Design Considerations for GPU-based Mixed Integer Programming on Parallel Computing Platforms

Kalyan Perumalla
Maksudul Alam
perumallaks@ornl.gov
alamm@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

ABSTRACT

Mixed Integer Programming (MIP) is a powerful abstraction in combinatorial optimization that finds real-life application across many significant sectors. The recent proliferation of graphical processing unit (GPU)-based accelerated computing architectures in large-scale parallel computing or supercomputing presents new opportunities as well as challenges in the advancement of MIP solver technology to effectively use the new accelerated computing platforms and scale to large parallel systems. Here, we recount the conventional processor-based strategies and focus on configurations where the most promising intersection lies between parallel MIP solver approaches and the specific strengths of accelerated parallel platforms. We note that the best potential lies in solving problems whose individual matrix sizes (of the linear program relaxation) fit entirely within one accelerator's memory and whose branch-and-bound (or branch-and-cut) trees cannot be fully contained within a small number of computational nodes. Additionally, we identify ideal features of computational linear algebra support on GPU accelerators that would help advance this direction of scalable parallel solution of MIP problems on GPU-based accelerated computing architectures.

CCS CONCEPTS

• **Mathematics of computing** → **Combinatorial optimization**; **Solvers**; **Mathematical software performance**; • **Computing methodologies**;

KEYWORDS

Accelerated computing, Parallel Computing, Mixed Integer Programming, Branch-and-Bound, Graphical Processing Units

ACM Reference Format:
Kalyan Perumalla and Maksudul Alam. 2021. Design Considerations for GPU-based Mixed Integer Programming on Parallel Computing Platforms. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3458744.3473366

ICPP Workshops '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-8441-4/21/08…$15.00
https://doi.org/10.1145/3458744.3473366

1 OVERVIEW

Mixed Integer Programming (MIP) is a powerful abstraction in combinatorial optimization that finds real-life application across many significant sectors [4, 21, 26, 37]. While MIP solver technology has seen remarkable advancements on conventional parallel processing hardware, it has not experienced similar progress in the latest accelerator-based parallel processing platforms. These latest parallel machines gain an immense amount of computational capacity from the graphical processing unit (GPU)-based acceleration for the most computationally intensive portions of the application. GPU-based execution offers immense potential for gains in speed, especially for linear algebra operations. More significantly, GPU-based execution is the only way to tap the core computational horsepower on many of the current parallel machines, making it inevitable to address GPU-based execution for MIP. Yet, there is inadequate amount of published literature on design considerations for realizing MIP solvers on large GPU-based parallel machines, in contrast to the large amount of literature and commerical/open-source software available for MIP solvers on traditional CPU-based architectures (such as SCIP[33, 34], BARON [18]). Here we attempt to bridge this gap between computational insights in such parallel platforms and the considerations in combinatorial optimization techniques when applied to those platforms.

Unfortunately, the special characteristics and idiosyncrasies of the GPU-based accelerated computing make it non-trivial to port existing MIP solver designs, algorithms, and implementations directly from CPU to GPU-based parallel execution. Important distinctions such as the restrictive single-instruction-multiple-data or SIMD style of execution of GPU (as opposed to the more general multiple-instruction-multiple-data or MIMD style of execution of multiple processor cores), and host-to-accelerator memory transfer costs complicate the MIP solver adaption from CPU-based to GPU-based execution. Compounding this problem is the difficulty of exploiting sparsity of the input matrix structure for the MIP problem. The problem of determining the most effective regimes for exterior-point methods as opposed to interior-point methods also arises more starkly in GPU-based computing because dense linear algebra is much more efficient on GPUs, and sparse matrix computations are generally not as efficient. There is the final challenge of finding adequate mathematical software support by the vendors for accelerated computing platforms; the linear algebra routines offered on those platforms are generally geared towards scientific computing and are mismatched for the unique uses of MIP solvers involving iterative updates, incremental updates and reuse of matrices being solved for the MIP problem.

In the next section, a brief background is presented for mixed integer programming and recent advancements in accelerator-based, large-scale parallel computing, along with coverage of related work. This is followed in Section 3 by an identification of different parallel execution strategies for MIP solution over accelerated parallel computing platforms. Key linear algebra support available on the GPU platforms is listed in Section 4 with selected packages suitable for MIP solvers. Section 5 captures important modes in which the linear algebra support of the GPUs is utilized for parallel MIP solution. The article is concluded in Section 6.

2 BACKGROUND

2.1 Mixed Integer Programming

Equation 1 shows the basic structure of a mixed integer program (MIP).

$$\begin{aligned} &\text{Maximize } c^T x \\ &\quad \text{such that } Ax \le b, \\ &\qquad \text{where } x = \{x_r, x_z\}, \\ &\qquad\quad x_r \in \mathcal{R} \text{ (reals), and} \\ &\qquad\quad x_z \in \mathcal{Z} \text{ (integers).} \end{aligned} \tag{1}$$

This formulation can be transformed into an equivalent form where the inequality of $Ax \le b$ can be replaced with equality ($A\hat{x} = \hat{b}$) with the introduction of variables $y \ge 0$ to capture the inequality slack. Also, upper and lower bounds, if any, on x are implicit in $Ax \le b$.

Many parallel solvers have experienced success in solving this formulation using a branch-and-bound approach that uses a divide and conquer strategy by relaxing the integrality of $x_z \in \mathcal{Z}$ to make $x \in \mathcal{R}$ and solving the relaxed problem, which is a linear programming problem without integer constraints. The linear program, if integer-infeasible, can be used to partition the solution space (for example, by branching on appropriate partitioning of one or more $x_i \in x_z$), which splits the problem into two or more sub-problems. The linear program relaxation provides an upper bound on the objective value of the corresponding integer problem. The branches create a tree of sub-problems that are accumulated for systematic evaluation. This basic branch-and-bound procedure is often enhanced with a range of variants such as branch-and-cut using dynamic cut generation with various types of cuts. The issues of branching, priming, variable value fixing, etc. have all been well studied in the past few decades and published in the literature.

The corresponding branch-and-bound search tree for a MIP is illustrated in Figure 1. All leaves in the tree are evaluated and tagged as feasible, infeasible or pruned. Intermediate nodes are tagged by their LP solutions and branching variables. Note that some leaves might be tagged as active during search. However, by the completion of the entire search, no nodes remain tagged as active – all of them are converted to feasible, infeasible or pruned.

A consistent snapshot of the branch-and-bound tree is defined as the set of leaves that preserves the optimal solution to the problem. Two simple consistent snapshots are easy to obtain: (1) the root node alone, and (2) the set of all leaves after the entire search. However, even during search, snapshots can be obtained that are consistent,

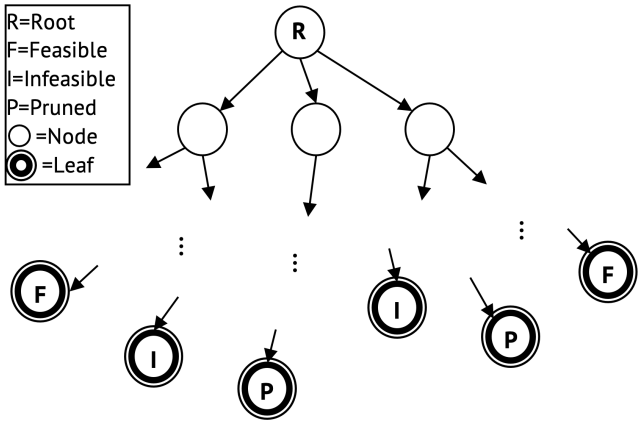

Figure 1: Solution tree

even though the optimal value is not yet found. In a sequential execution, a consistent snapshot is easy to obtain as follows. As soon as a node is solved and its children, if any, are generated, the active set is examined. The set of all leaf nodes from the active set constitute a valid consistent snapshot. In a parallel/distributed execution of branch-and-bound or branch-and-cut, a consistent snapshot is non-trivial to obtain. The complexity in this case comes from the fact that all processors need to synchronize with each other to account for (a) nodes that are being evaluated, i.e., whose LP relaxation is being computed, etc. (b) nodes that are in transit across processors, in the absence of a centralized active set, i.e., when a distributed scheme for exchange of active set nodes is used.

2.2 Parallel Computing Advancements

Parallel MIP solution methods traditionally have been designed and scaled on large parallel machines that were built with conventional CPU architectures. In recent years, however, the increases in computational capacity have been obtained not from increased number of CPU cores but from the addition of GPU-based accelerators. High performance computing in general has come to be dominated by GPUs so much so that GPUs are predominantly being used for scaling in much of the supercomputing world. In fact, the core computational capacity of seven of the top ten supercomputers is powered by GPUs[1]. The rate of performance gains using GPUs outpaces traditional CPUs by a wide margin. GPUs have become the core to modern AI-capable supercomputers. Summit, the fastest supercomputer in the USA (as of this writing) at the Oak Ridge National Laboratory uses NVIDIA Tesla V100 GPUs to achieve 200 petaFLOPS performance. The upcoming Aurora supercomputer at the Argonne National Laboratory is expected to have 1 exaFLOPS performance using Intel Xe GPUs. Frontier, the successor to Summit at the Oak Ridge National Laboratory is expected to deliver 1.5 exaFLOPS using AMD's Radeon Instinct GPUs. Lawrence Livermore National Laboratory also announced the El Capitan supercomputer to be launched in 2023 with 2 exaFLOPS using AMD's Radeon Instinct GPUs. This trend in the supercomputing arena provides a clear motivation for a focus on GPU-aware algorithms and packages

[1] http://top500.org

for wide range of scientific and mathematical problems. Furthermore, GPUs offer more energy efficient computing compared to the CPU counterpart. These performance efficiencies become more prominent as AI is increasingly added to the mix.

Given this background, it is clear that the immense scale of modern parallel machines can be effectively exploited for MIP solutions only if the GPU accelerators are correctly utilized and incorporated into the MIP solution approaches. Tied to this is the way the core computation in MIP solution is achieved, namely, linear algebra operations used in linear program relaxations and related computation.

2.3 Related Work

Linear programming solvers using an interior point method is the preferred method for solving sparse problems, which are prevalent in real-world scenarios. GPU based implementations of interior point methods have been proposed in [10, 17, 23]. Linear programming problems using dense matrices are well suited for the GPUs due to the nature of the GPU memory. Simplex and its many variants are typically used to solve dense problems. GPU-based implementation of Simplex algorithms have been discussed in [14, 16, 19, 24, 28, 31].

To the best of our knowledge there is no general-purpose MIP solver using GPU architectures. Ubiquity Generator (UG) framework [34] is a generic framework for parallelizing a branch-and-bound based MIP solver referred to as the base solver. It provides interfaces with different MPI protocols and also includes implementations of ramp-up, dynamic load balancing, and check-pointing and restarting mechanisms with a generic API. The base solver maintains the branch-and-bound tree. UG employs a Supervisor-Worker coordination mechanism for achieving parallelism with these sub-trees. UG manages a small number of sub-problems for load balancing. Solvers using UG have been developed for the non-commercial SCIP solver (ParaSCIP (=ug[SCIP, MPI]), FiberSCIP (=ug[SCIP, Pthreads]), the commercial Xpress solver (ParaXpress (=ug[Xpress, MPI]), and FiberXpress (=ug[Xpress, Pthreads]) [32]. ParaSCIP has successfully been used to solve 14 previously unsolved instances from MIPLIB2003 and MIPLIB2010. UG has been developed mostly with SCIP in mind. The largest-scale computation conducted with ParaSCIP uses up to 80,000 cores on TITAN at the Oak Ridge National Laboratory [33]. Note that these implementations have been focused on traditional CPU-based architectures. Initial attempts at porting CPU-based MIP solvers such as Gurobi to GPUs were abandoned because of the mismatch between SIMD style of execution of the GPUs and the existing code base for CPUs. However, they were aimed at realizing much of the branch-and-cut functionality on the GPU, which is not the best strategy, as will be described in the next section.

An early implementation of branch-and-bound for the knapsack problem was presented in [19]. An implementation of the branch-and-bound for the flow-shop problem on the GPU was presented in [5]. Another implementation of branch-and-bound version of the flow-shop problem was presented in [36]. Gmys et al. presented a pure GPU implementation of branch-and-bound algorithm in [13]. The key principle of their approach is the use of an Integer Vector Matrix (IVM) representation of the branch-and-bound problem tree rather than the linked list used in previous implementations. The IVM representation is well-suited for the GPU programming due to its memory structure.

3 PARALLEL EXECUTION STRATEGIES

Given the preceding background, there are different strategies possible for using the GPU-based large-scale parallel computing platforms for MIP, which include the following.

(1) Entirely GPU-based execution: The branch-and-cut tree is entirely stored and manipulated on the GPUs. Each branch-and-bound node is also solved on the GPU. This reduces needless transfers of data from CPU to GPU and back. It also can be fast if direct GPU to GPU communication is supported over the network by the parallel system architecture. However, this is one of the most challenging schemes to implement efficiently because of the difficulty of storing and manipulating very large trees on the GPU and fitting them within the limited confines of GPU memory. This approach also ignores the conventional CPU's processing power that is offered in combination with most GPU-based platforms. The single-instruction-multiple-data (SIMD) style of programming for GPUs also makes an entirely GPU-only solution less favorable; some of the open-source or commercial MIP solvers have indeed attempted and discounted GPU-only solution because of its faring poorly compared to the best CPU-based solvers.
(2) CPU-orchestration of GPU execution: The branch-and-cut tree is stored in the CPU main memory, while the GPU is used only as an accelerator for the computation of each branch-and-cut node which represents a linear program relaxation. This has the advantage of exploiting the large main memory of the CPU while also being efficient in tree handling, network communication, load distribution and similar functions that have been well understood and implemented in conventional parallel mixed integer programming (SCIP and UG). This allows the use of native and portable message passing interface-based parallel branch-and-cut orchestration across nodes.
(3) Hybrid CPU and GPU-based execution: In this mode, both the CPU and GPU architectures are employed for the heavy linear program relaxation solutions and other operations such as primal heuristics and cut generation. This mode is useful on modern architectures in which the CPUs comprise of many processor cores in addition to multiple GPUs serving as accelerators to the application running on the processor cores. The strength of this approach is the ease of implementing advanced heuristics such as probing, cut generation, column generation, etc. while also exploiting the concurrency offered by the many-core CPU architectures as well as the immense linear algebra efficiencies offered by the multi-GPU architectures.
(4) Big-MIP execution: In this strategy, the parallel machine is used as a bigger machine that can solve matrix sizes that cannot fit within a single node's memory. As in Earth-scale climate simulations, the matrix sizes can be so large that it is not possible to store the entire matrix on a single node,

and hence it is not possible to execute even a single branch-and-cut node (LP relaxation) on a single node. However, the basic matrix of the MIP problem could span many nodes; each LP relaxation itself, therefore, operates as a parallel matrix operation that spans multiple node in a distributed manner. In this strategy, one processor can act as the orchestrator of the serial branch-and-cut algorith, but each linear program relaxation is executed as a parallel job that utilizes the GPUs distributed across the parallel machine to perform each linear algebra operation (of the Simplex or its variants for the linear program solution of a branch-and-cut node) in a distributed manner. This is in contrast to the previously mentioned modes in which each LP relaxation fits entirely within one CPU or GPU's memory and hence the linear algebra operations on that relaxation are entirely contained within one machine. This is a useful mode for solving very large problems that have never been attempted before; however, the implementation can be very complex due to the need to not only store distributed branch-and-cut tree but also a distributed matrix across many nodes.

Considering all the current architectural factors in large GPU-based accelerated high performance computing platforms, we can choose the most effective strategy that has the most potential for effective scaling and efficient use of the GPU architectures. With the CPU and GPU memory sizes, interconnection network speeds, and the strengths of GPU-based linear algebra, it is clear that the strategy (2) or (3) are the most effective designs for gainful MIP solver implementations. Of the two, the least complex implementation is (2), namely, CPU-orchestration of GPU execution. GPU memory sizes have now reached 80GB at the time of this writing, which would only stand to increase even further in future. This amount of memory is sufficient to represent most MIPLIB problems entirely within a single GPU's memory. At the same time, the branch-and-cut trees are known to grow to such large sizes that the large capacity of CPU memory (which can be an order of magnitude greater in size, relative to GPU memory size) would be needed to hold the tree as it is being evaluated. High performance message passing interface implementations can be effectively tapped to perform inter-processor communication and synchronization for branch-and-cut tree handling even while the tree nodes are processed. GPU linear algebra routines are currently in a well developed state to allow very fast operation on any tree node's evaluation (of linear program relaxation). Sparse matrix solvers are, unfortunately, not as efficient on the GPUs, which may require strategy (3), namely, Hybrid CPU and GPU-base execution. With that strategy, sparse matrix computations (at least the set up stages) can be delegated to the multi-core processors which can be efficient for that purpose.

4 LINEAR SOLVERS ON GPU

In this section, we discuss the currently available linear algebra support with the latest GPU technology and how they can be utilized in the parallel GPU-based MIP solver code. Additionally, we describe features that would ideally complement the supported routines by adding functionality suitable for incremental updates to the matrix by the parallel MIP solver.

Linear solver using matrix factorization techniques such as Cholesky, LU, and QR decomposition is one of the most important computing routines for many scientific computing problems. Parallel processing of linear solvers on the GPU is a challenging problem due to high data dependency and irregular memory accesses.

4.1 Software Packages for Dense Matrices

There are multiple algorithms and software packages for dense linear algebra that can run on GPUs [1, 11, 25]. Some of these libraries provide basic BLAS functionality, such as NVIDIA's cuBLAS[2], AMD's rocBLAS[3], and Intel's MKL[4]. Some other packages provide a more comprehensive set of BLAS and LAPACK routines. MAGMA[5], is one of the prominent ones among them. It is an open source library that provides most BLAS kernels and LAPACK routines using hybrid CPU-GPU algorithms. The performance of the routines for the MAGMA dense matrrix solvers is very promising that can achieve approximately 80 percent of the GPU's theoretical peak performance [35]. NVIDIA's cuSOLVER[6] and AMD's rocSOLVER[7] also provide a few GPU-based LAPACK algorithms.

Generally speaking, accelerators were used to provide high performance compute-intensive BLAS routines. However, the development of GPU-only (accelerators-only) dense linear algebra algorithms were avoided in the past due to the fact that (1) accelerators were not well-suited for latency-sensitive tasks (such as factorization), and (2) hybrid algorithms were much faster, due to the fact that the CPU outperforms the GPU in such tasks. However, the recent advances in accelerators have opened the door for fully accelerators-based algorithms [3].

4.2 Software Packages for Sparse Matrices

Most of the packages support dense matrices for the factorization. However, there exist a few works targeting parallel sparse factorization using SIMD architecture. For instance, SuperLU_MT[8] is the multi-threaded parallel version of the LU decomposition in the CPU. KLU [9] is another implementation, which is specially optimized for circuit simulation. The KLU algorithm has been parallelized on multi-core architecture by exploiting the column-level parallelism [7].

A few sparse matrix LU factorization methods have also been realized on the GPU [8, 12, 22] by first converting the sparse matrices into many dense sub-matrices (blocks) and then solving them by dense matrix LU factorization. However, such a strategy may not work well for many real-world matrices, which hardly have dense sub-matrices. For example, circuit matrices are so sparse that BLAS-based methods are usually inefficient. As alternatives, parallel algorithms for LU decomposition on GPU have been explored [29], [6]. He et. al proposed a hybrid right-looking sparse LU factorization on the GPU, called GLU (GLU1.0) [15], which has later been revisited [20, 27]. There are publicly available software packages

[2] https://developer.nvidia.com/cublas
[3] https://rocsolver.readthedocs.io/en/latest/api_lapackfunc.html
[4] https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit.html
[5] https://icl.cs.utk.edu/projectsfiles/magma/doxygen/index.html
[6] https://docs.nvidia.com/cuda/cusolver/index.html
[7] https://github.com/ROCmSoftwarePlatform/rocSOLVER
[8] https://portal.nersc.gov/project/sparse/superlu/

that support sparse matrix computation, such as cuSPARSE[9], SuiteSparse (TAMU)[10], and rocSPARSE[11]. Note that many of the packages do not provide full support of routines required for sparse linear algebra computations. The LU solver in the libraries, such as cuSPARSE and rocSPARSE supplied by GPU manufacturers, provides relatively incomplete solver solutions. On the other hand MAGMA provides full support for sparse linear solvers. An up-to-date list of freely available software maintained by Dongarra et al is found at the Netlib website[12], which also includes sections on sparse direct solvers.

4.3 Special Considerations for Matrix Updates

In the GPU-accelerated methods presented thus far, computation is mainly performed via a single matrix operation at a time. Rennich, Stosic and Davis presented an approach to allow each GPU to work with many matrix operations at the same time [30]. Their approach batches together many small matrix updates in factorization that can be fit in the GPU memory. In general, packages that support batch matrix operation with a large number of small matrices (i.e. MAGMA) are desirable to take the full advantages of modern GPUs which have increased amount of high-bandwidth memory and thousands of SIMD cores offering a large amount of concurrency.

Note that our focus in this article is on MIP, although there are other related efforts on non-linear programming, which brings its own considerations and challenges that are qualitatively different from MIP's.

Given that an instance of $Ax = b$ has been solved (either implicitly or explicitly), there are three ways in which it needs to be reused in solving slightly updated versions (rank-1 updates) of that matrix: (1) the solution to the linear relaxation based on iterative exterior-point methods will need re-solves when variables enter and leave the basis, (2) cuts, which essentially are extra constraints, are generated and added to the matrix after a linear program relaxation has been solved, and (3) the matrix used for a parent node of the branch-and-cut can be reused for its children, with minor updates such as new bounds added for a subset variables.

One of the issues in accommodating such linear algebra updates is in eliminating or minimizing the data transfer latency and overhead between the host (CPU memory) and device (GPU memory). Another issue is the reuse of a previous solution as a starting point in iterative matrix factorization algorithms. Special algorithms and techniques are employed to avoid or reduce the number of such memory transfers and updates. In one such attempt, a modified product form of inverse was used in [28] and extended in [31]. Similar algorithmic techniques need to be considered for iterative decomposition methods.

5 USAGE MODES OF LINEAR ALGEBRA

The MIP solver operates on the branch-and-cut tree and launches the solution of the linear program relaxations (and other steps such as cut generation) as kernels that execute on the GPU. In this process, we identify three modes in which linear algebra manifests in the branch-and-cut solution process of parallel MIP solution: (1) iterative solution of the matrix corresponding to a linear program relaxation for a given branch-and-cut tree node, (2) modification and reuse of the matrix across generated cuts, (3) restoration and reuse of the matrix across tree nodes.

5.1 Iterative Solution of Linear Program Relaxation

In the first, linear algebra is used in the Simplex-based iterative solution (or one of its variants) of a branch-and-cut tree node, which is a sub-problem solved with linear program relaxation. There are several methods that have been employed over the years to optimize the solver on various computational architectures, problem variants, algorithmic variants (primal, dual, primal-dual, etc.). At the core, exterior point methods will involve modifications to the basic original matrix in the form of updates to entering and leaving basic variables. The net result of solution in each iteration is a feasible set of values assigned to the basic variables. Therefore, the GPU linear algebra will be exercised in this portion with rank-1 updates and resolving the updated matrix repeatedly with no data transfer from host to device or vice versa.

5.2 Incorporation of Generated Cuts

In the second mode of computation in which linear algebra is used, the matrix is updated with the incorporation of one or more cuts that are dynamically generated and added temporarily to the matrix for a particular tree node. We are not aware of any GPU-based cut generator published in the literature. Until GPU-based cut generators are developed, the cut generation can be assumed to be performed on the CPU, which will require the latest copy of the matrix (of the current branch-and-cut node) to be copied from the device to the host. After the generated cuts, if any, are transferred from host to device, they need to be incorporated into the device matrix before the linear algebra is again invoked on the GPU.

5.3 Reuse across Tree Nodes

In the third mode of computation in which linear algebra is used, the matrix is reused across the solution of different nodes of the branch-and-cut tree. Given two or more branch-and-cut nodes with a common ancestor that is only a few levels above them in the tree, it is possible to reuse the matrix across the nodes in solving their relaxations. Since it is cost-effective to minimize the number of transfers of the (potentially large) matrix between host and device memories, a GPU-based parallel MIP solver must strive to reuse the matrix on the GPU across as many branch-and-cut nodes as possible. This may warrant the use of a GPU-specific scheduling policy that picks the next node to evaluate from the branch-and-cut tree. It also influences how the tree nodes are exchanged and distributed across processors when the tree load is balanced dynamically. Thus, the design of a GPU-based MIP solver would entail choosing a branching scheme, a node evaluation ordering scheme, and so on that are qualitatively different from a traditional CPU-based solver's schemes.

[9] https://docs.nvidia.com/cuda/cusparse/index.html
[10] https://people.engr.tamu.edu/davis/suitesparse.html
[11] https://github.com/ROCmSoftwarePlatform/rocSPARSE
[12] http://www.netlib.org/utk/people/JackDongarra/la-sw.html

5.4 Sparse versus Dense Matrices

A critical consideration in using the GPUs is the nature of the matrix being solved in the MIP. Sparse matrix computation on the GPUs is not as efficient as dense matrix computation. Therefore, it would be necessary to create two distinct code paths, one for invoking spare matrix algebra and another for dense matrix algebra, when the branch-and-cut orchestrator invokes the GPU in the accelerated parallel MIP solver. While many CPU-based solvers operate, by default, on matrix inputs that are stored in sparse matrix format, this cannot be universally fixed for GPU-based execution. In fact, the code must handle user-provided inputs differently, based on whether the input matrix happens to be dense or sparse; this decision needs to be made at runtime, depending on the exact problem input by the user. Therefore, for the highest efficiency, two different MIP solver versions would need to be written: one specially built for sparse MIP problems and the other for dense MIP problems. Alternatively, a super-MIP solver for GPUs would need to be written which dynamically takes different code paths based on the input matrix characteristics.

5.5 Concurrent Solutions for Small Problems

In modern GPUs, the memory capacity has increased sufficiently to consider housing and solving multiple branch-and-cut nodes concurrently on the same GPU. Thus, for relatively small MIP problem sizes (such as those in which the matrix takes much less than tens of gigabytes of memory), it is conceivable (and potentially more efficient) to solve multiple nodes at a time. For example, if the matrix size fits in 1GB of memory; then, dozens of branch-and-cut nodes could be solved simultaneously by the GPU that has 64GB or more amount of memory. However, the linear algebra services on the GPU must support concurrent launches of multiple sub-problems on the same GPU. Such as support is offered on the NVIDIA GPUs with the concept of streams, such that multiple concurrent streams can be created and launched at a given time on the same GPU.

A batch routine executes the same operation on many independent matrices in a parallel fashion to take advantage of the GPU hardware resources. In the context of linear algebra, a batch routine applies some BLAS or LAPACK operation to a large number of small independent matrices. MAGMA provides support for batch operations on many of the BLAS and LAPACK operations [2].

In order to use this mode, there are two ways in which the execution can be structured. In one approach the same MPI rank (process mapped to a processor core) can make asynchronous launches of the linear program relaxation solver for each small problem, and collect the results when they terminate asynchronously on the GPU. This corresponds to a batch-style of processing of linear algebra calls. However, this mode is complex to implement due to the complexity of the solver method for linear program relaxation (such as Simplex, Dual-Simplex, etc.).

The other approach is to initiate multiple ranks per processor core so that there is little modification needed to the basic parallel engine which assumes a serial loop of work on linear program relaxation. However, this mode may not be supported on all systems because the runtime may limit the number of MPI ranks to be at most one per physical processor core.

6 SUMMARY

Mixed integer linear programs are an important class of optimization problems. Their high computational cost for solution demands the use of parallel computing platforms. While modern parallel computing platforms have started using GPUs as the new foundation blocks for scaling to large configurations, few MIP solvers exist that can effectively exploit those GPU-based platforms. Due to fundamental mismatches between traditional CPU-based execution and newer GPU-based execution, MIP solver designs need to be revisited to adapt and target scaling to large GPU-based high-performance parallel systems. We presented the strategies and design considerations behind such an evolution of the MIP technologies, and identified some of the promising strategies. We also highlighted key considerations at the implementation level, namely, at the level of linear algebra services offered by GPU platforms and the way their usage appears in MIP solver procedures. It remains to be seen how the next generation of MIP solvers scale to this new class of parallelism offered by modern supercomputers and how they can provide the next leap for MIP solutions.

ACKNOWLEDGEMENTS

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

ABOUT THE AUTHORS

KALYAN PERUMALLA is a Distinguished Research Staff Member at the Oak Ridge National Laboratory in the Computer Science and Mathematics Division.

MAKSUDUL ALAM is a Research Staff Member at the Oak Ridge National Laboratory in the Computer Science and Mathematics Division.

REFERENCES

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2018. Analysis and Design Techniques towards High-Performance and Energy-Efficient Dense Linear Solvers on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (dec 2018), 2700–2712. https://doi.org/10.1109/tpds.2018.2842785

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2018. Optimizing GPU Kernels for Irregular Batch Workloads: A Case Study for Cholesky Factorization. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. https://doi.org/10.1109/hpec.2018.8547576

[3] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2019. Progressive Optimization of Batched LU Factorization on GPUs. (sep 2019). https://doi.org/10.1109/HPEC.2019.8916270

[4] Pietro Belotti, Christian Kirches, Sven Leyffer, Jeff Linderoth, James Luedtke, and Ashutosh Mahajan. 2013. Mixed-integer nonlinear optimization. *Acta Numerica* 22 (apr 2013), 1–131. https://doi.org/10.1017/s0962492913000032

[5] Imen Chakroun, Nordine Melab, Mohand Mezmaz, and Daniel Tuyttens. 2013. Combining multi-core and GPU computing for solving combinatorial optimization problems. *J. Parallel and Distrib. Comput.* 73, 12 (2013), 1563–1577.

[6] Xiaoming Chen, Ling Ren, Yu Wang, and Huazhong Yang. 2015. GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling. *IEEE*

Transactions on Parallel and Distributed Systems 26, 3 (mar 2015), 786–795. Issue 3. https://doi.org/10.1109/tpds.2014.2312199

[7] Xiaoming Chen, Yu Wang, and Huazhong Yang. 2013. NICSLU: An adaptive sparse matrix solver for parallel circuit simulation. *IEEE transactions on computer-aided design of integrated circuits and systems* 32, 2 (2013), 261–274.

[8] D Yu Chenhan, Weichung Wang, et al. 2011. A CPU–GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Comput.* 37, 12 (2011), 759–770.

[9] Timothy A Davis and Ekanathan Palamadai Natarajan. 2010. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software (TOMS)* 37, 3 (2010), 1–17.

[10] Nicolai Fog Gade-Nielsen. 2014. Interior point methods on GPU with application to model predictive control. (2014).

[11] N. Galoppo, N.K. Govindaraju, M. Henson, and D. Manocha. 2005. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *ACM/IEEE SC 2005 Conference (SC'05)*. IEEE. https://doi.org/10.1109/sc.2005.42

[12] Thomas George, Vaibhav Saxena, Anshul Gupta, Amik Singh, and Anamitra R Choudhury. 2011. Multifrontal factorization of sparse SPD matrices on GPUs. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 372–383.

[13] Jan Gmys, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. 2016. A GPU-based Branch-and-Bound algorithm using Integer–Vector–Matrix data structure. *Parallel Comput.* 59 (2016), 119–139.

[14] Amit Gurung and Rajarshi Ray. 2019. Simultaneous solving of batched linear programs on a GPU. In *Proceedings of the 2019 acm/spec international conference on performance engineering*. 59–66.

[15] Kai He, Sheldon X-D Tan, Hai Wang, and Guoyong Shi. 2015. GPU-accelerated parallel sparse LU factorization method for fast circuit analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 3 (2015), 1140–1150.

[16] Lili He, Hongtao Bai, Yu Jiang, Dantong Ouyang, and Shanshan Jiang. 2018. Revised simplex algorithm for linear programming on GPUs with CUDA. *Multimedia Tools and Applications* 77, 22 (apr 2018), 30035–30050. https://doi.org/10.1007/s11042-018-5947-z

[17] Jin Hyuk Jung and DIANNE P OâĂŹLeary. 2008. Implementing an interior point method for linear programs on a CPU-GPU system. *Electronic Transactions on Numerical Analysis* 28, 174-189 (2008), 37.

[18] Mustafa Kılınç and NV Sahinidis. 2014. Solving MINLPs with BARON. In *MINLP Workshop, Pittsburgh http://http://minlp. cheme. cmu. edu/2014/papers/kilinc. pdf.*

[19] Mohamed Esseghir Lalami, Didier El-Baz, and Vincent Boyer. 2011. Multi GPU Implementation of the Simplex Algorithm. (sep 2011). https://doi.org/10.1109/HPCC.2011.32

[20] Wai-Kong Lee, Ramachandra Achar, and Michel S Nakhla. 2018. Dynamic GPU parallel sparse LU factorization for fast circuit simulation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 11 (2018), 2518–2529.

[21] JT Linderoth and TK Ralphs. 2004. Noncommercial Software for Mixed-Integer Linear Programming. (2004).

[22] Robert F Lucas, Gene Wagenbreth, Dan M Davis, and Roger Grimes. 2010. Multifrontal computations on GPUs and their multi-core hosts. In *International Conference on High Performance Computing for Computational Science*. Springer, 71–82.

[23] Marco Maggioni. 2016. *Sparse convex optimization on GPUs*. Ph.D. Dissertation. University of Illinois at Chicago.

[24] Xavier Meyer, Paul Albuquerque, and Bastien Chopard. 2011. A multi-GPU implementation and performance model for the standard simplex method. In *Proceedings of the 1st International Symposium and 10th Balkan Conference on Operational Research*. 312–319.

[25] Branko Lj Mrdakovic, Milan M Kostic, Dragan I Olcan, and Branko M Kolundzija. 2017. Acceleration of in-core LU-decomposition of dense MoM matrix by parallel usage of multiple GPUs. In *2017 IEEE International Conference on Microwaves, Antennas, Communications and Electronic Systems (COMCAS)*. IEEE, 1–4.

[26] James Ostrowski, Miguel F. Anjos, and Anthony Vannelli. 2012. Tight Mixed Integer Linear Programming Formulations for the Unit Commitment Problem. *IEEE Transactions on Power Systems* 27, 1 (feb 2012), 39–46. https://doi.org/10.1109/tpwrs.2011.2162008

[27] Shaoyi Peng and Sheldon X.-D. Tan. 2020. GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation. *IEEE Design & Test* 37, 3 (jun 2020), 78–90. https://doi.org/10.1109/mdat.2020.2974910

[28] Nikolaos Ploskas and Nikolaos Samaras. 2015. Efficient GPU-based implementations of simplex type algorithms. *Appl. Math. Comput.* 250 (jan 2015), 552–570. https://doi.org/10.1016/j.amc.2014.10.096

[29] Ling Ren, Xiaoming Chen, Yu Wang, Chenxi Zhang, and Huazhong Yang. 2012. Sparse LU factorization for parallel circuit simulation on GPU. In *Proceedings of the 49th Annual Design Automation Conference*. 1125–1130.

[30] Steven C Rennich, Darko Stosic, and Timothy A Davis. 2016. Accelerating sparse Cholesky factorization on GPUs. *Parallel Comput.* 59 (2016), 140–150.

[31] Usman Ali Shah, Suhail Yousaf, Iftikhar Ahmad, Safi Ur Rehman, and Muhammad Ovais Ahmad. 2020. Accelerating Revised Simplex Method Using GPU-Based Basis Update. *IEEE Access* 8 (2020), 52121–52138. https://doi.org/10.1109/access.2020.2980309

[32] Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. 2012. ParaSCIP: a parallel extension of SCIP. In *Competence in High Performance Computing 2010*, Christian Bischof, Heinz-Gerd Hegering, Wolfgang Nagel, and Gabriel Wittum (Eds.). 135 – 148. https://doi.org/10.1007/978-3-642-24025-6_12

[33] Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch, and Michael Winkler. 2016. Solving Open MIP Instances with ParaSCIP on Supercomputers Using up to 80,000 Cores. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. https://doi.org/10.1109/ipdps.2016.56

[34] Y. Shinano, M. Higaki, and R. Hirabayashi. 1995. A generalized utility for parallel branch and bound algorithms. In *Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing*. IEEE Comput. Soc. Press. https://doi.org/10.1109/spdp.1995.530710

[35] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. 2010. Dense linear algebra solvers for multicore with GPU accelerators. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE. https://doi.org/10.1109/ipdpsw.2010.5470941

[36] Trong-Tuan Vu and Bilel Derbel. 2016. Parallel Branch-and-Bound in multi-core multi-CPU multi-GPU heterogeneous environments. *Future Generation Computer Systems* 56 (2016), 95–109.

[37] Laurence A Wolsey and George L Nemhauser. 1999. *Integer and combinatorial optimization*. Vol. 55. John Wiley & Sons.