combination of the arrays *A* and *B* into *C*:

```
Procedure $Main(Args)$
    $n \leftarrow \dots$
    $A \leftarrow \dots$
    $\dots$ // Initialization

    $nThreads \leftarrow n$
    $memcpyCpuToGpuAsync(n, A, B, C)$
    $launchKernelAsync(combineArraysKernel, nThreads, [n, A, B, C])$
    $memcpyGpuToCpuAsync(C)$
    $waitGpu()$
```

```
Procedure $combineArraysKernel(n, A, B, C)$
    $i \leftarrow getThreadIdx()$ // n threads
    if $i \bmod 2 = 0$ then
        $C[i] \leftarrow C[i] + f(A[i], B[i])$
    else
        $C[i] \leftarrow C[i] + g(A[i], B[i])$
```

Programming Massively Parallel Processors: A Hands-on Approach, Wen-Mei et al., 2022

Introduction – GPU Programming

GPU execution model is known as Single-Instruction Multiple-Thread:

- Each thread executes the same C/C++ function, called a kernel.
- Each thread has a unique index used for data access and control flow.

Parallel combination of the arrays *A* and *B* into *C*:

```
Procedure combineArraysKernel($n$, $A$, $B$, $C$)
    $i \leftarrow$ getThreadIdx() // n threads
    if $i \bmod 2 = 0$ then
        $C[i] \leftarrow C[i] + f(A[i], B[i])$
    else
        $C[i] \leftarrow C[i] + g(A[i], B[i])$
```

Programming Massively Parallel Processors: A Hands-on Approach, Wen-Mei et al., 2022

Introduction – GPU Programming

GPU execution model is known as Single-Instruction Multiple-Thread:

- Each thread executes the same C/C++ function, called a kernel.
- Each thread has a unique index used for data access and control flow.

Parallel combination of the arrays *A* and *B* into *C*:

```
Procedure combineArraysKernel($n, A, B, C$)
    $i \leftarrow$ getThreadIdx() // n threads
    if $i \bmod 2 = 0$ then
        $C[i] \leftarrow C[i] + f(A[i], B[i])$
    else
        $C[i] \leftarrow C[i] + g(A[i], B[i])$
```

Programming Massively Parallel Processors: A Hands-on Approach, Wen-Mei et al., 2022

Introduction – GPU Programming

GPU execution model is known as Single-Instruction Multiple-Thread:

- Each thread executes the same C/C++ function, called a kernel.
- Each thread has a unique index used for data access and control flow.

Parallel combination of the arrays *A* and *B* into *C*:

```
Procedure Main(Args)
    $n \leftarrow \dots$
    $A \leftarrow \dots$
    $\dots$ // Initialization

    $nThreads \leftarrow \frac{n}{2}$
    $memcpyCpuToGpuAsync(n, A, B)$
    $launchKernelAsync(combineArraysKernelEven, nThreads, \dots)$
    $launchKernelAsync(combineArraysKernelOdd, nThreads, \dots)$
    $memcpyGpuToCpuAsync(C)$
    $waitGpu()$
```

```
Procedure combineArraysKernelEven(n, A, B, C)
    $i \leftarrow getThreadIdx()$ // n/2 threads
    $j \leftarrow 2i$
    $C[j] \leftarrow C[i] + f(A[j], B[j])$

Procedure combineArraysKernelOdd(n, A, B, C)
    $i \leftarrow getThreadIdx()$ // n/2 threads
    $j \leftarrow 1 + 2i$
    $C[j] \leftarrow C[i] + g(A[j], B[j])$
```

Programming Massively Parallel Processors: A Hands-on Approach, Wen-Mei et al., 2022

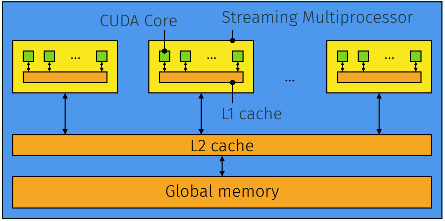
Introduction – GPU Programming

Simplified GPU architecture:

- Streaming Multiprocessors
- CUDA Cores
- L1 Cache
- L2 Cache
- Global Memory

Memory latency in clock cycles:

- L1 cache: 30
- L2 cache: 260 (8x slower)
- Global memory: 470 (15x slower)



Atomic operations:

- Min, Max, And, Or, ...
- Block other access to the memory

Programming Massively Parallel Processors: A Hands-on Approach, Wen-Mei et al., 2022

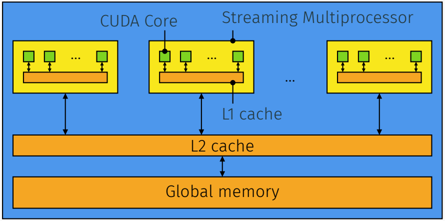
Introduction – GPU Programming

NVIDIA GeForce RTX 3080 specifications:

- Streaming Multiprocessors: 68
- CUDA Cores: 128 at 1.4 GHz

} Total of **8704** threads!

- L1 Cache: 128 KB
- L2 Cache: 5 MB
- Global Memory: 10 GB



Memory latency in clock cycles:

- L1 cache: 30
- L2 cache: 260 (8x slower)
- Global memory: 470 (15x slower)

Atomic operations:

- Min, Max, And, Or, ...
- Block other access to the memory

Programming Massively Parallel Processors: A Hands-on Approach, Wen-Mei et al., 2022

Introduction – GPU Programming

GPU acceleration enhances a wide range of fields:

- Machine Learning: PyTorch, TensorFlow, cuDNN, ...
- Numerical Analysis: MATLAB, cuBLAS, cuFFT, ...
- Scientific Simulation: GROMACS, ANSYS Fluent, Qiskit, ...
- Constraint Programming: **None**.

Constraint Programming has specific requirements, so it is necessary:

- Build **tailored** software from scratch.
- Make it **transparent** to the user.

https://www.nvidia.com/en-us/accelerated-applications

Design

Design – Analysis

Simplified propagation algorithm for the BinPacking constraint

```
Procedure: $propagate(c, W, k, X, L)$
for $j \leftarrow 1$ to $k$ do
    $doLoadCoherence(j, X, W, L)$
    $doLoadTightening(j, X, W, L)$
    for $i \in \{i \mid j \in x_i \wedge |x_i| > 1\}$ do
        $doItemEliminationCommitment(i, j, X, W, L)$
if $getLowerBound(c, W, k, X) > k$ then $Fail$
```

A Constraint for Bin Packing, Shaw, 2004

Design – Stronger pruning

Two prominent approaches to obtain a tighter lower bound:

- Linear relaxation of the strong Arc-Flow model.
- Enhance the weak L_1 bound with a Dual Feasible Function (DFF).

Bin packing and cutting stock problems: Mathematical models and exact algorithms, Delorme et al., 2016

Design – Stronger pruning

Given an instance with weights W and capacity c, the L_1 bound is:

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

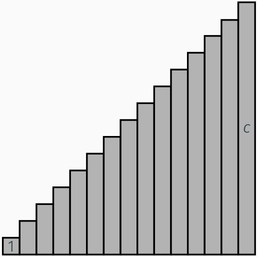
A function $f : \mathbb{N}_0 \to \mathbb{N}_0$ is **dual feasible** if, for every $W_S \subseteq W$:

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

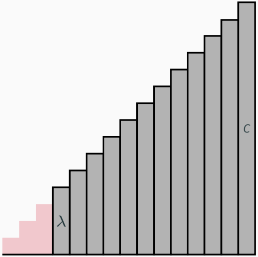
The combined lower bound L_f is:

$$L_f(c, W) = \left\lceil \frac{1}{f(c)} \sum_{w \in W} f(w) \right\rceil$$

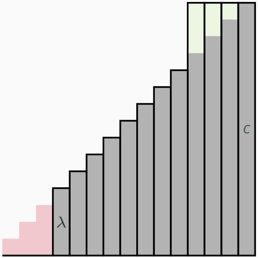
Design – Stronger pruning



Design – Stronger pruning



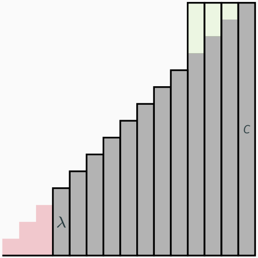
Design – Stronger pruning



Design – Stronger pruning

A basic DFF is f_{MT} depending on an integer parameter $0 \leq \lambda \leq \frac{c}{2}$:

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

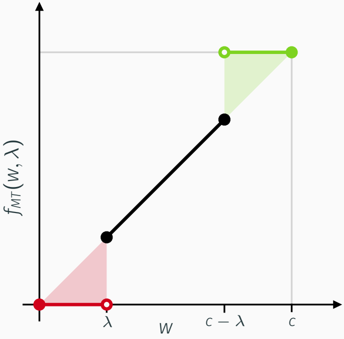


New classes of fast lower bounds for bin packing problems, Fekete et al., 2001

Design – Stronger pruning

A basic DFF is f_{MT} depending on an integer parameter $0 \leq \lambda \leq \frac{c}{2}$:

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



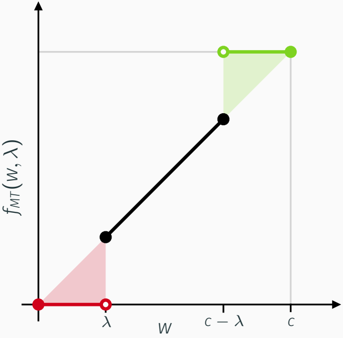
New classes of fast lower bounds for bin packing problems, Fekete et al., 2001

Design – Stronger pruning

A basic DFF is f_{MT} depending on an integer parameter $0 \leq \lambda \leq \frac{c}{2}$:

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

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



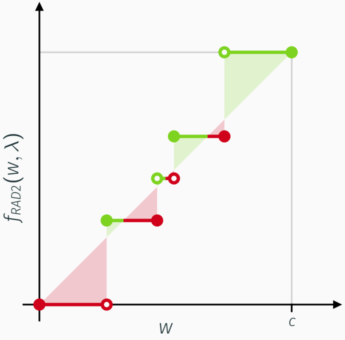
New classes of fast lower bounds for bin packing problems, Fekete et al., 2001

Design – Stronger pruning

Another basic DFF is f_{RAD2} with parameter $\frac{c}{4} < \lambda \leq \frac{c}{3}$:

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

$$L_{RAD2}(c,W) = \max_{\frac{c}{4}<\lambda\leq\frac{c}{3}} \left\lceil \frac{1}{f_{RAD2}(c,\lambda)} \sum_{w\in W} f_{RAD2}(w,\lambda) \right\rceil$$



Theoretical investigations on maximal dual feasible functions, Rietz et al., 2010

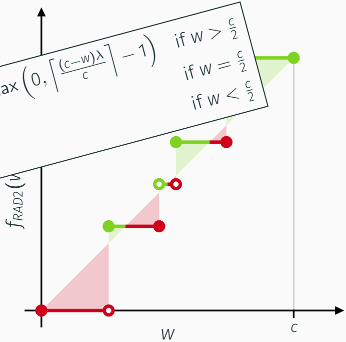
Design – Stronger pruning

Another basic DFF is f_{RAD2} with parameter $\frac{c}{4} < \lambda \leq \frac{c}{3}$:

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

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

$$L_{RAD2}(c,W) = \max_{\frac{c}{4}<\lambda\leq\frac{c}{3}} \left\lceil \frac{1}{f_{RAD2}(c,\lambda)} \sum_{w\in W} f_{RAD2}(w,\lambda) \right\rceil$$



A survey of dual-feasible and superadditive functions, Clautiaux et al., 2010

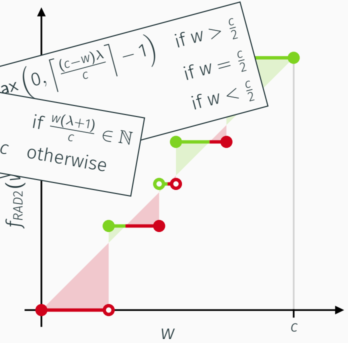
Another basic DFF is f_{RAD2} with parameter $\frac{c}{4} < \lambda \leq \frac{c}{3}$:

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

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

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

$$L_{RAD2}(c,W) = \max_{\frac{c}{4}<\lambda\leq\frac{c}{3}} \left\lceil \frac{1}{f_{RAD2}(c,\lambda)} \sum_{w\in W} f_{RAD2}(w,\lambda) \right\rceil$$



A survey of dual-feasible and superadditive functions, Clautiaux et al., 2010

Design – Stronger pruning

Another basic DFF is f_{RAD2} with parameter $\frac{c}{4} < \lambda \leq \frac{c}{3}$:

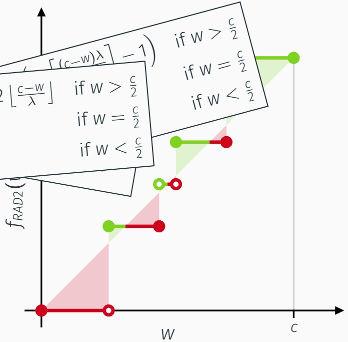
$$f_{\substack{RAD2\\ \frac{c}{4}<\lambda\leq\frac{c}{3}}}(w,\lambda) = \begin{cases} 0 \\ \left\lfloor \frac{c}{3} \right\rfloor \\ \left\lfloor \frac{c}{2} \right\rfloor \\ \end{cases}$$

$$\left\lceil \frac{(c-w)\lambda}{} \right\rceil - 1\Big) \quad \begin{aligned} &\text{if } w > \tfrac{c}{2} \\ &\text{if } w = \tfrac{c}{2} \\ &\text{if } w < \tfrac{c}{2} \end{aligned}$$

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

$$f_{\substack{VB2\\ 2\leq\lambda\leq c}}(w,\lambda) = \begin{cases} \\ 2\max(0, \end{cases}$$

$$L_{RAD2}(c,W) = \max_{\frac{c}{4}<\lambda\leq\frac{c}{3}} \left\lceil \frac{1}{f_{RAD2}(c,\lambda)} \sum_{w\in W} f_{RAD2}(w,\lambda) \right\rceil$$



A survey of dual-feasible and superadditive functions, Clautiaux et al., 2010

Design – Stronger pruning

Another basic DFF is f_{RAD2} with parameter $\frac{c}{4} < \lambda \le \frac{c}{3}$:

$$\underset{\frac{c}{4}<\lambda\le\frac{c}{3}}{f_{RAD2}(w,\lambda)} =$$

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

$$\underset{2\le\lambda\le c}{f_{VB2}(w,\lambda)} =$$

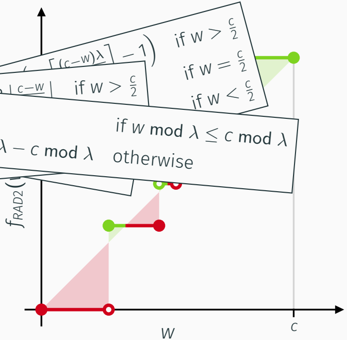
$1\le\lambda\le\frac{c}{2}$ … $2\max(0, \dots$ … $\left\lfloor \frac{c-w}{\lambda} \right\rfloor$ if $w > \frac{c}{2}$

$\left\lceil \frac{(c-w)\lambda}{c} \right\rceil - 1\Big)$ if $w > \frac{c}{2}$

if $w = \frac{c}{2}$

if $w < \frac{c}{2}$

$$L_{RAD2}(c,W) = \max_{\frac{c}{4}<\lambda\le\frac{c}{3}} \left\lceil \frac{1}{f_{RAD2}(c,\lambda)} \sum_{w\in W} f_{RAD2}(w,\lambda) \right\rceil$$



A survey of dual-feasible and superadditive functions, Clautiaux et al., 2010

Design – Stronger pruning

Which DFF is the best?

Lower bound	Total Optimal	Only Optimal
L_{CCM1}	1219	40
L_{MT}	1151	2
L_{BJ1}	1101	47
L_{VB2}	973	1
L_{FS1}	742	2
L_{RAD2}	189	10

None dominate.

Design – Stronger pruning

Which DFF is the best?

Lower bound	Total Optimal	Only Optimal
L_{CCM1}	1219	40
L_{MT}	1151	2
L_{BJ1}	1101	47
L_{VB2}	973	1
L_{FS1}	742	2
L_{RAD2}	189	10

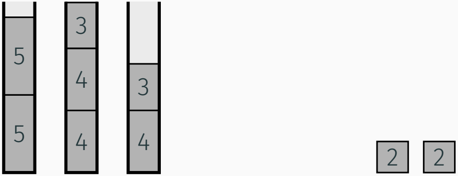
Use the **GPU** to calculate **all** the lower bounds in **parallel**!

Design – Stronger pruning

Lower bounds apply to instances, while search handles partial packings.

Reduce a partial packing to an “equivalent” instance.

The basic reduction is R_0, but R_{Min} and R_{Max} are generally more accurate.



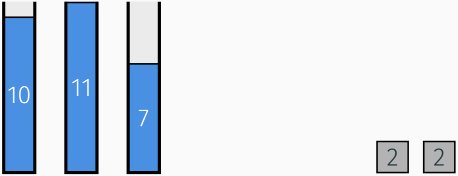
Consistency Check for the Bin Packing Constraint Revisited, Dupuis et al., 2010

Design – Stronger pruning

Lower bounds apply to instances, while search handles partial packings.

Reduce a partial packing to an “equivalent” instance.

The basic reduction is R_0, but R_{Min} and R_{Max} are generally more accurate.



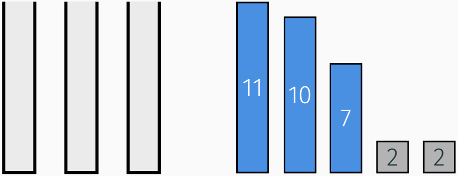
Consistency Check for the Bin Packing Constraint Revisited, Dupuis et al., 2010

Design – Stronger pruning

Lower bounds apply to instances, while search handles partial packings.

Reduce a partial packing to an “equivalent” instance.

The basic reduction is R_0, but R_{Min} and R_{Max} are generally more accurate.



Consistency Check for the Bin Packing Constraint Revisited, Dupuis et al., 2010

Design – Stronger pruning

Sequential calculation of the DFFs-based lower bound:

```
Function: $getLowerBound(c, W, k, X) \rightarrow lb$
$lb \leftarrow 0$
for $R \in \{R_0, R_{Min}, R_{Max}\}$ do
    $(c_R, W_R) \leftarrow R(c, W, X)$
    for $f \in \{f_{CCM1}, f_{MT}, f_{BJ1}, f_{VB2}, f_{FS1}, f_{RAD2}\}$ do
        $(\underline{\lambda}, \overline{\lambda}) \leftarrow getParametersMinMax(f, c_R)$
        $L_f \leftarrow 0$
        for $\lambda \leftarrow \underline{\lambda}$ to $\overline{\lambda}$ do
            $L_f \leftarrow max\left(L_f, \left\lceil \frac{1}{f(c,\lambda)} \sum_{w \in W} f(w, \lambda) \right\rceil\right)$
        $lb \leftarrow max(lb, L_f)$
        if $lb > k$ then return $lb$
return $lb$
```

Design – Stronger pruning

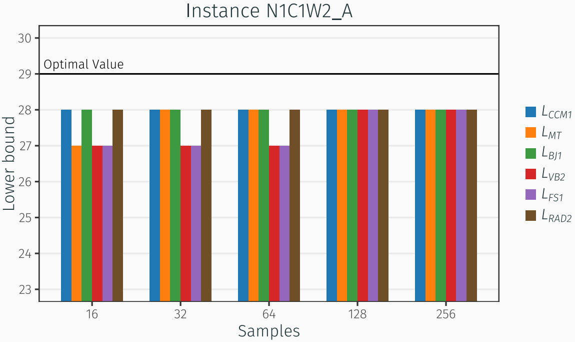
Sequential calculation of the DFFs-based lower bound:

```
Function: $getLowerBound(c, W, k, X) \rightarrow lb$
$lb \leftarrow 0$
for $R \in \{R_0, R_{Min}, R_{Max}\}$ do
    $(c_R, W_R) \leftarrow R(c, W, X)$
    for $f \in \{f_{CCM1}, f_{MT}, f_{BJ1}, f_{VB2}, f_{FS1}, f_{RAD2}\}$ do
        $(\underline{\lambda}, \overline{\lambda}) \leftarrow getParametersMinMax(f, c_R)$
        $L_f \leftarrow 0$
        for $\lambda \leftarrow \underline{\lambda}$ to $\overline{\lambda}$ do
            $L_f \leftarrow max\left(L_f, \left\lceil \frac{1}{f(c,\lambda)} \sum_{w \in W} f(w, \lambda) \right\rceil\right)$
        $lb \leftarrow max(lb, L_f)$
        if $lb > k$ then return $lb$
return $lb$
```

- Independent iterations.
- Fits in L1 cache.
- No if/else branches.
- Few atomics.

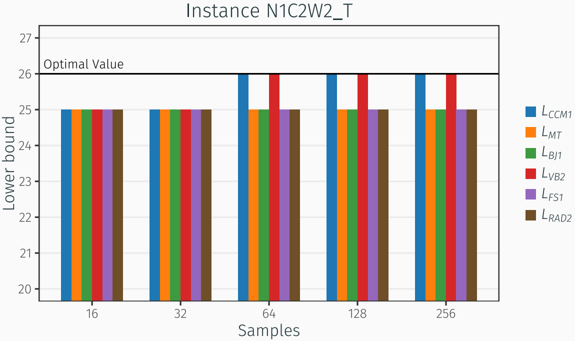
Design – Stronger pruning

Lower bounds for different sampling of λ values:



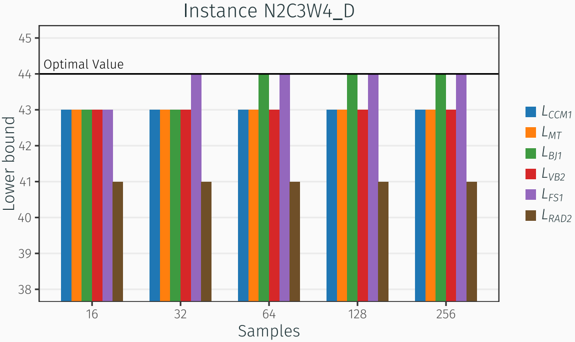
Design – Stronger pruning

Lower bounds for different sampling of λ values:



Design – Stronger pruning

Lower bounds for different sampling of λ values:



Results

Results

We used various BPP benchmarks from the literature to compare:

- Standard lower bound (L2).
- Linear relaxation of the Arc-Flow model (Arc-Flow)[1].
- Sampled sequential implementation (DFFs-Seq-CPU).
- Sampled multithread implementation (DFFs-Par-CPU).
- Complete GPU implementation (DFFs-GPU).

---

[1] Propagating the Bin Packing Constraint Using Linear Programming, Cambazard et al., 2010

Results

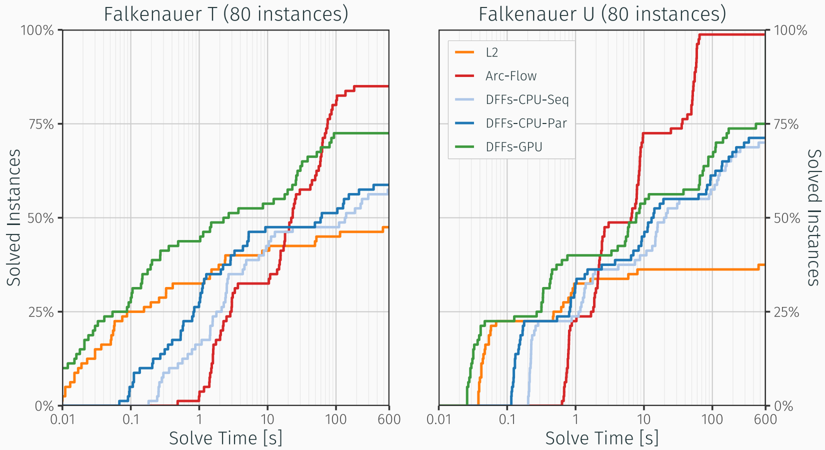
Model and search setup:

- Add one more bin until solution is found.
- Decreasing Best Fit heuristic.
- Symmetry breaking and dominance rules.
- 10 minutes timeout.

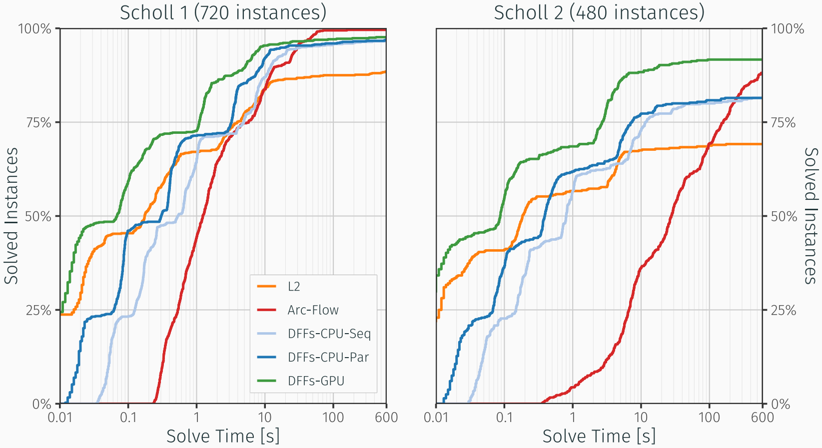
Benchmark system:

- Intel Core i7-10700K (8 Cores at 3.8 GHz).
- NVIDIA GeForce RTX 3080 (8704 CUDA Cores at 1.4 GHz).
- 32 GB RAM.
- Ubuntu Linux 22.04 LTS.
- CUDA 11.8.
- CPLEX 22.1 (for Arc-Flow).

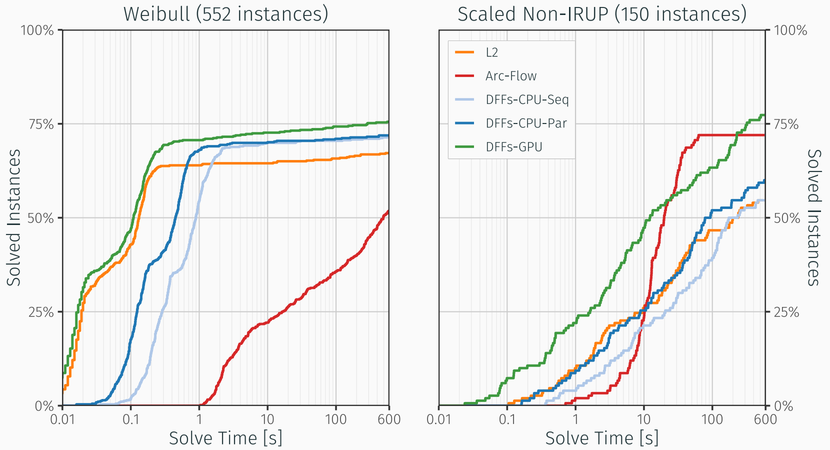
Results


Falkenauer T (80 instances)
Falkenauer U (80 instances)
Solved Instances
Solved Instances
Solve Time [s]
Solve Time [s]
100%
75%
50%
25%
0%
0.01
0.1
1
10
100
600
L2
Arc-Flow
DFFs-CPU-Seq
DFFs-CPU-Par
DFFs-GPU

Results


Scholl 1 (720 instances)
Scholl 2 (480 instances)
Solved Instances
Solve Time [s]
0%
25%
50%
75%
100%
0.01
0.1
1
10
100
600
L2
Arc-Flow
DFFs-CPU-Seq
DFFs-CPU-Par
DFFs-GPU

Results


Weibull (552 instances)
Scaled Non-IRUP (150 instances)
Solved Instances
Solve Time [s]
100%
75%
50%
25%
0%
0.01
0.1
1
10
100
600
L2
Arc-Flow
DFFs-CPU-Seq
DFFs-CPU-Par
DFFs-GPU

Conclusion

Conclusion – Code

Code repository:

`https://bitbucket.org/constraint-programming/bpp-minicpp`
`https://bitbucket.org/constraint-programming/fzn-minicpp`

Transparent GPU utilization in MiniZinc (▶):

```
include "globals.mzn";
include "minicpp.mzn";

int: n_items;                                 % Number of items
int: capacity;                                % Capacity of each bin
array [1..n_items] of int: weights;  % Weights of items
...

constraint bin_packing(capacity, items_bin, weights) ::gpu;
...
```

Conclusions

BinPacking constraint:

- Novel trade-off pruning time vs pruning strength.
- Applicable to 2D, 3D, and Vector Packing Problems.

GPU in Constraint Programming:

- GPU can also be utilized for pruning.
- It is essential to **rethink algorithmic design** choices.