Accelerating Domain Propagation:
an Efficient GPU-Parallel Algorithm over Sparse Matrices

Boro Sofranac   sofranac@zib.de

Berlin Institute of Technology and Zuse Institute Berlin

Ambros Gleixner   gleixner@zib.de

HTW Berlin and Zuse Institute Berlin

Sebastian Pokutta   pokutta@zib.de

Berlin Institute of Technology and Zuse Institute Berlin

Abstract

Fast domain propagation of linear constraints has become a crucial component of today's best algorithms and solvers for mixed integer programming and pseudo-boolean optimization to achieve peak solving performance. Irregularities in the form of dynamic algorithmic behaviour, dependency structures, and sparsity patterns in the input data make efficient implementations of domain propagation on GPUs and, more generally, on parallel architectures challenging. This is one of the main reasons why domain propagation in state-of-the-art solvers is single thread only. In this paper, we present a new algorithm for domain propagation which (a) avoids these problems and allows for an efficient implementation on GPUs, and is (b) capable of running propagation rounds entirely on the GPU, without any need for synchronization or communication with the CPU. We present extensive computational results which demonstrate the effectiveness of our approach and show that ample speedups are possible on practically relevant problems: on state-of-the-art GPUs, our geometric mean speed-up for reasonably-large instances is around 10x to 20x and can be as high as 195x on favorably-large instances.

1 Introduction

Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and a subset $I \subseteq N = \{1, \ldots, n\}$, a *mixed integer linear program* (MIP) is an optimization problem that can be written as

$$\min\{c^T x \mid Ax \leq b, x \in \mathbb{R}^n, x_j \in \mathbb{Z} \text{ for all } j \in I\}. \tag{1}$$

Typically, MIPs are $\mathcal{NP}$-hard to solve, but surprisingly fast algorithms exists in practice [1, 2]. The most successful methods to solve MIPs are the *branch-and-bound* algorithm and its extensions. The main idea of this algorithm is to split the original problem into subproblems which are easier to solve (*branching*). By repeatedly branching on the subproblems, a search tree is obtained. At each subproblem, the *bounding* step uses relaxations to compute lower bounds and prune suboptimal nodes of the tree in order to avoid enumerating exponentially many subproblems. Relaxations are typically obtained by dropping integrality requirements on variables x_j for $j \in I$, at which point a *linear program* (LP) is obtained and solved by, e.g., the simplex algorithm [3].

One of the supplementary techniques used to improve the initial problem formulation and decrease the size of the branching tree is to limit the domains of the variables x_j to the value assignments which can be completed to *feasible solutions* of a given (sub)problem. The process of computing these ranges is called *domain propagation* [4,5].

Whereas many areas of applied mathematics (e.g., Deep Learning) have strongly profited from the increased computing power of parallel coprocessors like *Graphics Processing Units* (GPUs), success in using these resources to solve MIPs has been very limited [6]. The main challenges for developing GPU-accelerated MIP techniques have been (a) highly sparse input data with irregular sparsity patterns; (b) inherent sequential properties of many of the best methods used in the field; and (c) their non-uniform algorithmic behavior. On top of this, many of the existing algorithms were designed with a sequential execution model in mind, which does not allow to transform them easily to an efficient GPU implementation.

In this paper, we focus on domain propagation and propose an efficient algorithm for GPUs. In addition to being able to obtain results faster with this algorithm under almost all circumstances, we hope that this work can pave

the way for the implementation of other methods or motivate the development of new optimization methods on GPUs. Our algorithm is capable of running propagation rounds entirely on the GPU, without the involvement of the CPU. Consequently, this also presents an opportunity to investigate the use of domain propagation in conjunction with existing or future methods that also run completely on the GPU. This is especially interesting given the fact that state-of-the-art MIP solvers largely do not exploit GPUs during the solving process, leaving them idle.

We begin by formally introducing the necessary background and notation.

1.1 Domain Propagation

Domain propagation is an iterative technique used to tighten the bounds of variables at each node of the branch-and-bound tree. It is also used as a pre-processing technique to improve the formulation of a given MIP model [7]. In the following, we consider linear constraints of the form

$$\underline{\beta} \leq a^T x \leq \overline{\beta}, \tag{2}$$

where $\underline{\beta}, \overline{\beta} \in \mathbb{R} \cup \{-\infty, \infty\}$ are left and right hand sides, respectively, and $a \in \mathbb{R}^n$ is the vector of constraint coefficients. We assume that variables x_j have lower and upper bounds $\ell_j, u_j \in \mathbb{R} \cup \{-\infty, \infty\}$. Before stating the formulas for updating variable bounds, we need the following definition.

Definition 1 (activity bounds) *Given a constraint of form* (2) *and bounds* $\ell \leq x \leq u$, *we call*

$$\underline{\alpha} := \sum_{i=0}^{n} a_i b_i \ \text{with} \ b_i = \begin{cases} \ell_i & \text{if } a_i > 0, \\ u_i & \text{if } a_i \leq 0, \end{cases} \tag{3a}$$

the minimum activity, *and*

$$\overline{\alpha} := \sum_{i=0}^{n} a_i b_i \ \text{with} \ b_i = \begin{cases} u_i & \text{if } a_i > 0, \\ \ell_i & \text{if } a_i \leq 0, \end{cases} \tag{3b}$$

the maximum activity *of the constraint.*

Then domain propagation is based on three observations [5, Sec. 7.1] and translate into the following algorithmic steps.

1: If $\underline{\beta} \leq \underline{\alpha}$ and $\overline{\alpha} \leq \overline{\beta}$, then the constraint is redundant and can be removed.

2: If $\underline{\alpha} > \overline{\beta}$ or $\underline{\beta} > \overline{\alpha}$, then the constraint cannot be satisfied and hence the entire (sub)problem is infeasible.

3: Let x satisfy (2), then for all $j = 1, \ldots, n$ with $a_j > 0$,

$$\frac{\underline{\beta} - \overline{\alpha}}{a_j} + u_j \leq x_j \leq \frac{\overline{\beta} - \underline{\alpha}}{a_j} + \ell_j, \tag{4a}$$

and for all $j = 1, \ldots, n$ with $a_j < 0$,

$$\frac{\overline{\beta} - \underline{\alpha}}{a_j} + u_j \leq x_j \leq \frac{\underline{\beta} - \overline{\alpha}}{a_j} + \ell_j. \tag{4b}$$

If x_j is integral, lower bounds can be rounded up and upper bounds can be rounded down to strengthen them further.

If Steps 1 and 2 are not applicable, the propagation algorithm computes new bounds $\ell_j^{\text{new}}, u_j^{\text{new}}$ in Step 3. If $\ell_j^{\text{new}} > \ell_j$ or $u_j^{\text{new}} < u_j$ for some j, then the bounds of variable x_j are updated. To apply domain propagation to a MIP formulation, one simply applies these steps to all of its constraints.

An actual implementation may skip Steps 1 and 2 without changing the result. This is because for redundant constraints Step 3 correctly detects no bound tightenings, and for infeasible constraints, Step 3 leads to at least one variable with an empty domain, i.e., $\ell_j^{\text{new}} > u_j^{\text{new}}$.

Let us consider what happens when the steps are applied to all constraints of the system and some bound changes have been found. First, note that the formulas for computing new variable bounds in (4a) and (4b) depend on the activity bounds $\underline{\alpha}$ and $\overline{\alpha}$. The activity bounds themselves depend on ℓ_j and u_j. Second, note that several constraints typically share the same variable x_j. In this case, updating the bounds of x_j during the processing of one constraint changes the bound and activity values for all other constraints that contain x_j. Propagating the affected constraints again can then lead to further tightenings.

Hence, domain propagation is usually applied iteratively. Each iteration of applying Steps 1 to 3 to all or a subset of the constraints is also called a *propagation round*. If no bound updates are found during a given round, then no further improvements are possible, and the algorithm terminates.

As pointed out in [4], iterated domain propagation can be interpreted as a fixed-point iteration in the space of variable and activity bounds, and there exists a unique limit point of this fixed-point iteration. Iterated domain propagation converges to this well-defined result, however, not necessarily in finite time [4]. Also in practice convergence can be very slow [5]. The most common way to deal with non-finite or slow convergence in practice is to terminate when the improvements made fall below a specified threshold. Such tolerance-based termination criteria yield finite termination, but may fail to compute the tightest bounds possible.

1.2 Graphics Processing Units

In this section we briefly introduce GPU concepts used throughout the paper. Although the key concepts are independent, we restrict our notation to NVIDIA's GPUs and the CUDA programming model. GPUs are massively parallel processors capable of running hundreds of thousands of threads. Threads are grouped into *warps* in hardware (usually 32 threads each) and forced to execute in a SIMD (Single Instruction Multiple Data) fashion and share the same resources. On the programming level, threads are divided into *thread blocks*. Different thread blocks are scheduled for execution independently, but they all have access to the same *global memory*. Threads inside a single thread block can be synchronized and a given amount of exclusive *shared memory* is available to each block. Accesses to shared memory are usually much faster than accesses to global memory.

1.3 CSR Sparse Matrix Storage Scheme

To store a sparse matrix A, we adopt a ubiquitously used *Compressed Sparse Row (CSR)* storage scheme. This scheme consists of three arrays: *values*, *cols* and *row_delimiters*. Non-zeros of A are stored row-wise in the *values* array. The column index for each non-zero is stored in *cols* array. Finally, the *row_delimiters* array stores the indices of beginning elements for each row.

1.4 Related Work

While our motivation to study domain propagation are MIPs and we restrict ourselves to linear constraints, the concept is more general and has been rediscovered several times since the 1970s in different communities. First, it was used in the AI (Artificial Intelligence) and CP (Constraint Programming) communities, where its origins can be traced back to the Waltz algorithm [8]. Several other techniques related to domain propagation are widely used in AI and CP and known under the names *domain filtering*, *domain reduction*, *bound reduction*, *range reduction* and *constraint propagation*. In MIP, domain propagation was first discussed in the context of presolving [7,9]. In global optimization and mixed integer non-linear programming, it is most commonly known as *feasibility-based bounds tightening* (FBBT) and was first used in [10].

The literature on previous efforts to successfully exploit GPUs in exact MIP methods is scarce. We refer to the recent survey by Boyer et al. [6] and references therein and we only provide a very brief summary here. Most work in this direction has focused on the simplex method (especially its linear algebra), on dynamic programming approaches for solving the knapsack problem, and on branch-and-bound implementations for a few special problem classes. Somewhat more attention was paid to GPU implementations of metaheuristic

methods which aid exact methods, as their compute tasks seem more amenable to an efficient implementation on GPUs.

1.5   Contribution

To the best of our knowledge, this paper presents the first algorithm for domain propagation on GPUs. We discuss and address two main sources of irregularity that challenge efficient GPU implementations: 1. the dynamic behaviour of the existing CPU-based algorithms, and 2. the highly sparse and irregular structure of the constraint matrix A.

We propose a new algorithm which better matches the *throughput-based* model of GPUs. To deal with the highly irregular structure of the input matrix A, we found inspiration in existing work on parallelizing sparse linear algebra and demonstrate that some of these ideas can be carried over to domain propagation. As a result, we obtain an efficient method that runs entirely on the GPU and yields significant speedups for domain propagation on the MIPLIB 2017 test bed, the *de-facto* standard for MIP solvers.

The rest of the paper is organized as follows. In Section 2 we describe the new parallel algorithm and its features. In Section 3 we discuss techniques for an efficient implementation of the algorithm, and finally, in Section 4 we present computational results.

2   Parallelizing Domain Propagation

We strive to exploit two sources of parallelism from the domain propagation algorithm: First, we exploit the fact that we can apply the propagation reductions to each constraint, i.e., to each row of A, independently and thus do so in parallel. The repercussions of this on the algorithm are discussed in Section 2.2. Second, we exploit parallelism inside the processing of each constraint. This is discussed in Section 3.

2.1   Features of Sequential Domain Propagation

We first present the main features of the sequential domain propagation algorithm and discuss why this algorithm is not suited for efficient implementation on GPUs. CPU-based, sequential implementations of the domain propagation algorithm follow the *latency-based* sequential programming model. The main steps of this algorithm are summarized in Algorithm 1.

Informally speaking, this algorithm starts with the first non-zero entry of A, computes everything it can for the corresponding variable, tightens its bounds if possible, and then moves on to the next non-zero. It can stop processing a constraint or a variable early if the sufficient conditions in Line 9 and Line 11 are met. This avoids unnecessary work as these checks can be performed before the new bound candidates are computed and compared to the old bounds. In addition, this algorithm takes advantage of the fact that a constraint can trigger propagation of other constraints only if they share at least one variable by implementing a marking mechanism (Lines 1, 6, 7 and 20).

This irregular behaviour poses challenges for an efficient implementation on a GPU. Threads assigned to units of work that are stopped early would remain idle during the rest of the execution in a given propagation round, leaving hardware underutilized. The remaining threads would potentially be accessing memory far apart, resulting in uncoalesced accesses. Notice that we do not know a priori which parts would terminate early and at which level, or which constraints will be marked in the next round. This induces highly dynamic behaviour both inside a given propagation round and between different rounds as well.

Additionally, notice that the amount of work a given thread would perform heavily depends on where, if at all, it will terminate early. Computing $\underline{\alpha}$ and $\overline{\alpha}$ in Line 8, e.g., also requires a loop over the variables in the constraint, see (3a) and (3b). The marking operation in Line 20 involves iterating over the columns of A and finding all constraints that contain the variable in question. The dynamic behaviour of the algorithm described in the previous paragraph would also make load balancing the work between different threads difficult.

Algorithm 1 Sequential domain propagation

Input: System of linear constraints $\underline{\beta} \leq Ax \leq \overline{\beta}$, $\ell \leq x \leq u$
Output: Tightened variable bounds $\ell^{\text{new}} \leq x \leq u^{\text{new}}$
1: mark all constraints c in $\underline{\beta} \leq Ax \leq \overline{\beta}$
2: bound_change_found $\leftarrow$ true
3: **while** bound_change_found **do**
4:     bound_change_found $\leftarrow$ false
5:     **for each** constraint c in $\underline{\beta} \leq Ax \leq \overline{\beta}$ **do**
6:         **if** c marked **then**
7:             unmark c
8:             compute $\underline{\alpha}$, $\overline{\alpha}$ via (3a) and (3b)
9:         **if** can c propagate **then**
10:            **for each** variable v in c **do**
11:               **if** can v be tightened **then**
12:                 compute ℓ_v^{new}, u_v^{new} via (4a) and (4b)
13:                 **if** $\ell_v^{\text{new}} > \ell_v$ **then**
14:                     $\ell_v \leftarrow \ell_v^{\text{new}}$
15:                     bound_change_found $\leftarrow$ true
16:                 **if** $u_v^{\text{new}} < u_v$ **then**
17:                     $u_v \leftarrow u_v^{\text{new}}$
18:                     bound_change_found $\leftarrow$ true
19:               **if** bound_change_found **then**
20:                 mark all constraints c with v in c
21: **return** ℓ^{new}, u^{new}

2.2 The Price of Parallelism

As mentioned earlier, the constraints of a given system can be propagated independently. Exploiting this parallelism, however, comes at a cost; it will result in a less efficient algorithm in the worst-case scenario. The main reason for this is that a bound change found during the sequential execution of the algorithm becomes immediately available to the propagation of the subsequent constraints. If this bound change triggers propagation of one of the subsequent constraints, this propagation can be done during the same round. In the parallel case however, all constraints are propagated independently, hence propagation triggered in this way would have to wait until the next propagation round.

The worst case of such a propagation pattern is a *cascading propagation pattern*, where Constraint 1 triggers propagation of Constraint 2, then Constraint 2 triggers Constraint 3, and so forth. A sequential implementation could propagate this pattern in one round, while a parallel implementation that propagates each constraint independently would require m rounds, where m is the number of constraints in the system. In the best-case scenario, the parallel and sequential algorithm propagate the system in the same number of rounds.

To roughly gauge this "price of parallelism", we conducted a preliminary experiment over 903 instances from the MIPLIB 2017 test set [11] on which both our parallel and sequential implementations converge to identical results. In this experiment, the average number of propagation rounds of the sequential implementation was 3.9, but increased to 5.0 rounds on average for the parallel implementation, hence an average increase by a factor 1.3. However, the maximum increase observed for an instance was as large as 33.7.

2.3 A GPU-Targeted Parallelization

In this section, we describe the proposed GPU-parallel domain propagation algorithm and discuss why it better matches the GPU model. Finally, we highlight how the missing features of this algorithm, compared to the sequential algorithm, are mostly remedied by the massively parallel GPU model.

The parallel algorithm we propose for implementation on the GPUs is shown in Algorithm 2. It does not contain the early termination checks of the Algorithm 1. This means that at the expense of performing more

computations we avoid the irregular behaviour of the CPU version induced by these checks. Taking constraint marking as an example, note that unmarked constraints would not be processed by the CPU algorithm, while the GPU algorithm would process such constraint even though it cannot yield improved bounds. What we gain though is an algorithm with a static sequence of computations that can exploit the massively parallel GPU model.

Algorithm 2 GPU-parallel domain propagation (schematic)

Input: System of linear constraints $\underline{\beta} \leq Ax \leq \overline{\beta}$, $\ell \leq x \leq u$
Output: Tightened variable bounds $\ell^{\text{new}} \leq x \leq u^{\text{new}}$

1: bound_change_found $\leftarrow$ true
2: **while** bound_change_found **do**
3:     **for** constraint c in A **parallel do**
4:         compute $\underline{\alpha}$, $\overline{\alpha}$ via (3a) and (3b)
5:     **for** constraint c in A **parallel do**
6:         **for** variable v in c **parallel do**
7:             compute ℓ_v^{new}, u_v^{new} via (4a) and (4b)
8:             **if** $\ell_v^{\text{new}} > \ell_v$ **then**
9:                 $\ell_v \leftarrow \ell_v^{\text{new}}$
10:                bound_change_found $\leftarrow$ true
11:            **if** $u_v^{new} < u_v$ **then**
12:                $u_v \leftarrow u_v^{\text{new}}$
13:                bound_change_found $\leftarrow$ true
14: **return** ℓ^{new}, u^{new}

Notice the shift in the way the two algorithms progress through the computation. The CPU-based Algorithm 1 starts with the first constraint and its first variable and continues processing that variable until no more processing is possible. It then moves to the next variable and the process is repeated. Hence, the activities $\underline{\alpha}$ and $\overline{\alpha}$ are computed for the first constraint, then all the variables in that constraint are processed, then the process is repeated on the remaining constraints. The GPU-based Algorithm 2 instead starts with computing $\underline{\alpha}$ and $\overline{\alpha}$ for *all* constraints. Then it computes new bound candidates for *all* non-zeros of A. A useful analogy to graph algorithms here is that the CPU Algorithm 1 resembles a "depth-first search" and the GPU algorithm resembles a "breadth-first search" of a graph. In terms of the GPU model, the new algorithm allows for better load balancing, maximizing the *throughput*.

This parallel algorithm also allows us to access large parts of necessary memory in a coalesced way. However, to understand how the memory is accessed, we also need to take in account the sparsity pattern and the storage scheme of A. Thus, we discuss memory accesses in Section 3 which deals with the irregular structure of A.

For an example of why our proposed algorithm leads to better load balancing in the parallel case, notice that all variables in a given constraint share the same $\underline{\alpha}$ and $\overline{\alpha}$. If we first use all available threads to precompute the activities, then synchronize and perform bound updates, no threads are left idle and no computation is repeated. Otherwise, either all the threads compute the same activity values (best-case scenario we get the result in the time a single thread takes to compute the activity) or some of them are idle and waiting for others to compute the activities before performing the bound updates. Additionally, having a group of threads cooperate on computation of activities will lead to a faster algorithm. This is because the computation of activities allows for some parallelism to be exploited; see Section 3.1.

We now highlight how the *throughput-based* GPU programming model partly remedies the lack of early termination checks of Algorithm 1. Let us only consider the constraint marking feature; the conclusion drawn will be analogous for other early termination checks.

Assume we have m constraints in the system, k of which are marked for propagation, where $k \leq m$. Furthermore, assume that the costs of propagating one constraint on the hardware where the sequential and parallel algorithms are run are C_{seq} and C_{par}, respectively, and that the parallel algorithm has p processing units for parallel computation at its disposal. The sequential algorithm would only propagate the k marked constraints and thus finish the propagation with the cost of $k \cdot C_{\text{seq}}$. The parallel algorithm, on the other hand, would propagate the system with the cost of $\lceil \frac{m}{p} \rceil \cdot C_{\text{par}}$. For the case of $m \leq p$, it would not matter for the parallel algorithm

whether it propagates all the m constraints or only the k marked constraints. For $m > p$, we pay the price for propagating one constraint, i.e., C_{par} for each additional p constraints that are propagated.

In practice, it is hard to exactly quantify p for GPUs due to the complexity of their hardware and execution model. However, we can get a rough idea of what orders of magnitude are involved in the computation. Our MIPLIB 2017 test bed used for comparison from Section 4.3 shows on average 223,094 constraints, 154,681 variables, and 1,766,173 non-zeros, respectively. On the other hand, modern GPUs offer order of tens of thousands of threads running in parallel. Thus, given the size of practically relevant problems and the amount of parallelism modern GPUs offer, we can expect that the price for extra amounts of work in the parallel algorithm will be mostly negligible.

Further properties of GPUs help in reducing the cost we pay for extra computations. For example, the GPU hardware is optimized to combine multiple memory accesses into a single transaction (*memory coalescing*). As will be discussed in Section 3, the GPU algorithm allows us to access large parts of necessary memory in a fully coalesced manner. So the memory to be operated on by the extra threads will often be already loaded. In low arithmetic intensity algorithms like domain propagation memory accesses are typically dominant parts of GPU implementations in terms of runtime. Hence, we do not expect these extra arithmetic operations to have a significant effect on the total runtime of the algorithm; our computational results confirm this.

3 Handling Irregular Structure

In practice, constraint matrices A coming from MIPs are very sparse. Additionally, MIPs often contain so-called *connecting constraints* which are very dense. Consequently, even though matrix A might be very sparse overall, it may contain a few very dense rows. This poses a challenge for load balancing on GPUs. Consider two naive approaches of assigning a single thread per constraint and assigning a warp or block of threads per constraint. In the former case, threads assigned to dense rows would have much more work to do, leaving other threads idle. In the latter case, warps or blocks assigned to rows with a small number of non-zeros would remain underutilized.

We already discussed how GPU hardware is optimized around coalescing memory accesses of threads in a warp into one or as few as possible memory accesses. The approach of assigning one thread per constraint would perform poorly in this regard as neighboring threads running in parallel would access memory locations which are not adjacent in the row-major CSR format. Assigning a warp of threads per constraint, on the other hand, results in fully coalesced memory accesses: each thread of a warp takes one non-zero element, which are stored next to each other in the CSR format.

In this section, we address above-mentioned challenges and present an algorithm which combines good load balancing with coalesced memory accesses. To introduce the main idea, we first focus on the computation of the minimum and maximum activities $\underline{\alpha}$ and $\overline{\alpha}$ in Section 3.1. The second part of the algorithm, which computes new bound candidates is discussed in Section 3.4.

3.1 Computing Minimum and Maximum Activities

The first step of the parallel algorithm computes minimum and maximum activities $\underline{\alpha}$ and $\overline{\alpha}$ for all constraints of the system via (3a) and (3b). Our main approach for implementing these formulas is the observation that their efficient implementation for a matrix A on GPUs is conditioned by the same factors that condition implementations of a sparse matrix-vector product (SpMV) of A and some right-hand side vector. SpMVs and their efficient implementations on GPUs have been extensively studied over the years as they play a crucial role in computational science. We will then be able to carry over ideas to overcome challenges described at the beginning of this section for computing activities.

To support our observation, we rely on the fact that SpMV is bandwidth bound (i.e., its arithmetic intensity is very low), so good memory access patterns greatly improve performance [12]. Notice that arithmetic intensity of SpMV and computing $\underline{\alpha}$ or $\overline{\alpha}$ is the same. When it comes to memory needed to perform the operations, notice that each l_i and u_i is accessed exactly once: either for $\underline{\alpha}$ or for $\overline{\alpha}$. This means that we need to access A, l, and u exactly once to implement (3a) and (3b). This is exactly the same as if we were computing SpMVs $A \cdot l$ and $A \cdot u$ with the same matrix A.

Greathouse and Daga introduced an algorithm in [12] which addresses issues discussed at the beginning of this section in the context of SpMVs. This algorithm is called *CSR-adaptive*. It handles well both structured and unstructured matrices and explicitly addresses the case of having very long rows in an otherwise sparse matrix. We now briefly introduce the main idea of this algorithm before discussing the changes made to facilitate computation of activities. For a detailed discussion of *CSR-adaptive* see [12].

3.2   The CSR-Adaptive Algorithm

The main idea of the algorithm is to divide the matrix A into *row blocks*, and have one CUDA threads block work on one row block. If the number of non-zeros in some rows is small, they will be grouped together in one row block. A so-called *CSR-stream* algorithm will then be applied to this row block. If a given row block consists of only one or few rows, a so-called *CSR-vector* variant will be applied.

The CSR-stream algorithm first assigns one thread to each non-zero in the row block and loads them into shared memory. This results in fully coalesced memory accesses. Afterwards, a number of threads is assigned per each row that is present in the row block. These threads then carry out the necessary computations and reductions. The CSR-vector algorithm assigns one warp of threads to the row block which then perform the computations and reductions.

3.3   Computing Activities with CSR-Adaptive

On the implementation level, we make the following changes to the CSR-adaptive algorithm as presented in [12]. First, instead of working on one right-hand side array, we adapt the algorithm to work on two arrays: ℓ and u. As part of the same change, the algorithm now outputs two arrays, the minimum and maximum activities.

Second, the arithmetic operations are adjusted to (3a) and (3b). Note that this leaves the reductions (i.e., summations) unchanged as it only differs from SpMV in computing the local summands $a_i b_i$.

Third, we allow the CSR-vector algorithm to use all warps in a CUDA thread block if the rows are extremely long. In this case, each warp computes partial sums of its elements, after which the partial sums are reduced in shared memory. In our implementation, we use a (somewhat arbitrary) threshold value of 64 to switch between the two variants of CSR-vector.

Both the CSR-stream and the CSR-vector variant store the computed activities in local shared memory, as they will be utilized in forthcoming computations (see Section 3.4).

3.4   Computing New Bound Candidates

The new bound candidates for each variable of a constraint $\underline{\beta} \leq a^T x \leq \overline{\beta}$ are computed by (4a) and (4b). This operation *maps* each non-zero element a_{ij} of A to its lower and upper bound candidates for variable j.

Our variant of the *CSR-adaptive* algorithm for computing activities from Section 3.3 works on a granularity level of one thread per non-zero element of A, which is the same granularity we need to map non-zero elements of A to new bound candidates. Moreover, each thread loads the values a_j, l_j and u_j into shared memory, and the computed $\underline{\alpha}$ and $\overline{\alpha}$ are also saved in shared memory. Thus, we extend the activities kernel to also compute (4a) and (4b) after the computation of activities is complete. In most cases, this implementation benefits from reusing the values that are kept in shared memory and avoids expensive GPU global memory loads.

Finally, once individual threads have the corresponding lower and upper bound candidate, they are compared to the best current bounds and updated if necessary. We use atomic operations to make sure these updates do not produce race conditions. The pseudocode in Algorithm 3 summarizes the final algorithm.

The necessity to avoid race conditions comes from the situation when several constraints share the same variable(s), and different threads try to update their bounds at the same time. Threads updating bounds for non-zero elements in the same column of the constraint matrix A will have to compete for hardware resources if they happen to be updating their bounds at the same time. Because different columns can be processed independently, this step of the algorithm will favor matrices with more columns than rows for a fixed number of non-zeros, from the perspective of performance. At the same time, because of the summation of elements within

Algorithm 3 Kernel performing one propagation round in Algorithm 2. *blockIdx* is the index of the current CUDA block the thread executing the code belongs to.

1: start_row ← row_blocks[blockIdx]
2: end_row ← row_blocks[blockIdx + 1]
3: nnz_block ← row_ptrs[end_row] - row_ptrs[start_row]
4: **if** end_row - start_row > 1 **then**
5:     compute $\underline{\alpha}_{\text{row}}, \overline{\alpha}_{\text{row}}$ by *CSR-stream*
6: **else**
7:     **if** nnz_block < length_threshold **then**
8:         compute $\underline{\alpha}_{\text{row}}, \overline{\alpha}_{\text{row}}$ by *CSR-vector* with one warp
9:     **else**
10:         compute $\underline{\alpha}_{\text{row}}, \overline{\alpha}_{\text{row}}$ by *CSR-vector* with all warps
11: __syncthreads()
12: *// each thread knows the row idx i and column idx j of the non-zeros it is processing*
13: compute $\ell_{i,j}^{\text{cand}}, u_{i,j}^{\text{cand}}$ via (4a) and (4b)
14: atomic $u_j \leftarrow \min(u_j, u_{i,j}^{\text{cand}})$
15: atomic $\ell_j \leftarrow \max(\ell_j, \ell_{i,j}^{\text{cand}})$

---

a row of the matrix, the activities computation (see Section 3.1) can process rows independently and will favor matrices with more rows than columns. As a result, the performance of the algorithm will not only depend on the number of non-zeros of the constraint matrix A, but also on its shape.

4   Computational Experiments

In this section, we present our experimental setup for studying the behavior of the proposed algorithms and discuss the results obtained.

4.1   Test Set

To benchmark the algorithms, we use the MIPLIB 2017 collection set with 1065 MIP instances, which is currently the largest and most widely adopted general test bed for MIP algorithms [11]. The open source reader we used was not able to load 46 instances, leaving the set at 1019 instances. The number of non-zeros in the constraint matrix A of this test set ranges from 3 to 256,963,402. Because small instances do not provide enough workload to justify its parallelization on GPUs, we further eliminate all 114 instances that have less than 1000 variables and 1000 constraints, which leaves us with a set of 905 instances.

To better capture the performance of the algorithms with respect to the size of the input problems, we partition this set into eight subsets of increasing size. We denote by $[s, t)$ the set of instances that contain at most t variables and t constraints, but have at least s variables or s constraints. We consider the partition $[1k, 10k)$, $[10k, 20k)$, $[20k, 40k)$, $[40k, 80k)$, $[80k, 160k)$, $[160k, 320k)$, $[320k, 640k)$, and $[640k, \infty)$, and refer to these sets as *Set-1* to *Set-8*.

4.2   Algorithms and Hardware

We compare four algorithms:

cpu_seq is a sequential implementation of Algorithm 1. It is always executed on a single core, following closely the current state-of-the-art implementation [13].

cpu_omp is a shared memory-parallel implementation of Algorithm 1. OpenMP is used to parallelize the loop at Line 5. To improve load balancing, the set of constraint indices is pre-processed and only those marked for propagation are assigned to available threads. OpenMP locks are used to prevent race conditions while updating bounds.

gpu_atomic is the implementation of Algorithm 3 on GPUs. Atomic operations are used to prevent race conditions while updating bounds.

gpu_reduction is a variant of the discussed GPU algorithm where instead of using atomics to avoid race conditions while updating bound candidates, the bound candidates are saved in a column-major format in GPU global memory. Afterwards, the best bound is computed using a reduction. As this version turned out inferior to using atomic operations, we did not provide a detailed description in previous sections, but we include the results nevertheless. The main drawback of this algorithm is that it has to write the vectors of new lower and upper bound candidates in GPU global memory and load them again to perform the reduction. This results in additional costly global memory loads which the atomic kernel avoids. Additionally, this algorithm requires column-major storage of the matrix A to index the saved bound candidates.

The algorithms are tested on the following architectures:

- **V100** NVIDIA Tesla V100 PCIe 32GB GPU,

- **RTX** NVIDIA Titan RTX 24GB GPU,

- **P400** NVIDIA Quadro P400 2GB GPU,

- **amdtr** 64-core AMD Ryzen Threadripper 3990X @ 3.30 GHz with 128 GB RAM CPU,

- **xeon** 24-core Intel Xeon Gold 6246 @ 3.30GHz with 384 GB RAM CPU,

All algorithms are executed in double precision arithmetic. The *cpu_omp* algorithm is run on the *xeon* machine with 24 threads, and on the *amdtr* machine with 64 threads.

4.3   Evaluation Methodology

The main metric we use to compare two executions is *speedup* defined over wall clock time. Average speedups are computed as geometric means. We choose the *cpu_seq* algorithm running on the *xeon* machine as a base reference execution. All speedups are computed against this execution.

Both the sequential and GPU versions have a certain amount of initialization work which only needs to be performed once. In the sequential case, e.g., this is computing the column-major CSC storage of the matrix, which is needed for the marking mechanism (see Algorithm 1). For the GPU algorithm, the blocking of the matrix A is precomputed on the CPU (see Section 3.2), and the necessary memory is sent to the GPU. As in [12], we do *not* include these one-time initialization tasks in our time measurement. Timing starts before the first propagation round and ends after the last propagation round is executed and the results are available (in CPU memory in the CPU case, in GPU memory in the GPU case).

As explained in Section 1.1, domain propagation may not converge to its limit point in finite time. Additionally, some instances may run into numerical difficulties due to floating-point arithmetic. If these difficulties do not happen, the domain propagation algorithm has a unique limit point. When comparing two executions over a test set, only those instances are considered where for all variable bounds the algorithms converge to the same value, within numerical tolerances.

On the *V100* and *RTX* GPUs, the algorithms *gpu_atomic* and *gpu_reduction* converge to the same result as the base *cpu_seq* execution for 789 instances out of the 905 instances in total. On the *P400* GPU, 778 instances reaches the same result. For the shared memory algorithm *cpu_omp*, the executions produce slightly more comparable instances: 839 on *xeon* and 836 on *amdtr*. As a result, also the sizes of the subsets *Set-1* to *Set-8* may differ slightly across executions. The smallest set contains 37, the largest set 288 instances.

4.4   Results

The speedups that can be observed for the six algorithm-machine combinations tested are visualized in Figure 1. Figure 1a) on the left shows the average speedups for the eight subsets *Set-1* to *Set-8* of instances with increasing size. Figure 1b) on the right additionally provides the distributions of the speedups over all instances with
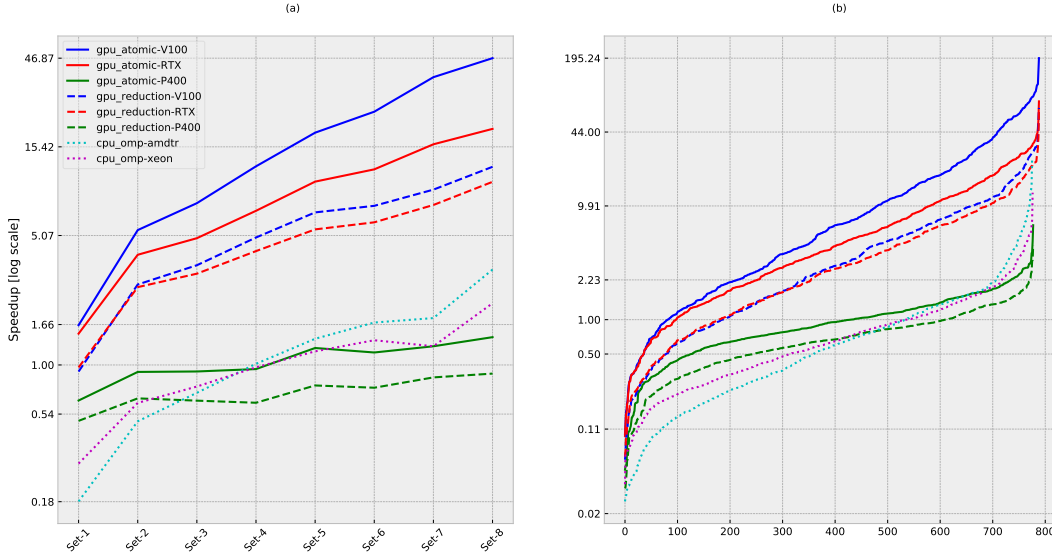
Figure 1: The left graph (a) shows geometric mean of speedups over the eight subsets of instances of increasing size. The right graph (b) shows the distribution of speedups by plotting the individual speedup for each instance, sorted in ascending order. Each curve belongs to one combination of algorithm and machine.

comparable execution. Table 1 provides a summary and reports the average speedup over all instances and over the subsets *Set-1* to *Set-8*, plus the 5th percentile, the median, and the 95th percentile speedup.

As can be seen immediately, both GPU implementations on the high-end hardware *V100* and *RTX* drastically outperform both the sequential algorithm and the CPU-parallel algorithms. Furthermore, *gpu_atomic* outperforms *gpu_reduction* consistently on all three GPUs.

Let us first evaluate the top-performing combination, *gpu_atomic* on *V100*, in more detail. Its average speedup closely follows a linear trend in the size of the instances from *Set-1* to *Set-8*. It is always greater than 1.6, and for *Set-8*, which contains all instances with at least 640,000 variables or constraints, a speedup factor of 46.9 is achieved. This shows that, in general, the speedup of the GPU-parallel algorithms crucially depends on the size of the instances. Over the whole test set, the average speedup is a factor of 6.1, which reflects that a significant portion of the test set are small instances which have limited potential for parallelism. For 5% of the instances, however, *gpu_atomic* on *V100* is at least 62.9 times faster than the sequential implementation.

In contrast, the low-end consumer grade GPU *P400* does not offer enough resources to exploit the parallelism available. On *P400*, the *gpu_reduction* kernel performs worse than *cpu_seq* on all subsets. The *gpu_atomic* kernel incurs a slowdown on more than half of the instances. Nevertheless, given that GPUs are currently a ressource completely unused by MIP solvers, this is an interesting result: On a large fraction of instances, *gpu_atomic* is competitive with an optimized sequential CPU-based implementation of domain propagation even on the consumer grade *P400*. This result suggests that there is potential in exploiting GPUs as coprocessors to supplement a main CPU-based MIP solver even on standard desktop machines.

Also the shared memory-parallel algorithm *cpu_omp* cannot compete with the GPU algorithms on *V100* and *RTX*, which outperform it by a large margin. The comparatively high cost of managing CPU threads proves to not be justified by the low arithmetic intensity of the parallel units of work processed by the threads. This is especially visible for subsets *Set-1* to *Set-4*, which contain all instances with less than 80,000 variables and constraints. Here, *cpu_omp* on both architectures, *amdtr* and *xeon*, is slower than *cpu_seq*. Interestingly, the curves for *amdtr* and *xeon* can be seen to cross in both plots. Initially, the 64-core *amdtr* run loses against the 24-core *xeon* run. Figure 1b) allows two further observations. First, the slopes for the runs on the high-end GPUs *V100* and *RTX* are noticeably steeper, showing that they respond faster to the growth in parallelism. Second, the break-even point from which on *cpu_omp* becomes faster than the sequential implementation lies around the 65th percentile for both *xeon* and *amdtr*. In contrast, *gpu_atomic* on *V100* or *RTX* breaks even latest

11

Table 1: Average speedups across test sets and speedup percentiles

	V100		RTX		P400		amdtr	xeon
	atom.	red.	atom.	red.	atom.	red.	omp	omp
geometric mean speedups								
Set-1	1.64	0.92	1.48	0.96	0.64	0.49	0.18	0.29
Set-2	5.42	2.74	3.98	2.65	0.91	0.65	0.49	0.62
Set-3	7.60	3.49	4.90	3.14	0.92	0.63	0.70	0.76
Set-4	12.0	4.94	6.91	4.16	0.94	0.62	1.01	0.97
Set-5	18.41	6.77	9.96	5.47	1.23	0.77	1.38	1.18
Set-6	23.95	7.35	11.62	5.98	1.16	0.75	1.70	1.36
Set-7	36.93	9.01	15.91	7.43	1.26	0.85	1.80	1.26
Set-8	46.87	12.00	19.31	9.94	1.41	0.89	3.30	2.16
All	6.08	2.69	4.12	2.51	0.87	0.61	0.53	0.62
percentile speedups								
5%	0.55	0.30	0.55	0.32	0.28	0.19	0.07	0.14
50%	6.47	2.89	4.28	2.75	0.94	0.66	0.56	0.62
95%	62.93	18.77	26.34	15.50	2.28	1.57	3.92	2.75

at the 15th percentile. It wins against the sequential implementation for more than 85% of the instances. This highlights that the best GPU-parallel versions are not only significantly faster than the shared memory-parallel versions, but that they are also much more robust.
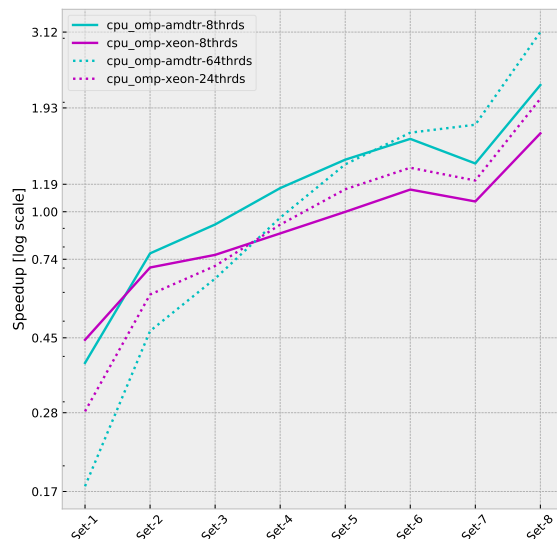

Figure 2: The Geometric mean of speedups over the eight subsets of instances of increasing size. Each curve belongs to one combination of algorithm and machine with a specific number of OpenMP threads used in the execution.

As mentioned earlier, the *cpu_omp* execution on *amdtr* machine uses 64 OpenMP threads, while the execution on *xeon* uses 24 threads. Given that scalability of these runs is suboptimal, one may suspect whether executions with a smaller number of threads would perform better due to reduced thread management cost. Figure 2 shows the same two executions with 24 and 64 threads we had before (dotted lines), plus the two executions *cpu_omp-amdtr* and *cpu_omp-xeon* with 8 threads each (solid lines). As expected, we can observe that the

executions with 8 threads perform favourably on smaller instances, where the thread management effort makes up a larger part of the total runtime. Eventually, as the size of instances grows, the algorithm manages to exploit enough parallelism to make the extra threads worthwhile, thus outperforming the 8 thread executions. However, these differences are still relatively small compared to the best performing GPU implementations. We conclude that manging the number of threads of the *cpu_omp* algorithm might help tune the performance of the shared memory implementation, but that the conclusions drawn with respect to the GPU implementations remain unchanged.

When it comes to the effects of hardware on the new *cpu_omp* executions with 8 threads, we can still observe that the lines cross, albeit earlier than before: the *xeon* execution outperforms *amdtr* on the smallest instances in Set-1. For the rest of the subsets, the *amdtr* execution wins over the *xeon* one. In the case of GPU algorithms, *gpu_atomic* consistently outperforms *gpu_reduction*. While the performance difference on *RTX* is a constant factor offset, on *V100* the difference even increases with speedup. This highlights the architectural differences between the *V100* and the *RTX*.

Finally, for all runs we can observe a steep increase of the speedup distribution after the 95th percentile. This indicates that for special applications GPU parallelization can have an even stronger impact beyond what our general observations over the heterogeneous MIPLIB 2017 test set suggest.

References

[1] T. Achterberg and R. Wunderling, *Mixed Integer Programming: Analyzing 12 Years of Progress.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 449–481.

[2] T. Koch, A. Martin, and M. E. Pfetsch, *Progress in Academic Computational Integer Programming.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 483–506. [Online]. Available: https://doi.org/10.1007/978-3-642-38189-8_19

[3] G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc., 1988. [Online]. Available: https://doi.org/10.1002/9781118627372

[4] P. Belotti, S. Cafieri, J. Lee, and L. Liberti, "Feasibility-based bounds tightening via fixed points," in *Combinatorial Optimization and Applications, Proc. of COCOA 2010*, W. Wu and O. Daescu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 65–76. [Online]. Available: https://doi.org/10.1007/978-3-642-17458-2_7

[5] T. Achterberg, "Constraint integer programming," Ph.D. dissertation, TU Berlin, 2009.

[6] V. Boyer, D. E. Baz, and M. Salazar-Aguilar, "GPU computing applied to linear and mixed-integer programming," *Advances in GPU Research and Practice*, pp. 247–271, 2017. [Online]. Available: https://doi.org/10.1016/B978-0-12-803738-6.00010-0

[7] M. W. P. Savelsbergh, "Preprocessing and probing techniques for mixed integer programming problems," *ORSA Journal on Computing*, vol. 6, pp. 445–454, 1994.

[8] D. Waltz, "Understanding line drawings of scenes with shadows," in *The Psychology of Computer Vision.* McGraw-Hill, 1975, p. pages.

[9] E. D. Andersen and K. D. Andersen, "Presolving in linear programming," *Mathematical Programming*, vol. 71, pp. 221–245, 1995. [Online]. Available: https://doi.org/10.1007/BF01586000

[10] J. Shectman and N. V. Sahinidis, "A finite algorithm for global minimization of separable concave programs," *Journal of Global Optimization*, vol. 12, pp. 1–36, 1998.

[11] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano, "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library," Optimization Online, Technical Report, 2019. [Online]. Available: http://www.optimization-online.org/DB_HTML/2019/07/7285.html

[12] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14.   IEEE Press, 2014, pp. 769–780. [Online]. Available: https://doi.org/10.1109/SC.2014.68

[13] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig, "The SCIP Optimization Suite 7.0," Optimization Online, Technical Report, March 2020. [Online]. Available: http://www.optimization-online.org/DB_HTML/2020/03/7705.html