GPU for Mixed Integer Optimization

Computational Integer Optimization · SoSe 2026

Gennesaret Tjusila

Modes of computing

- In this course, we want to solve MIPs as fast as possible.
- Our method for doing this is branch-and-cut.
- Assuming the algorithm is fixed, its speed is determined by how fast we can push the whole computation through the CPU.
- We say that our computation is latency-oriented.

Modes of computing

- Until the mid-2000s, single-core CPU performance increased rapidly. Since then, improvements have slowed substantially because of power and thermal constraints.
- Manufacturers switched to multicore CPUs. The problem: we need algorithms that can exploit them.
- Parallel branch-and-cut has been developed; however, its benefit diminishes as the number of threads increases.
- A multicore CPU replicates a latency-oriented design so that multiple cores can run at once.
- What if we care less about latency and instead maximize how many parallel cores we can run at once? Enter GPUs.

Latency versus throughput

CPU: minimize latency

- Execute each instruction as quickly as possible.
- Sophisticated mechanisms support this, e.g. branch prediction.
- Deliver operands quickly to a few powerful cores.

GPU: maximize throughput

- Use simpler mechanisms to accommodate many cores under the same power and thermal limits.
- Deliver operands to many cores at approximately the same time.
- Hence the emphasis on memory bandwidth: how much data can pass through memory at once?

A quote that did not age too well

Up to 2023, the Gurobi support site stated:[1]

> *"The Gurobi development team is watching GPUs (Graphics Processing Units) closely, but up to this point, all of the evidence indicates that they aren't well suited to the needs of an LP/MIP/QP solver. Specifically:*
> *GPUs don't work well for sparse linear algebra, which dominates much of linear programming. GPUs rely on keeping hundreds or even thousands of independent processors busy at a time. The extremely sparse matrices that are typical in linear programming don't admit nearly that level of parallelism.*
> *GPUs are built around SIMD computations, where all processors perform the same instruction in each cycle (but on different data). Parallel MIP explores different sections of the search tree on different processors. The computations required at different nodes in the search tree are quite different, so SIMD computation is not well suited to the needs of parallel MIP.*
> *Note that CPUs and GPUs are both improving parallelism as a means to increase performance. The Gurobi Optimizer is designed to effectively exploit multiple cores in a CPU, so you'll definitely see a benefit from more parallelism in the future."*

[1] Gurobi, "Does Gurobi support GPUs?" Gurobi Optimization, LLC2025].

What changed?

- Traditional LP algorithms used within MIP solvers—simplex and barrier—are factorization-based. Sparse factorization is generally not GPU-friendly. We will shortly see why.
- GPU branch-and-bound had not become viable for general-purpose MIP solving.
- Historically, many GPUs prioritized FP32 throughput, whereas MIP solvers rely on reliable FP64 computation.

SIMT: how a GPU executes code

A GPU uses a **Single Instruction, Multiple Threads** execution model: many threads execute the *same instruction* in lockstep, each operating on *different values* in its registers.

Example: vector addition $C = A + B$. One thread per array entry.

thread	A	B	instruction	C	
t_1	3	7	`ADD A, B -> C`	10	
t_2	1	4	`ADD A, B -> C`	5	
t_3	8	2	`ADD A, B -> C`	10	
t_4	5	5	`ADD A, B -> C`	10	
t_5	6	2	`ADD A, B -> C`	3	*masked*

Each thread loads $A[i], B[i]$ into its registers, executes the same `ADD`, and stores $C[i]$. **Threads are bundled into units of 32 called warps that execute instructions together.**

Scale: an NVIDIA H100 (SXM5) has 132 SMs $\times$ 128 FP32 cores $=$ **16,896 FP32 operations issued per cycle**.

Branch divergence

If threads in one warp take different branches, the warp executes one branch at a time.

- Threads taking the current branch remain active.
- The other threads are temporarily masked.
- With four different branches, the warp may need four passes.

Example: vector addition on a GPU

```
__global__ void vector_add(
    const float* A, const float* B, float* C, int size)
{
  int idx = blockDim.x * blockIdx.x + threadIdx.x;
  if (idx < size)
    C[idx] = A[idx] + B[idx];
}

constexpr int n = 5000;
constexpr int threads_per_block = 256;
constexpr int num_blocks =
    (n + threads_per_block - 1) / threads_per_block;

vector_add<<<num_blocks, threads_per_block>>>(d_a, d_b, d_c, n);
```

- The launch creates enough threads to assign one thread to each entry.
- Each thread computes its own index and performs the same operation.

Moving data to and from the GPU

```
constexpr int bytes_needed = n * sizeof(float);

cudaMalloc(&d_a, bytes_needed);
cudaMalloc(&d_b, bytes_needed);
cudaMalloc(&d_c, bytes_needed);

cudaMemcpy(d_a, h_a.data(), bytes_needed, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b.data(), bytes_needed, cudaMemcpyHostToDevice);

vector_add<<<num_blocks, threads_per_block>>>(d_a, d_b, d_c, n);

cudaMemcpy(h_c.data(), d_c, bytes_needed, cudaMemcpyDeviceToHost);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

Takeaway: delegating work to a GPU may not be worthwhile if there is too little work to amortize the transfers.

GPU key idea 2: locality

Full GH100 GPU with 144 SMs. Source: NVIDIA Andersch et al.2022].

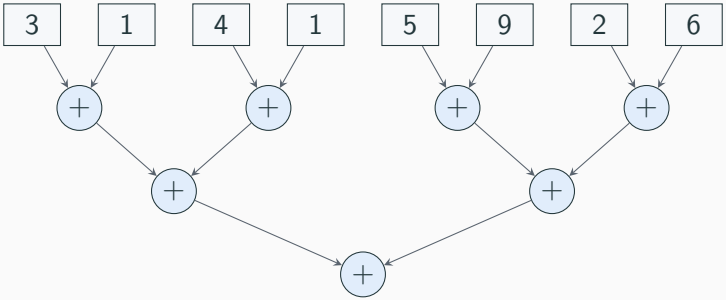
GPU key idea 2: locality

Data can be kept at different levels: registers and shared memory close to the threads, then L2 cache, then global memory.

- Keep data close to the threads that use it.
- Communication is easiest within a warp, then within a thread block, and most expensive across blocks.

GH100 streaming multiprocessor. Source: NVIDIA Andersch et al.2022].

Example: array summation



A tree reduction performs independent additions in parallel and halves the active data at each step.

Example: array summation

```
__global__ void sum_reduction(const float* input, float* result, int size) {
  // Keep this block's data in fast, local shared memory.
  __shared__ float shared[256];

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int tid = threadIdx.x;
  shared[tid] = (idx < size) ? input[idx] : 0.0f;

  // Synchronize threads working on this chunk.
  __syncthreads();

  for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
    if (tid < stride)
      shared[tid] += shared[tid + stride];
    __syncthreads();
  }

  if (tid == 0)
    // Combine block results through a global-memory atomic update.
    atomicAdd(result, shared[0]);
}
```

Why is this important? Sparse matrix–vector multiplication benefits heavily from locality.

GPU key idea 3: coalescing—the good case

- On modern NVIDIA GPUs, global-memory requests are serviced in aligned 32-byte sectors.
- A warp issues the same instruction to 32 threads. Suppose every thread requests one 4-byte float.
- If the threads request consecutive floats, they collectively request bytes 0–127.
- The warp needs only four 32-byte sectors.

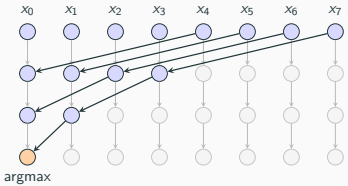
GPU key idea 3: coalescing—the bad case

- Suppose thread 0 requests bytes 0–3, thread 1 bytes 32–35, thread 2 bytes 64–67, and so on.
- Each request lies in a different 32-byte sector.
- The warp now needs 32 sectors instead of four.
- Sparse matrix factorization often produces irregular memory-access patterns, making coalescing difficult.

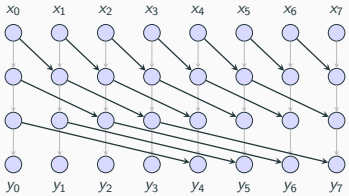
Parallel primitives: reduction, scan, sort

Three GPU primitives we use:

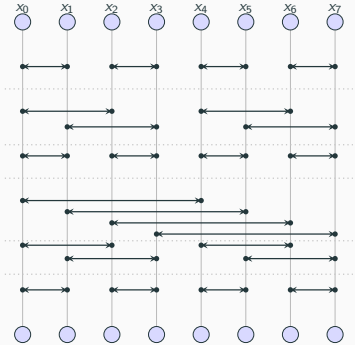
Reduction (argmax)



Scan (prefix sum, $y_j = \sum_{i=0}^{j} x_i$)



Sort



GPU for MIP solving

- Up to 2023, the Gurobi blog wrote:
  "... all of the evidence indicates that [GPUs] aren't well suited to the needs of an LP/MIP/QP solver."
- This changed with first-order methods for LP on GPU (cuPDLPx Lu et al.2025]); the **PDHG** iteration takes the form

$$x^{k+1} = \mathrm{proj}_X(x^k - \tau(c - K^\top y^k))$$
$$y^{k+1} = \mathrm{proj}_Y(y^k + \sigma(q - K(2x^{k+1} - x^k)))$$

  These rely on **sparse matrix–vector products**, heavily optimized on GPUs, though convergence to high precision can be slow.
- **Low-precision LP solutions already suffice to guide primal heuristics** (feasibility pump Mexi et al.2023], fix-and-propagate Kempke and Koch2025]).

GPU for MIP solving (Part II)

- **LP-free primal heuristics** have also improved: **Feasibility Jump** Luteberget and Sartor2023], **Local-MIP** Lin et al.2025], and **fix-propagate-repair** Salvagnin et al.2024].
- **NVIDIA cuOpt** Çördük et al.2025] ported Local-MIP to the GPU, confirming LP-free heuristics are GPU-viable.
- At the NVIDIA-sponsored **Land-Doig MIP Competition 2026**, our solver **CHAP** (a CPU + GPU portfolio with a GPU-native tabu search) won.

GPU Tabu Search

Tabu search: the basic loop

Given an initial iterate $\bar{x}$, repeat until termination:

- **generate** candidate moves on $\bar{x}$, dropping moves on *tabu* variables
- **score** each move and pick $m^\star \leftarrow \arg\max_{m \in \mathcal{M}} \mathrm{score}(m, \bar{x})$
- **apply** $m^\star$ to update $\bar{x}$, and mark the touched variables as tabu
- if $\bar{x}$ is feasible and improves the incumbent, **store** it

Tabu search: the basic loop

Given an initial iterate $\bar{x}$, repeat until termination:

- **generate** candidate moves on $\bar{x}$, dropping moves on *tabu* variables
- **score** each move and pick $m^\star \leftarrow \arg\max_{m \in \mathcal{M}} \mathrm{score}(m, \bar{x})$
- **apply** $m^\star$ to update $\bar{x}$, and mark the touched variables as tabu
- if $\bar{x}$ is feasible and improves the incumbent, **store** it

generate and **score** can be done in parallel.

Scoring function and best shift moves

Scoring. Constraints carry weights $w_1, \ldots, w_m$. To score a move that changes x_j from $\bar{x}_j$ to $\hat{x}_j$, each constraint i contributes a reward/penalty of p_{ij}:

- $\pm w_i$ when it **flips** between satisfied and violated,
- $\pm \frac{1}{2} w_i$ when an already-violated constraint becomes **less / more violated**.

The move's score is the sum over all constraints:

$$s_j(\hat{x}_j, \bar{x}) := \sum_{i=1}^{m} p_{ij}.$$

Best shift moves

Over $D_j := \{x_j : l_j \leq x_j \leq u_j,\ x_j \neq \bar{x}_j,\ x_j \in \mathbb{Z} \text{ if } j \in \mathcal{I}\}$, a best shift move assigns

$$\hat{x}_j \in \arg\max_{x_j \in D_j} s_j(x_j, \bar{x}).$$

Observations.

- As a function of $\hat{x}_j$, the score $s_j(\cdot, \bar{x})$ is a **step function** — D_j reduces to a finite candidate set Luteberget and Sartor2023].
- For **binary** variables, best shift moves are just **flips**.

Evaluating binary flip candidates: a worked example

Four binary variables, incumbent $\bar{x} = (1, 0, 1, 0)$, three knapsack ($\leq$) constraints.

Problem

Constraint	w_i
$C_1: \; 2x_1 + 6x_3 + x_4 \leq 5$	2
$C_2: \; x_1 + 4x_2 \leq 3$	1
$C_3: \; x_3 + 4x_4 \leq 2$	1

Activities $\bar{y} = (8, 1, 1)$; only C_1 violated ($8 > 5$).

Reward p_{ij} for flipping x_j

	x_1 $1\to0$	x_2 $0\to1$	x_3 $1\to0$	x_4 $0\to1$
C_1				
C_2				
C_3				
s_j				

Evaluating binary flip candidates: a worked example

Four binary variables, incumbent $\bar{x} = (1, 0, 1, 0)$, three knapsack ($\leq$) constraints.

Problem

Constraint	w_i
$C_1: \ 2x_1 + 6x_3 + x_4 \leq 5$	2
$C_2: \ x_1 + 4x_2 \leq 3$	1
$C_3: \ x_3 + 4x_4 \leq 2$	1

Activities $\bar{y} = (8, 1, 1)$; only C_1 violated ($8 > 5$).

Reward p_{ij} for flipping x_j

	x_1 1→0	x_2 0→1	x_3 1→0	x_4 0→1
C_1	+1			
C_2				
C_3				
s_j				

Flip x_1:1→0 drops C_1 activity $8 \to 6$: still violated but by less $\Rightarrow +\frac{1}{2}w_1 = +1$.

Evaluating binary flip candidates: a worked example

Four binary variables, incumbent $\bar{x} = (1, 0, 1, 0)$, three knapsack ($\leq$) constraints.

Problem

Constraint	w_i
$C_1: \; 2x_1 + 6x_3 + x_4 \leq 5$	2
$C_2: \; x_1 + 4x_2 \leq 3$	1
$C_3: \; x_3 + 4x_4 \leq 2$	1

Activities $\bar{y} = (8, 1, 1)$; only C_1 violated ($8 > 5$).

Reward p_{ij} for flipping x_j

	x_1 1→0	x_2 0→1	x_3 1→0	x_4 0→1
C_1	+1		+2	
C_2				
C_3				
s_j				

Flip x_3:1→0 drops C_1 activity $8 \to 2$: now satisfied $\Rightarrow$ flip reward $+w_1 = +2$.

Evaluating binary flip candidates: a worked example

Four binary variables, incumbent $\bar{x} = (1, 0, 1, 0)$, three knapsack ($\leq$) constraints.

Problem

Constraint	w_i
$C_1: \; 2x_1 + 6x_3 + x_4 \leq 5$	2
$C_2: \; x_1 + 4x_2 \leq 3$	1
$C_3: \; x_3 + 4x_4 \leq 2$	1

Activities $\bar{y} = (8, 1, 1)$; only C_1 violated ($8 > 5$).

Reward p_{ij} for flipping x_j

	x_1 $1\to0$	x_2 $0\to1$	x_3 $1\to0$	x_4 $0\to1$
C_1	$+1$	$\cdot$	$+2$	-1
C_2	0	-1	$\cdot$	$\cdot$
C_3	$\cdot$	$\cdot$	0	-1
s_j	$+1$	-1	$\mathbf{+2}$	-2

Score each candidate over all its constraints; pick the best.

Evaluating binary flip candidates

For each binary variable j, we want $s_j(1 - \bar{x}_j, \bar{x})$. Iterate over the binary nonzeros of A and *scatter* contributions to per-variable scores.

Input: binary part of A in row-sorted COO form $\{(i, j, a_{ij})\}$
solution bitset $\bar{x}$, weights w_i, residuals $r_i := (A\bar{x})_i - b_i$, rows normalized to $\leq$

Output: s_j for every binary j

1: $s_j \leftarrow 0$ for all binary j
2: **for** each nonzero (i, j, a_{ij}) **in parallel do**
3: $\quad \delta \leftarrow (1 - 2\bar{x}_j)a_{ij}$ {flip changes x_j by ± 1}
4: $\quad r' \leftarrow r_i + \delta$ {residual after flip}
5: $\quad$ compute p_{ij} from $(r_i, r'), w_i$
6: $\quad s_j \mathrel{+}= p_{ij}$ {`atomicAdd`}
7: **end for**
8: **return** s

(a) row-sorted COO $\Rightarrow$ coalesced a_{ij}, r_i, w_i reads

(b) random $\bar{x}_j$ reads $\Rightarrow$ L2-resident bitset

(c) precomputed residuals $\Rightarrow$ one load per nonzero

(d) normalize rows to $\leq \Rightarrow$ fewer cases and less divergence

Evaluating binary flip candidates

For each binary variable j, we want $s_j(1 - \bar{x}_j, \bar{x})$. Iterate over the binary nonzeros of A and *scatter* contributions to per-variable scores.

Input: binary part of A in row-sorted COO form $\{(i, j, a_{ij})\}$
solution bitset $\bar{x}$, weights w_i, residuals $r_i := (A\bar{x})_i - b_i$, rows normalized to $\leq$

Output: s_j for every binary j

1: $s_j \leftarrow 0$ for all binary j
2: **for** each nonzero (i, j, a_{ij}) **in parallel do**
3: $\delta \leftarrow (1 - 2\bar{x}_j)a_{ij}$ {flip changes x_j by ± 1}
4: $r' \leftarrow r_i + \delta$ {residual after flip}
5: compute p_{ij} from $(r_i, r'), w_i$
6: $s_j \mathrel{+}= p_{ij}$ {`atomicAdd`}
7: **end for**
8: **return** s

(a) row-sorted COO $\Rightarrow$ coalesced a_{ij}, r_i, w_i reads

(b) random $\bar{x}_j$ reads $\Rightarrow$ L2-resident bitset

(c) precomputed residuals $\Rightarrow$ one load per nonzero

(d) normalize rows to $\leq \Rightarrow$ fewer cases and less divergence

Evaluating binary flip candidates

For each binary variable j, we want $s_j(1 - \bar{x}_j, \bar{x})$. Iterate over the binary nonzeros of A and *scatter* contributions to per-variable scores.

Input: binary part of A in row-sorted COO form $\{(i, j, a_{ij})\}$
solution bitset $\bar{x}$, weights w_i, residuals $r_i := (A\bar{x})_i - b_i$, rows normalized to $\leq$

Output: s_j for every binary j

1: $s_j \leftarrow 0$ for all binary j
2: **for** each nonzero (i, j, a_{ij}) **in parallel do**
3: $\quad \delta \leftarrow (1 - 2\bar{x}_j)a_{ij}$ {flip changes x_j by ± 1}
4: $\quad r' \leftarrow r_i + \delta$ {residual after flip}
5: $\quad$ compute p_{ij} from $(r_i, r'), w_i$
6: $\quad s_j \mathrel{+}= p_{ij}$ {`atomicAdd`}
7: **end for**
8: **return** s

(a) row-sorted COO $\Rightarrow$ coalesced a_{ij}, r_i, w_i reads

(b) random $\bar{x}_j$ reads $\Rightarrow$ L2-resident bitset

(c) precomputed residuals $\Rightarrow$ one load per nonzero

(d) normalize rows to $\leq$ $\Rightarrow$ fewer cases and less divergence

Evaluating binary flip candidates

For each binary variable j, we want $s_j(1 - \bar{x}_j, \bar{x})$. Iterate over the binary nonzeros of A and *scatter* contributions to per-variable scores.

Input: binary part of A in row-sorted COO form $\{(i, j, a_{ij})\}$
solution bitset $\bar{x}$, weights w_i, residuals $r_i := (A\bar{x})_i - b_i$, rows normalized to $\leq$

Output: s_j for every binary j

1: $s_j \leftarrow 0$ for all binary j
2: **for** each nonzero (i, j, a_{ij}) **in parallel do**
3: $\quad \delta \leftarrow (1 - 2\bar{x}_j)a_{ij}$ {flip changes x_j by ± 1}
4: $\quad r' \leftarrow r_i + \delta$ {residual after flip}
5: $\quad$ compute p_{ij} from $(r_i, r'), w_i$
6: $\quad s_j \mathrel{+}= p_{ij}$ {`atomicAdd`}
7: **end for**
8: **return** s

(a) row-sorted COO $\Rightarrow$ coalesced a_{ij}, r_i, w_i reads

(b) random $\bar{x}_j$ reads $\Rightarrow$ L2-resident bitset

(c) precomputed residuals $\Rightarrow$ one load per nonzero

(d) normalize rows to $\leq \Rightarrow$ fewer cases and less divergence

Evaluating binary flip candidates

For each binary variable j, we want $s_j(1 - \bar{x}_j, \bar{x})$. Iterate over the binary nonzeros of A and *scatter* contributions to per-variable scores.

Input: binary part of A in row-sorted COO form $\{(i, j, a_{ij})\}$
solution bitset $\bar{x}$, weights w_i, residuals $r_i := (A\bar{x})_i - b_i$, rows normalized to $\leq$

Output: s_j for every binary j

1: $s_j \leftarrow 0$ for all binary j
2: **for** each nonzero (i, j, a_{ij}) **in parallel do**
3: $\quad \delta \leftarrow (1 - 2\bar{x}_j)a_{ij}$ {flip changes x_j by ± 1}
4: $\quad r' \leftarrow r_i + \delta$ {residual after flip}
5: $\quad$ compute p_{ij} from $(r_i, r'), w_i$
6: $\quad s_j \mathrel{+}= p_{ij}$ {`atomicAdd`}
7: **end for**
8: **return** s

(a) row-sorted COO $\Rightarrow$ coalesced a_{ij}, r_i, w_i reads

(b) random $\bar{x}_j$ reads $\Rightarrow$ L2-resident bitset

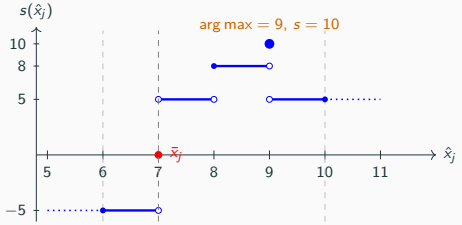
(c) precomputed residuals $\Rightarrow$ one load per nonzero

(d) normalize rows to $\leq \Rightarrow$ fewer cases and less divergence

Best shift for continuous variables: a worked example

Variable x_j with incumbent $\bar{x}_j = 7$, bounds $[l_j, u_j] = [6, 10]$. Five constraints, each shown *after substituting the other variables at their incumbent values*:

	Constraint	w	At $\bar{x}_j = 7$
C_1	$a_1^\top x \le b_1 \rightsquigarrow x_j \ge 4$	2	satisfied
C_2	$a_2^\top x \le b_2 \rightsquigarrow x_j \ge 8$	6	violated
C_3	$a_3^\top x \le b_3 \rightsquigarrow x_j \ge 9$	4	violated
C_4	$a_4^\top x \le b_4 \rightsquigarrow x_j \le 9$	5	satisfied
C_5	$a_5^\top x \le b_5 \rightsquigarrow x_j \le 11$	3	satisfied



Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = 0 \quad \alpha = 0$

position emitted by Δs	
prefix β+prefix $+\alpha$ if pos$>\bar{x}_j$ σ	

Initialize; sweep $\hat{x}_j$ from $-\infty$ (all $\geq$ violated).

Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = -2 \quad \alpha = 0$

position	4
emitted by	C_1
Δs	$+2$
prefix	
β+prefix	
$+\alpha$ if pos$>\bar{x}_j$	
σ	

Emit C_1 ($x_j \geq 4$): sat at $\bar{x}_j$, so $\beta \mathrel{-}= w$. Breakpoint at t=4, $\Delta s = +2$.

Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = -5 \quad \alpha = +6$

position	4	8
emitted by	C_1	C_2
Δs	$+2$	$+3$
prefix		
β+prefix		
$+\alpha$ if pos$>\bar{x}_j$		
σ		

Emit C_2 ($x_j \geq 8$): viol at $\bar{x}_j$, so $\beta \mathrel{-}= \frac{1}{2}w$ and $\alpha \mathrel{+}= w$=6. Breakpoint at t=8, Δs= + 3.

Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = -7 \quad \alpha = +10$

position	4	8	9	9	9+
emitted by	C_1	C_2	C_3	C_4	C_4
Δs	$+2$	$+3$	$+2$	0	-5
prefix					
β+prefix					
$+\alpha$ if pos$>\bar{x}_j$					
σ					

Emit C_4 ($x_j \leq 9$, sat): pushes **two** entries $(9, 0)$ and $(9^+, -5)$ — the $(9, 0)$ is redundant.

Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = -7 \quad \alpha = +10$

position	4	8	9	9	9+	11	11+
emitted by	C_1	C_2	C_3	C_4	C_4	C_5	C_5
Δs	$+2$	$+3$	$+2$	0	-5	0	-3
prefix							
β+prefix							
$+\alpha$ if pos$>\bar{x}_j$							
σ							

Emit C_5 ($x_j \leq 11$, sat): pushes $(11, 0), (11^+, -3)$; no β/α change.

Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = -7 \quad \alpha = +10$

position	4	8	9	9	9+	11	11+
emitted by	C_1	C_2	C_3	C_4	C_4	C_5	C_5
Δs	$+2$	$+3$	$+2$	0	-5	0	-3
prefix	$+2$	$+5$	$+7$	$+7$	$+2$	$+2$	-1
β+prefix							
$+\alpha$ if pos$>\bar{x}_j$							
σ							

Sort by position, **inclusive scan** the $\Delta s \Rightarrow$ prefix.

Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = -7 \quad \alpha = +10$

position	4	8	9	9	9+	11	11+
emitted by	C_1	C_2	C_3	C_4	C_4	C_5	C_5
Δs	$+2$	$+3$	$+2$	0	-5	0	-3
prefix	$+2$	$+5$	$+7$	$+7$	$+2$	$+2$	-1
β+prefix	-5	-2	0	0	-5	-5	-8
$+\alpha$ if pos$>\bar{x}_j$							
σ							

Add β: shift every prefix by the score at $-\infty$.

Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = -7 \quad \alpha = +10$

position	4	8	9	9	9+	11	11+
emitted by	C_1	C_2	C_3	C_4	C_4	C_5	C_5
Δs	$+2$	$+3$	$+2$	0	-5	0	-3
prefix	$+2$	$+5$	$+7$	$+7$	$+2$	$+2$	-1
β+prefix	-5	-2	0	0	-5	-5	-8
$+\alpha$ if pos$>\bar{x}_j$	no	yes	yes	yes	yes	yes	yes
σ							

Positions $>\bar{x}_j$ also pick up the crossing offset α.

Best shift via sort-scan-argmax

The score $s(\hat{x}_j)$ moves only at **breakpoints**. Each constraint emits 1–2 deltas; sweep $\hat{x}_j$ from $-\infty$ to $+\infty$, accumulating β (score at $-\infty$) and α ($\bar{x}_j$-crossing offset). Same example, $\bar{x}_j = 7$, bounds $[6, 10]$:

	Constraint	w	at $\bar{x}_j$
C_1	$x_j \geq 4$	2	sat
C_2	$x_j \geq 8$	6	**viol**
C_3	$x_j \geq 9$	4	**viol**
C_4	$x_j \leq 9$	5	sat
C_5	$x_j \leq 11$	3	sat

$\beta = -7 \quad \alpha = +10$

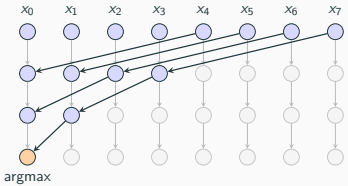
position	4	8	9	9	9+	11	11+
emitted by	C_1	C_2	C_3	C_4	C_4	C_5	C_5
Δs	$+2$	$+3$	$+2$	0	-5	0	-3
prefix	$+2$	$+5$	$+7$	$+7$	$+2$	$+2$	-1
β+prefix	-5	-2	0	0	-5	-5	-8
$+\alpha$ if pos$>\bar{x}_j$	no	yes	yes	yes	yes	yes	yes
σ	-5	$+8$	**$+10$**	**$+10$**	$+5$	$+5$	$+2$

$\sigma = \beta + \text{prefix} + \alpha\,[\text{pos} > \bar{x}_j]$; **argmax** at pos 9, $\sigma = +10$.

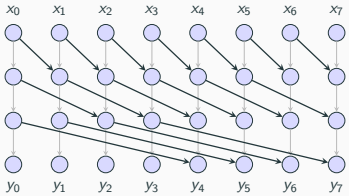
Parallel primitives: reduction, scan, sort

We now have an algorithm that uses our three primitives!

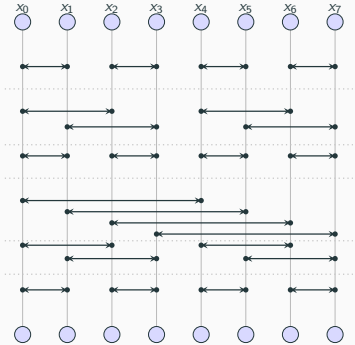
Reduction (argmax)



Scan (prefix sum, $y_j = \sum_{i=0}^{j} x_i$)



Sort



Can we optimize further?

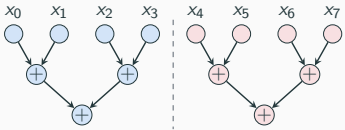
GPU libraries are designed for long vectors, but most MIP columns contain only a few nonzeros.

Can we process several short columns together?

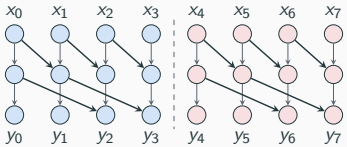
Short vectors: segmented primitives

Pack two columns of length 4 into one warp and run independent networks within each segment.

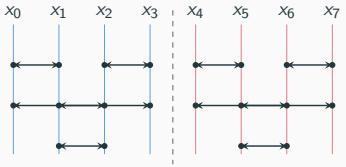
Segmented reduction



Segmented scan



Segmented sort



column 1 column 2

Hand rolling our own GPU kernel

Our best shift algorithm relies on **sort**, **scan**, and **reduction (argmax)** primitives. In practice, most MIP columns are **short vectors** (few nonzeros); GPU libraries, by contrast, are designed for **long vectors**.

Idea. Pack the operations of *multiple columns* into the same warp or thread block. Dispatch by vector length:

- **Length 2–32**: pack several vectors into one warp; sort via **bitonic networks**, scan/reduce via warp shuffles.
- **Length up to 64/128/256**: pack into 256-thread blocks; one thread per element, communicate via shared memory.
- **Length up to 1024**: one block of the nearest enclosing power of two.
- **Length** > 1024: grid-wide primitives across multiple blocks.

Pushing it further: CUDA Graphs

Launching a GPU kernel from the CPU has overhead. In an iterative algorithm, repeated launches and synchronization can become costly even when all data remains on the GPU.

Launch every iteration

```
copy_to_gpu(state);
for (int i = 0; i < n; ++i) {
  tabu_iteration(state);
  if (found_solution(state))
    break;  // synchronize every iteration
}
copy_to_cpu(state);
```

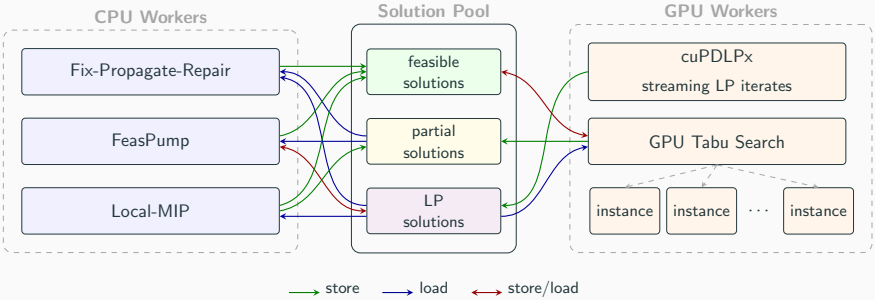
Record once, replay many times

```
copy_to_gpu(state);
start_graph_recording();
for (int i = 0; i < 100; ++i)
  tabu_iteration(state, success_flag);
graph = stop_graph_recording();

for (int batch = 0; batch < n / 100; ++batch) {
  replay(graph);
  if (check(success_flag)) break;
}
copy_to_cpu(state);
```

CHAP records multiple tabu-search iterations in a CUDA Graph, reducing kernel-submission overhead and synchronization frequency while allowing kernels to execute in parallel.

The Architecture of CHAP



- CPU workers (left) and GPU workers (right) communicate through a **shared solution pool** (center)
- Pool stores: feasible solutions, partial (infeasible) solutions, LP solutions
- Each component reads what it needs (LP iterate, incumbent, partial sol.) and writes back what it produces

References I

M. Andersch, G. Palmer, R. Krashinsky, N. Stam, V. Mehta, G. Brito, and S. Ramaswamy. NVIDIA hopper architecture in-depth. NVIDIA Technical Blog, 2022. URL https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/. Accessed 2026-03-25.

A. Çördük, P. Sielski, A. Boucher, and K. Aatish. GPU-accelerated primal heuristics for mixed-integer programming, 2025. URL https://arxiv.org/abs/2510.20499. NVIDIA cuOpt.

Gurobi Optimization, LLC. Does gurobi support gpus? Gurobi Help Center, 2025. URL https://support.gurobi.com/hc/en-us/articles/360012237852-Does-Gurobi-support-GPUs. Accessed 2026-03-25.

References II

N.-C. Kempke and T. Koch. Low-precision first-order method-based fix-and-propagate heuristics for large-scale mixed-integer linear optimization, 2025. URL https://arxiv.org/abs/2503.10344.

P. Lin, S. Cai, M. Zou, and J. Lin. Local-mip: Efficient local search for mixed integer programming. *Artificial Intelligence*, 348:104405, 2025. ISSN 0004-3702. doi: 10.1016/j.artint.2025.104405.

H. Lu, Z. Peng, and J. Yang. cupdlpx: A further enhanced gpu-based first-order solver for linear programming, 2025. URL https://arxiv.org/abs/2507.14051.

B. Luteberget and G. Sartor. Feasibility jump: an lp-free lagrangian mip heuristic. *Mathematical Programming Computation*, 15(2):365–388, Mar. 2023. ISSN 1867-2957. doi: 10.1007/s12532-023-00234-8.

References III

G. Mexi, M. Besançon, S. Bolusani, A. Chmiela, A. Hoen, and A. Gleixner. Scylla: A matrix-free fix-propagate-and-project heuristic for mixed-integer optimization. In *International Conference on Operations Research*, pages 65–72. Springer, 2023. doi: 10.1007/978-3-031-58405-3_9.

D. Salvagnin, R. Roberti, and M. Fischetti. A fix-propagate-repair heuristic for mixed integer programming. *Mathematical Programming Computation*, Oct. 2024. ISSN 1867-2957. doi: 10.1007/s12532-024-00269-5.

Backup: computational results

Setup. 50 presolved instances, 5-min time limit.
Hardware: Intel Xeon 8481C + NVIDIA H100. Gap% and Primal Integral (PI) w.r.t. best known solution[2].

Configuration	Found	Wins	Gap%	PI
Pure CPU	47	3	19.57	72.59
CPU + GPU LP (cuPDLPx)	46	13	12.17	51.01
CPU + GPU LP + GPU tabu	**47**	**22**	**8.84**	**41.84**

Solver	Found	Wins	Only
CHAP	**47**	11	1
Gurobi (5 min)	44	25	1
cuOpt (heuristics only)	43	6	0

[2] best known solution by Gurobi/Xpress in 8 h.