Scalable Algorithms for Deterministic Global Optimization

Robert Xavier Gottlieb, Ph.D.

University of Connecticut, 2026

ABSTRACT

Engineers frequently rely on mathematical models to describe and simulate the behavior of complex systems. For chemical engineers, these models are frequently nonconvex as a consequence of including complex physical phenomena. This nonconvexity creates challenges because the models can easily get stuck in suboptimal local minima. For tasks such as safety verification and robust design, it is critical that these suboptimal results are avoided in favor of globally optimal solutions. However, the deterministic global optimization algorithms that can provide guarantees of global optimality are highly computationally expensive, making them impractical to use on large-scale, industrially relevant problems. While there have been innumerable advances in these algorithms in recent years, a major bottleneck for this class of software has been the computing hardware on which it runs. Practically all available global solvers rely exclusively on a computer's CPU to perform calculations and are incapable of utilizing massively parallel computing hardware such as graphics processing units (GPUs). If global optimization tools could be adapted to run on GPUs, significant speed-ups could be obtained that would expand the reach of this powerful method. This thesis addresses the GPU-acceleration of two critical algorithmic components and the integration of those components into a general-purpose hybrid CPU-GPU global optimizer.

The first portion of this thesis focuses on the GPU acceleration of relaxation calculations and linear program solving. Each of these tasks is fundamental to the spatial branch-and-bound (B&B) method used by all major global optimizers. These tasks are numerous, but they are also irregular, comparatively fast, and are typically performed serially, posing challenges to their parallelization. I present a batched approach in which parallel hardware is used to perform many similar calculations simultaneously on different portions of a problem. This approach enables component-level speed-ups in each of these tasks.

In the second portion, I integrate the parallelized components together in an adapted B&B implementation. The resulting solver is general and can address the same class of problems that other CPU-based global optimizers can solve. Through incorporating the efficient GPU-based components, this combined solver demonstrates a clear improvement over the CPU-based solver on which it is based.

Scalable Algorithms for Deterministic Global Optimization

Robert Xavier Gottlieb

B.S., Rensselaer Polytechnic Institute, 2015

M.S., University of Wisconsin-Madison, 2017

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

in Chemical & Biomolecular Engineering

at the

University of Connecticut

2026

Doctor of Philosophy Dissertation

Scalable Algorithms for Deterministic Global Optimization

Presented by

Robert Xavier Gottlieb, B.S., M.S.

Approved by

Major Advisor	Prof. Matthew Stuber, Ph.D.
Associate Advisor	Prof. Burcu Beykal, Ph.D.
Associate Advisor	Prof. Baikun Li, Ph.D.
Associate Advisor	Prof. Jeffrey McCutcheon, Ph.D.
Associate Advisor	Prof. Ranjan Srivastava, Ph.D.

University of Connecticut

2026

Acknowledgments

Thinking back on the past several years, I am happy to say that I have greatly enjoyed my time as a graduate student. On a philosophical level, I have always been enamored with the idea of contributing to the scientific progress of the world, and this path has given me the time and opportunity to make some small, and hopefully meaningful, impact. On a more personal level, I think it is important to recognize that no Ph.D. is completed in a vacuum (figuratively speaking) and that there are a great number of people spanning the Venn diagram of friends, family, coworkers, colleagues, mentors, academic faculty and staff, and an assortment of other categories that have contributed to my success. At the risk of sounding like the winner of some Academy Award, I am incredibly grateful to have had the formal and informal support of such a wide array of people, and I would like to use the remainder of this space to thank a number of people individually.

To my Ph.D. advisor, Prof. Matthew Stuber, I owe a great deal of gratitude. Truly, I doubt that I could have found a better advisor. He is flexible in allowing his students to work on aspects of projects we find interesting, supportive of new ideas and approaches, and devoted to improvement in all things, which manifests in ways such as recommending improvements to administrative workflows he encounters only briefly, rewriting the entire graduate student handbook to create more clarity, and always being open to criticism. These and other aspects combine to make him an effective and, above all, reliable mentor. Incredibly, he is also a fount of knowledge on a seemingly endless variety of topics, ranging from formal math to formal wear. I learned quite a lot in my Ph.D., in no small part thanks to him. It was an absolute pleasure to work under his mentorship, and I am honored to have had the opportunity to work with someone of his caliber.

While on the topic of academic mentorship, I am grateful to each and every member of my advisory committee for their support, comments, and guidance throughout my Ph.D. Although my work took a fairly early turn toward the theoretical and software development side, I am thankful to have had Prof. Jeffrey McCutcheon on my committee, who was instrumental in keeping my work grounded to fundamental chemical engineering and reminding me to consider the engineering and

ethical aspects of my work. Similarly, I greatly valued having the perspective of Prof. Baikun Li, who helped me think practically about how my work could be applied to real-world engineering tasks. On the computational side, it was a pleasure to have Prof. Ranjan Srivastava as a part of my committee. His focus on "big picture" concepts ensured that I considered my work within the contexts of the broader optimization and chemical engineering communities I am a part of, which was important not only for the framing of my work in Chapter 1 and my thesis defense, but also for broadening my perspective on the potential impacts of my research. Finally, it is with much gratitude that I thank Prof. Burcu Beykal for joining my committee and especially for asking challenging questions during the yearly committee meetings. Such resistance is a crucial part of the Ph.D. process, and these questions were certainly a positive force in making my work more rigorous and requiring me to be cognizant of the literature. Each of my committee members provided a uniquely valuable perspective and I am honored to have had the opportunity to receive their feedback.

Formal academics aside, my Ph.D. gave me the chance to interact with a number of talented, good people that made the years of focused research bearable (and even fun!). First and foremost, I must thank Dr. Chenyu Wang for his instantly welcoming attitude toward me joining the lab, and for showing me around downtown Storrs and introducing me to his favorite local restaurants. His positive attitude and enthusiasm for life and research was infectious and set a strong example. Thanks are also no doubt earned by Dr. Matthew Wilhelm for getting me familiar with programming in Julia in our weekly meetings, and for his "go for it" attitude that influenced the path of much of my work. Matt and Chenyu set a quite positive precedent for work and interpersonal interactions in the Stuber group, which proved to be monumentally important given the largely remote nature of our work. Following their defenses, when I became the most senior group member, I did my best to follow in their footsteps and propagate the environment that I found so uplifting and valuable.

Relatedly, I must acknowledge the relatively newer members of the Stuber group for their undeniably positive contributions to my experience. To Pengfei Xu, I am grateful for the many great conversations we had throughout the years about life, politics, AI, grad school, and many other fun topics. These interactions turned our remote, computational research into a more personal and "human" experience, which I am sure did wonders for our overall sanity. I must also specially thank Dimitri Alston, who, if we count based on hours, probably accounts for a good 50–75% of my (non-family) interactions with anyone during my time at UConn. This is, of course, bolstered by our nearly weekly 3-or-more-hour meetings working on code, debugging and documenting everything we could touch, discussing movies, games, and other topics, and also by our mutual attendance of a slew of domestic and international conferences. Without doubt, Dimitri was a positive and influential

part of my time at UConn, for which I am absolutely grateful. To my coworker and friend Alireza Miraliakbar, I am thankful for the many conversations we've had and to have experienced your energy and enthusiasm that pushed the whole group forward and helped me to complete my work. Lastly, I must thank Dr. Nicole Beauregard, an unofficial and honorary Stuber group member, for the many cumulative hours spent getting coffee, playing chess, and giving and receiving presentation feedback, and for generally making UConn feel like a community. To all of my friends at UConn, thank you for making this such a positive, memorable experience.

Separate from my research life itself, I believe it is also important to give special thanks to my family for their support, and for collectively contributing to my upbringing in a way that led me to this point. Thank you, of course, to my parents, Peter Gottlieb and Dr. Jeri Riggs, for their proud love and support over the years and for inspiring my interest in math and science. To my brother, Michael, I thank you for being a role model in the pursuit of excellence and the search for challenging problems to solve. Sincere thanks are also due to my maternal grandparents, Margaret and David Riggs, for their role in instilling in me a sense of family and a love of the sciences, and to my paternal grandparents, Bernice and Ferdinand Gottlieb, for their unending love and support. Finally, it is unthinkable not to thank my wife, Kathleen, for practically everything. Her unwavering support, through multiple cross-country moves over the years, made it possible for me to embark on this path in the first place, and it allowed me to both focus on research at work and relax and enjoy my time outside of work. She also inexplicably managed to sit through every one of my practice presentations, sometimes multiple times for the same talk, and provided feedback that lifted everything she saw. Truly, this could not have been possible without you, and I thank you for putting up with all of this.

To wrap up this section, I would like to thank the Department of Energy, the National Science Foundation, and the University of Connecticut for their funding contributions to this work.

Things are only impossible until they're not.

-Jean-Luc Picard

(Sir Patrick Stewart)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Within process systems engineering, the ability to identify optima is critical for maximizing the effectiveness of a process. Chief among the set of optimal solutions are the global optima, so named because they represent solutions that are superior to all other possible or considered scenarios. A global optimum of a process may represent a set-up that yields the maximum possible profit, or the lowest possible environmental impact, or the least amount of generated waste. One branch of global optimization, known as deterministic global optimization (DGO), produces global solutions with a theoretical guarantee of optimality. Such techniques are of great importance in safety verification [134] and robust control applications [85], where a guarantee of global optimality can represent operation that is safe and resilient in the face of any possible realization of uncertainty. DGO can also be used, for example, in the task of model verification [127], to identify the presence of structural inadequacies in system models. In applications that serve the public good, such as wastewater treatment and related anaerobic digestion processes, applying mathematical optimization techniques to identify globally optimal process parameters can equate to more efficient, lower-maintenance, and lower-cost processes that safely and cheaply provide the maximum possible public benefit.

While the potential benefits are substantial, the primary drawback of DGO is its high computational cost. Particularly, DGO techniques are used to solve nonconvex nonlinear programs (NLPs)

of the form:

$$\begin{aligned} f^* = \min_{\mathbf{x}} \; & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \mathbf{x} \in X \subset \mathbb{R}^n, \end{aligned} \tag{1.1}$$

which belong to the NP-hard complexity class [65]. The typical method of solving such problems is the spatial branch-and-bound (B&B) algorithm [104], and stemming from the problem complexity, this algorithm exhibits worst-case exponential scaling with problem dimension. Consequently, while low-dimensional problems may potentially be fast to solve, B&B quickly becomes impractical to use on larger problems. This is problematic for process systems engineers since, for many problems of practical interest, models that capture relevant system properties tend to be nonconvex and high dimensional. Due to the worst-case runtime of the algorithm, such problems typically cannot be addressed with DGO in a practical amount of time.

One sparsely explored approach to address the high computational cost of global optimization is the use of specialized computing architectures. Since 2005, computational power has increasingly moved toward parallel processing [84], with specialty hardware such as graphics processing units (GPUs) being designed specifically to handle massively parallelized workloads. The internal design of GPUs enables them to perform compute-intensive tasks faster than traditional CPUs, and with greater energy efficiency per calculation [100], if the task can be efficiently performed in parallel. GPUs have already been utilized with great success for a number of scientific computing tasks, including machine learning model training [3] and generative AI [136], data analysis tasks [128], large-scale simulations including molecular dynamics [50] and computational fluid dynamics [38], and many others [47]. GPUs are also a major and growing component of supercomputing resources [2], which can only be effectively utilized by processes designed to run in parallel. However, despite the abundance of such success stories and the notorious computational expense of DGO methods, to the best of our knowledge there remains no mature DGO software, commercial or otherwise, that can utilize GPUs. Given the wide availability of GPUs and the computational bottleneck that limits the application of DGO, the development of an efficient, GPU-accelerated DGO method would greatly expand the reach of this class of techniques and allow this powerful optimization tool to be used more practically to address a greater number of real-world engineering problems.

The topic of this dissertation is the development of DGO techniques and algorithms that can utilize the relatively untapped hardware resource of GPUs. More specifically, standard implementations of spatial B&B are designed to run in serial, and are therefore not able to be run on the parallel GPU hardware. In this work, GPU-compatible implementations of major, computationally expensive B&B algorithmic components are created and then integrated together to construct a hybrid CPU-GPU B&B algorithm. The underlying hypothesis is that these components can individually be made compatible with, and performant on, parallelized GPU hardware, and that their combination will result in an optimizer capable of addressing the same classes and types of problems as traditional global optimizers, but with the added benefits of faster calculation throughput and cheaper, more energy-efficient scalability for addressing large problem instances.

1.2 Thesis Structure

The subsequent chapters of this thesis are organized in the following manner. In Chapter 2, relevant background information is presented on mathematics, global optimization techniques and components of spatial B&B, and GPU hardware. Chapters 3–5 then cover the development of GPU-accelerated versions of individual components in B&B, followed by Chapter 6, in which these components are integrated together into a full algorithm. Finally, the thesis is concluded in Chapter 7 with a summary of the research contributions made herein and perspectives on avenues for future research. A detailed summary of each chapter is as follows.

- Chapter 2 provides an overview of relevant hardware and mathematical preliminaries, starting with a discussion on graphics processing unit (GPU) hardware the tools that are used to interface with this hardware. It then continues with preliminary information on interval analysis and linear programming, before delving into a deeper discussion of DGO techniques covering the standard implementation of the spatial B&B algorithm, mathematical relaxations that build on interval arithmetic, and the use of linear programming within DGO for NLPs. Finally, it covers the challenges for GPU parallelization of DGO methods that this thesis aims to address.

- Chapter 3 details an exploratory approach for parallelizing the calculation of relaxations on GPUs. Calculating relaxations is a core and fundamentally critical element of spatial B&B, and by showing that this component could be made performant on GPUs, this chapter supports the notion that DGO could feasibly be adapted to this alternate hardware. This chapter also

demonstrates, for the first time to our knowledge, a minimal working example of a GPU-accelerated DGO solver using relaxations for bounding.

- Chapter 4 builds on the work in Chapter 3 by creating a more versatile, efficient, and user-friendly method of calculating relaxations on GPU hardware. Benchmarking is performed on a wide set of expressions from a library test set of typical global optimization problems, ultimately showing speed ups of up to four orders-of-magnitude.

- Chapter 5 explores the GPU acceleration of the typically small linear programs seen during global optimization. Parallelization is employed to solve batches of small problems simultaneously, as opposed to more typical GPU-accelerated linear program solvers that work on large, individual problems. The performance of this approach is demonstrated on a benchmark test set of linear program batches derived from typical global optimization problems, ultimately achieving solution speeds competitive with existing commercial and open-source solvers.

- Chapter 6 integrates together the research products of the previous chapters to create a general global optimizer for NLPs where the lower-bounding math is predominantly shifted to GPU hardware. The GPU-accelerated approach is built as an extension of the existing EAGO solver, and is shown to consistently outperform the original CPU-only approach across the entire benchmark test set.

- Chapter 7 wraps up this work with an overview of how the original challenges laid out in Chapters 1 and 2 have been addressed. This developments herein are contextualized within the broader global optimization community, and perspectives on future work and improvements are discussed.

Chapter 2

Computing and Mathematical Background

In this chapter, relevant background information is provided on computer hardware and mathematics. This information is necessary to describe the developments of later chapters, and it also allows for a more detailed description of what this work is meant to accomplish. Organizationally, relatively broad information is provided on graphics processing unit (GPU) hardware, interval analysis, and linear programming, which is followed by a more detailed dive into information more directly pertinent to deterministic global optimization.

2.1 Modern Computing

In 1965, Gordon Moore, one of the co-founders of Intel, published his prediction that the number of components on integrated circuits would continue to double every two years. This cadence, dubbed "Moore's law," has persisted into the modern computing era, but unlike in the late 20th century, this doubling no longer translates to ever-faster calculation speeds of individual microprocessors (i.e., the "clock speed" or "frequency" of CPUs). Rather, as a result of increasing power and cooling requirements for only incremental benefit, clock speeds reached a plateau in 2005 [84]. Instead, chip makers shifted their focus from creating increasingly powerful single microprocessors ("cores") to designing chips with multiple energy efficient cores, which could boost the overall performance of the system by handling tasks in parallel [48].

The second major driver in the growth of computing power was the emergence of heterogeneous

computing: the combination of general-purpose CPUs with specialized computing hardware built to accelerate performance in specific applications. The most common piece of such specialized hardware, the graphics processing unit (GPU), employs an architecture built for "SIMD" (single instruction, multiple data) or "SIMT" (single instruction, multiple threads) operation, with the distinction between these two described in greater detail in Section 2.1.1. Fundamentally, this architecture combines computing units together into groups that share an instruction set and operate on a shared block of memory. This design enables greater computing power for much lower power consumption and manufacturing cost than CPUs, though as a consequence of the design, applications must be tailor-made to work effectively with the hardware. Section 2.1.2 provides more detail on how applications can be made to work on GPUs, but briefly, to enable parallelization on a GPU, a process must be reconfigured so that identical, independent operations can be performed simultaneously on different subsets of data. However, redesigning computing tasks for GPU operation may not end up being faster or more efficient in practice. An initial point to consider is whether the number of calculations being performed is great enough that the expense of transferring data between CPU and GPU hardware can be outweighed by the faster overall calculation speed. Depending on the task, this requirement may be a substantial obstacle the limits the effectiveness of a GPU implementation.

Nonetheless, due to the paradigm shift in computing hardware toward parallelization by design, the major advances in computer hardware since 2005 can only be exploited by adapting algorithms for the new paradigm. Hardware accelerators such as GPUs, for example, can only be effectively utilized by software that was intentionally developed with the "SIMD"/"SIMT" architecture in mind. This development is not uncommon: there is a wide range of applications that have benefited from GPU-parallelized processing [47], including applications relevant to the process systems engineering (PSE) community such as the training of machine learning models [3] and for model predictive control (MPC) [4]. Examples such as these suggest that other presently serial PSE tasks, such as deterministic global optimization, may potentially find success with GPU parallelization.

2.1.1 GPU Hardware Architecture

To better understand the use of GPUs, it is important to first develop a firm understanding of GPU hardware. The hardware details of GPUs are discussed here in fairly general terms, as every new generation of GPUs features design changes and updates relative to previous generations. Additionally, unless otherwise noted, this discussion will be limited to GPUs produced by NVIDIA, as this is the type of GPU programmed for in this work.

At a high level, the parallelization ability of GPUs comes from the relatively large number of cores they possess. GPUs may have several thousand cores, as compared to CPUs that typically have on the order of ten cores. Loosely speaking, a core is capable of performing some kind of operation on data, such as multiplying two numbers, and by virtue of having more cores, a GPU can execute more of these operations simultaneously. However, GPUs are not simply CPUs with more cores. CPUs dedicate more space to data caching and control flow [110], which allows its cores to be both faster and more logically independent from one another than GPU cores. GPUs dedicate more space to data processing [110], which is accomplished by bundling cores into groups, where each group follows a consistent instruction set from a central source. Effectively, CPUs are built to execute serial instructions quickly, and GPUs sacrifice single-operation performance to obtain greater instruction throughput [110].

To facilitate further discussion, it is helpful to transition away from the concept of cores (the primary compute unit), and toward the concept of *threads* (the lowest-level execution unit) [101]. This distinction is helpful because GPUs operate by concurrently executing a large number of threads, and while the cores are performing the computations, they are constantly context-switching between different threads [101]. GPU hardware is also designed around the concept of threads moreso than cores: computing power in NVIDIA GPUs is segmented into hardware components called *streaming multiprocessors* (SMs), each of which supports 1024–2048 concurrent threads (depending on the Compute Capability of the specific device [110]). To organize these threads, SMs bundle them into groups of 32 called *warps* [86], which are then passed to available cores via a *warp scheduler*. The warp scheduler can rapidly switch between active warps, which enables large numbers of threads to be handled [101]. This rapid switching also allows high-latency operations to be hidden by switching to eligible warps while slow operations are in progress [101]. Notably, because warp schedulers issue instructions to groups of 32 threads simultaneously, the instructions for the whole group must generally be the same, which is a feature of GPUs that supports their efficiency. This is discussed in greater detail in Section 2.1.2.

Another important component supporting the hierarchical architecture of GPUs is the memory. Particularly, GPUs have a hierarchy of memory that aligns with the organization into threads and SMs. At the lowest level, each thread has access to a small number of *registers* that can store data for immediate use [101]. These registers are fast and are located physically close to the cores, which allows them to be used for the lowest-level operations. Registers for one thread are not accessible by other threads, but the volume of data they hold is comparatively small. At a higher level sits the *L1 cache*, which is private memory for each SM [101]. The L1 cache has a significantly larger volume

than the registers and is accessible to all threads within an SM, but is roughly an order-of-magnitude slower than registers [101]. Even higher is the *L2 cache* [110], sometimes also referred to as the *GPU RAM* (random access memory), which is significantly larger than the L1 cache (typically up to several gigabytes in space) and accessible by all SMs [101]. To operate on data originally stored in CPU memory, the data must first be transferred to the GPU RAM so that it can be accessed by other GPU components. Separately, there is also memory associated with the relatively modern tensor cores in GPUs, which is not utilized in this work, but which is described in greater detail in the GPU Glossary compiled by Modal [101].

2.1.2 Computing with GPUs

Although originally designed to parallelize graphics calculations for real-time 3D rendering, GPUs have been accessible to programmers for more general calculations since NVIDIA's 2006 release of *CUDA* (Compute Unified Device Architecture) [110]. The CUDA model assumes the user is on a heterogeneous system with one or more CPUs and GPUs, and it allows the CPU (host) to copy data to and from a GPU (device) and execute code on the GPU [110]. With this model, programmers are able to pass information to a device and then execute custom instructions, taking full advantage of the parallel capabilities of GPUs. Programming using CUDA typically takes place using a low-level language such as C++, but within the past decade it has become accessible in the higher-level Julia programming language through the `CUDA.jl` software package [16]. The work in this document primarily took place in Julia, with GPU acceleration enabled by the `CUDA.jl` package, though many of the programming concepts described in this section apply to CUDA more generally as well.

For clarity within this work, named blocks of computer code that execute well defined tasks will be referred to as *subroutines* (as opposed to the more typical *functions*, to help distinguish between computer code and mathematical expressions). If the subroutines are intended for GPU use, they will be referred to as *kernels*, or equivalently, because of the use of CUDA, *CUDA kernels*. More specifically, a kernel is a type of subroutine that is *launched* (initiated) on a CPU, but then executes on GPU (device) hardware. There are also subroutines that are called from the device and execute on the device, which are referred to as *device functions*. In large part due to `CUDA.jl`, kernels written in Julia are not syntactically different from typical CPU-based subroutines, but it is useful to create the distinction because of differences in their design and operation.

One of the more critical aspects of kernel development is the reliance on SIMD/SIMT architecture. While CPU-based subroutines are typically written to operate serially but can be specialized

to make use of parallel computing elements, such as multiple CPU threads, kernels should always be designed to execute on many parallel threads within a GPU. It is technically possible to write kernels that do not take advantage of this parallelization, but this results in substantial performance degradation as most of the GPU cannot be used. As described in Section 2.1.1, NVIDIA GPUs operate on a warp basis, issuing instructions to groups of 32 threads at a time. According to the taxonomy originally described by Flynn [44], this makes GPUs SIMD processors, though because each thread has its own registers, they fall under the subcategory of *array processors*, which are now referred to as SIMT.

From a programming perspective, CUDA and the SIMT design enable instructions to be written uniquely for individual threads, which allows for greater programming flexibility. For example, a kernel can dictate that the first 16 threads in a warp execute one instruction, and the next 16 threads execute a different instruction. However, as a SIMD processor, the hardware itself is incapable of issuing different instructions to different threads within one warp at the same time. One way GPUs can handle cases such as this is *predication*, where the warp scheduler will mask some threads to prevent them from executing the instructions [101]. For the preceding example, the first instruction would be given to all 32 threads in the warp with the latter 16 masked, and once that instruction has been executed, the second instruction would be given with the first 16 threads masked. Threads taking different execution paths in this way is referred to as *warp divergence*, and while it is possible to force warp divergence to occur due to the SIMT design, it fundamentally degrades the performance of the GPU [101]. The example effectively takes twice as long as a similar kernel that only executes one type of instruction, since the two instructions must be performed sequentially. To maintain strong performance, warp divergence should generally be avoided where possible.

In addition to thread-level control, CUDA allows programmers to specify the number and organization of threads called for any kernel. Particularly, threads are organized into *blocks* (or equivalently, *thread blocks*), and blocks exist within a *grid*. One consequence of this organization is that all threads within a block are scheduled onto the same SM [101]. This allows threads within a block to operate on shared data stored in the L1 cache, and it allows operations such as thread syncing, which pauses all threads within a block until every thread has reached a specified point in the kernel. This contrasts threads in different blocks, which can only share data through the slower global memory and which typically cannot be synced with one another [110]. By organizing threads in this way, individual threads can be identified using built-in ID variables associated with the placement of a thread within a block, and the placement of a block within the grid. With knowledge about the number of threads per block and the number of blocks, the programmer can use these ID variables

to point each thread to a unique portion of global data to ensure that all necessary data is eventually processed.

The organization into threads and blocks also has implications for kernel design. For example, GPUs have limits on the number of threads per block and the maximum number of active threads per SM, among other limits. At a kernel level, these limits may influence decisions about how shared memory is used, whether loops are necessary, and so on. At a hardware level, these limits may influence the GPU *occupancy*, which is the ratio between the number of active warps and the maximum number of active warps [101]. E.g., if a GPU with Compute Capability 11.0 is used, which has a maximum of 1536 resident threads per SM [110], and a kernel is launched with 1024 threads per block, then because all threads within a block are required to be scheduled to the same SM, each SM can run at most one block for a maximum theoretical occupancy of 1024/1536 (not counting memory effects). Many factors contribute to performance, and increasing occupancy does not always have a positive effect, but having occupancy that is too low may not fully utilize GPU resources and may limit the ability of the GPU to hide instruction latency [101].

While there is much to consider with individual kernel design, it may also be valuable to consider performance from the perspective of an overall algorithm. That is, if accomplishing a task with one kernel would be too memory intensive or difficult to optimize, the task may be split up into multiple smaller kernels. Each of these smaller kernels can be launched with their own launch configuration, so there is increased opportunity for individual portions of the larger algorithm to be run more optimally. Of course, splitting up an algorithm can also add computational cost. Globally stored data may need to be retrieved multiple times, if needed by multiple parts of the algorithm, which can add cost that would not exist with a single kernel. Additionally, there is an $\mathcal{O}(10)$ μs launch time for CUDA kernels. If the kernels are doing a large amount of work, this cost may be insignificant overall, but if the task is split into several kernels, this cost is paid once for each kernel. This overhead time can be particularly impactful if the algorithm being created requires multiple iterations: if each iteration calls several kernels, the individual launch times may become a sizable portion of the end-to-end algorithm run time [110]. These effects can be ameliorated in part through a model called *CUDA Graphs*, which can encompass a workflow containing multiple kernels (and other actions), pay the launch time once during instantiation, and then rerun the workflow with low overhead in future iterations [110].

Significantly more can be said about GPU programming, kernel design, CUDA graphs, and many other techniques, though the information provided thus far is sufficient to cover much of the work described in this document. For more detailed analysis of kernels and GPU-accelerated workflows,

tools such as NVIDIA's Nsight Systems [111] and Nsight Compute [109] provide valuable metrics and insights. For additional information on CUDA and GPUs in general, we highly recommend NVIDIA's CUDA guide [110] and Modal's GPU glossary [101].

2.2 Initial Mathematical Notation

To begin the transition into more mathematical concepts, it is helpful at this point to describe some basic notation that is used throughout the remainder of this document. In all remaining sections, scalar quantities are denoted by lower-case letters (e.g., a) and vectors are denoted by boldface lower-case letters (e.g., $\mathbf{a}$). A set B^n is defined as the Cartesian product $B^n \equiv B \times B \times \ldots \times B$ for $B \subset \mathbb{R}$. Further notation will be introduced as it becomes necessary within this chapter.

2.3 High-Level Optimization and Problem Classification

At an extremely high level, this work is focused on the task of optimization. In a general sense, optimization entails solving a problem such as:

$$\begin{aligned} &\text{minimize } f(\mathbf{x}) \\ &\text{s.t.} \quad \mathbf{x} \in D, \end{aligned} \tag{2.1}$$

where $D \subset \mathbb{R}^n$ is a nonempty closed set and the so-called *objective function* $f : A \to \mathbb{R}$ is a continuous function with $A \subset \mathbb{R}^n$ being a suitable set containing D [65]. *Global optimization* involves finding a point $\mathbf{x}^* \in D$ satisfying $f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in D$ [65]. Such a point is called a *global minimizer*, and the corresponding value $f(\mathbf{x}^*)$ is called the *global minimum* of f over D and is denoted by $\min f(D)$ [65]. Note that (2.1) may also be used generally to cover maximization problems due to the relationship:

$$\max f(D) = -\min(-f(D)).$$

In addition to global minima, depending on the problem, there may also be *local minimizers* and corresponding *local minima*, which do not satisfy $f(\mathbf{x}^*) \leq f(\mathbf{x}) \;\; \forall \mathbf{x} \in D$, but do satisfy $f(\mathbf{x}^*) \leq f(\mathbf{x})$ in a region immediately surrounding $\mathbf{x}^*$. Mathematically, let $\|\cdot\|$ denote the Euclidean norm in $\mathbb{R}^n$ and let $\epsilon > 0$ be a real number. Then, an (open) ϵ-neighborhood of a point $\mathbf{x}^* \subset \mathbb{R}^n$ is defined as the open ball:

$$N(\mathbf{x}^*, \epsilon) := \{\mathbf{x} \subset \mathbb{R}^n : \|\mathbf{x} - \mathbf{x}^*\| < \epsilon\}$$

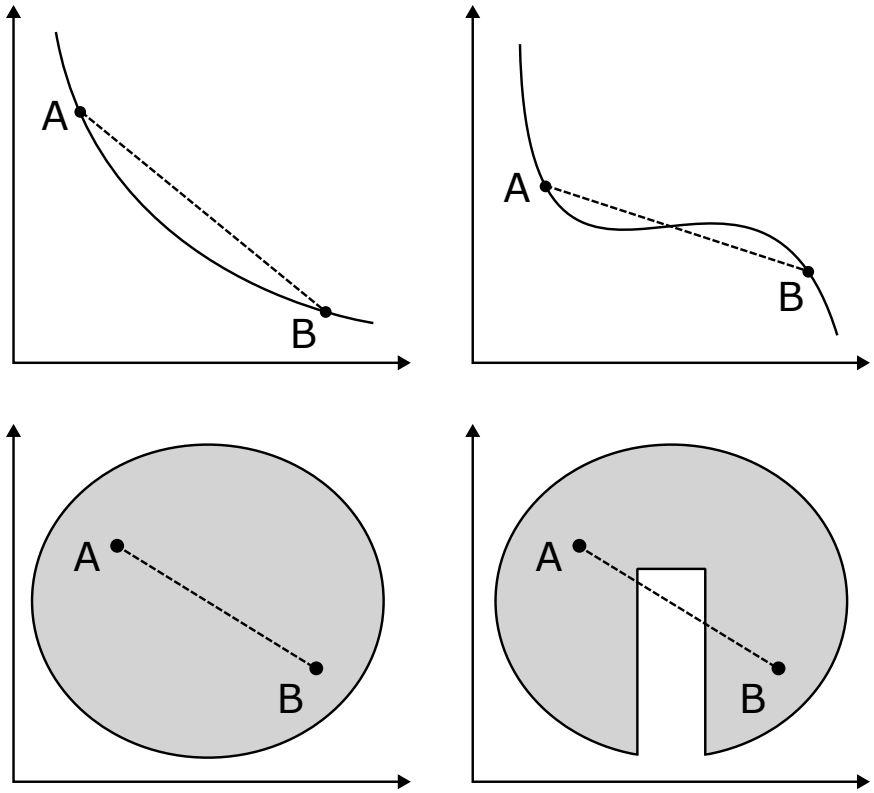

Figure 2.1: Examples of convex (top left) and nonconvex (top right) functions, and convex (bottom left) and nonconvex (bottom right) sets.

centered at $\mathbf{x}^*$ with radius ϵ [65]. Using this definition, a point $\mathbf{x}^*$ is a local minimizer of f over D if there is an ϵ such that $f(\mathbf{x}^*) \leq f(\mathbf{x}) \ \ \forall \mathbf{x} \in N(\mathbf{x}^*, \epsilon) \cap D$ [65]. Practically speaking, local minima may appear to be optimal, as slight perturbations around $\mathbf{x}^*$ would only increase or maintain the value of the objective function, but superior solutions may still exist elsewhere in D.

An important optimization concept related to the discussion on local vs. global optima is *convexity*. A function f is said to be convex if for every $\mathbf{x}, \mathbf{y} \in D$, the following relationship holds:

$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}), \quad \forall\lambda \in [0,1].$$

Similarly, a set D is convex if for every $\mathbf{x}, \mathbf{y} \in D$, $\lambda\mathbf{x} + (1-\lambda\mathbf{y}) \in D, \quad \forall\lambda \in [0,1]$. Verbally, a function is convex if you can draw a line segment between any two points on the function and the entire line segment will always lie entirely above the function, and a set is convex if every point on the line segment is always in the set. This concept is represented visually in Figure 2.1, with the left examples showing a convex function and set and the right examples showing a nonconvex function and set. There is an analogous definition for *concave* functions, which have the relationship that for

every $\mathbf{x}, \mathbf{y} \in D$, $f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \geq \lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}), \ \ \forall\lambda \in [0,1]$. Referring back to the general optimization problem of (2.1), if f is convex, then any local minimizer is also a global minimizer. If f is not convex, then verifying whether a local minimum is truly a global minimum can be a significant challenge and may require advanced (and computationally expensive) solution methods such as branch-and-bound (B&B). A more complete discussion of global optimization techniques and B&B is provided in Section 2.6.

With some preliminaries out of the way, we may continue by describing and differentiating several classes of optimization problems. Consider a more complex optimization problem taking the form:

$$\begin{aligned} \min_{\mathbf{z}} \ & f(\mathbf{z}) \\ \text{s.t. } & \mathbf{g}(\mathbf{z}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{z}) = \mathbf{0} \\ & \mathbf{l} \leq \mathbf{z} \leq \mathbf{u} \\ & \mathbf{z} = (\mathbf{x}, \mathbf{y}) \\ & \mathbf{x} \in \mathbb{R}^n \\ & \mathbf{y} \in \mathbb{Z}^m. \end{aligned} \tag{2.2}$$

In addition to the objective function f, we have now introduced a set of *inequality constraints* $\mathbf{g}$, a set of *equality constraints* $\mathbf{h}$, and lower and upper *bounds* of $\mathbf{l}$ and $\mathbf{u}$, respectively. Additionally, we have segmented the variables into real-valued ($\mathbf{x}$) and integer-valued ($\mathbf{y}$) variables, which we concatenate for simplicity as $\mathbf{z}$. This formulation is considerably more complex than (2.1), but it serves as a useful foundation for the ensuing classification as elements can be simplified or removed to create special cases of this more general form.

The following are classifications that can be made based on simplifications applied to (2.2). Note that this is by no means an exhaustive list, but it should serve as a useful high-level overview of some important classes of problems considered in the field:

- **Linear Program (LP)**. (2.2) is an LP if f, $\mathbf{g}$, and $\mathbf{h}$ are affine, and there are no integer-valued variables $\mathbf{y}$. LPs are the easiest class of optimization problems, and even problems with hundreds of millions of variables are practically solvable. Because affine functions are convex, LPs are inherently convex, and consequently any local solution to an LP is also a global solution. More detail is provided on LPs in Section 2.4.

- **Quadratic Program (QP)**. (2.2) is a QP if f is quadratic, $\mathbf{g}$, and $\mathbf{h}$ are affine, and there are no integer-valued variables $\mathbf{y}$. In a QP, f is typically expressed as $\frac{1}{2}\mathbf{x}^\top Q\mathbf{x} + \mathbf{c}^\top\mathbf{x}$. In this form, if Q is positive (semi)definite, then the QP is convex and can be solved in polynomial time [159]. If Q has at least one negative eigenvalue, however, the problem becomes NP-hard and can therefore no longer be solved in polynomial time (unless $P = NP$) [112].

- **Quadratically Constrained Quadratic Program (QCQP)**. Building on the definition of a QP, if (2.2) has f, $\mathbf{g}$, and $\mathbf{h}$ quadratic, and with no integer-valued variables $\mathbf{y}$, then it is a QCQP. Much like with QPs, the difficulty of a QCQP depends on whether the matrices describing all quadratic terms are positive semidefinite, which would make the problem convex.

- **Nonlinear Program (NLP)**. If f, $\mathbf{g}$, and $\mathbf{h}$ are permitted to contain any arbitrary nonlinear expressions, then (2.2) is an NLP. NLPs can be significantly more challenging than LPs and QPs, and even determining convexity of an NLP may be difficult. A number of techniques are regularly employed to solve NLPs to local optimality (i.e., terminating with a hopefully good local minimum), such as the interior-point method used in the widely available IPOPT solver [148]. This work focuses on solving NLPs to guaranteed global optimality, as discussed in greater detail in Section 2.6.

- **Integer Program (IP)**. (2.2) is considered an IP if there are no real-valued variables $\mathbf{y}$. IP is a general classification and can be further classified, such as by defining an integer linear program (ILP) where f, $\mathbf{g}$, and $\mathbf{h}$ are affine and all variables $\mathbf{y}$ are integers.

- **Mixed-Integer Program (MIP)**. Similar to IPs, MIP refers to a general classification of optimization problems that contain both real-valued variables ($\mathbf{x}$) and integer-valued variables ($\mathbf{y}$). The term MIP is also occasionally used interchangeably with MILP, to refer to linear programs with both real-valued and integer-valued variables. This definition can also be extended to mixed-integer versions of other problem types, such as MIQCQPs, which may contain quadratic f, $\mathbf{g}$, and $\mathbf{h}$ terms and both discrete and continuous variables. MINLPs are perhaps the most challenging class of optimization problems, effectively represented by the full form of (2.2). The work described in this document may be used with some modification to address MINLPs, but these specific modifications are outside the scope of what is presented herein.

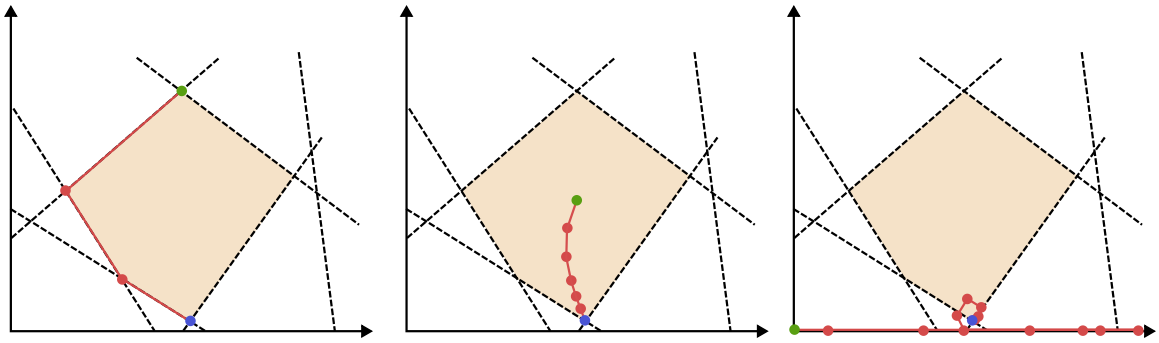

Figure 2.2: A graphical illustration of the simplex algorithm (left), interior point method (middle), and a first-order method (right) used to solve an LP. Dashed lines represent constraints, and the tan polytope represents the feasible region. Each method begins at a start point (green) and takes discrete steps (red) toward the solution of the LP (blue). The simplex algorithm moves between vertices of the polytope, the interior point method traverses the interior of the polytope, and the first-order method approaches and encircles the solution.

2.4 Linear Programming

Although a goal of this work is to address NLPs, as further described in Section 2.6, a typical solution method involves creating LP subproblems that provide information about the NLP. Consequently, a preliminary description of LPs and the typical solution methods used to solve them is of direct relevance. The standard form of an LP is given by the following [144]:

$$\begin{aligned} \min_{\mathbf{x}} \ & \mathbf{c}^\top \mathbf{x} \\ \text{s.t. } & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}. \end{aligned} \tag{2.3}$$

There are several commonly used methods to address (2.3). The first of these methods is the *simplex algorithm* (Figure 2.2, left), which can be understood as a method that proceeds by moving between vertices of the feasible region. Due to the affine nature of the objective and constraints in (2.3), the feasible region is a polytope, and if the solution exists and is unique, it is guaranteed to be at a vertex of the polytope. Typical implementations of the simplex method involve solving systems of equations with LU factorization [144]. While generally fast, even for large problems, the speed of factorization depends heavily on the sparsity pattern of the system and can be memory intensive [11]. A thorough workup of the simplex algorithm and some of its variants is covered by Vanderbei [144].

Another common method to address (2.3) is the *interior-point method* [71]. This method works

by adding slack variables $\mathbf{w}$ to (2.3) to convert the problem into equality form:

$$\begin{aligned} &\min_{\mathbf{x}} \mathbf{c}^\top \mathbf{x} \\ &\text{s.t. } \mathbf{A}\mathbf{x} + \mathbf{w} = \mathbf{b} \\ &\mathbf{x}, \mathbf{w} \geq \mathbf{0}, \end{aligned} \tag{2.4}$$

followed by replacing inequalities with extra terms in the objective function such as [144]:

$$\begin{aligned} &\min_{\mathbf{x}} \mathbf{c}^\top \mathbf{x} - \mu \sum_j \log x_j - \mu \sum_j \log w_i \\ &\text{s.t. } \mathbf{A}\mathbf{x} + \mathbf{w} = \mathbf{b}. \end{aligned} \tag{2.5}$$

This transformation has resulted in a *logarithmic barrier function* as the objective function [144]. Instead of constraints specifying that $\mathbf{x}$ and $\mathbf{w}$ are non-negative, this objective function increasingly penalizes values of $\mathbf{x}$ and $\mathbf{w}$ as they approach zero. The solution method effectively proceeds by starting with a large value of μ, which forces the solution of (2.5) to lie in the interior of (2.4) [144]. The value of μ is then decreased, which relaxes the logarithmic barriers, and the previous solution is moved toward the solution of the more relaxed version of (2.5) [144]. As μ approaches zero, the solution approaches the solution of (2.4), as visualized in Figure 2.2 (middle). As with the simplex method, steps within the interior-point method are calculated by solving linear system subproblems using factorization [11, 87].

As the primary focus of this work is GPU parallelization, it is valuable to pause briefly to reflect on GPU-parallel versions of the simplex and interior point methods. Although the approaches used by the two types of methods are different, both require matrix factorization, which is challenging to parallelize [11]. Nonetheless, there have been several reported successes of GPU parallelization for varieties of the simplex algorithm [17, 82, 96, 114, 123, 132] and interior point method [45, 70, 129] in the literature. These approaches utilize the parallel processing capabilities of GPUs to attempt to solve LPs with typically thousands of variables and constraints, with the relative performance over serial implementations increasing as the problem size increases. Commercial solvers were slow to adopt these methods, however, as GPUs were seen as poorly suited for handling sparse linear algebra until recent developments were made by NVIDIA in 2024 [108, 140]. At the time of writing, the Gurobi solver—widely considered to be a best-in-class commercial LP solver—now employs both CPU and GPU versions of a barrier method, but still exclusively uses CPU-based simplex algorithms

[25].

A third set of methods for solving LPs are *first-order methods* (FOMs), represented visually in Figure 2.2 (right). These methods are "first-order" in the sense that they rely only on gradient information to update their iterates [87]. Historically, FOMs were much less popular than simplex or interior point methods because they struggled to quickly obtain moderate or high-accuracy solutions [11]. However, they are advantageous in that their main computational bottleneck is matrix-vector multiplication [11], as opposed to the matrix factorization used in simplex and interior-point methods, which is reliably fast, not memory intensive, and more suited to extremely large-scale problems [105]. The use of FOMs for LPs experienced significant growth in 2022 with the development of a version of the primal-dual hybrid gradient (PDHG) method specifically adapted to LPs, referred to as *PDLP* [11]. As shown in Figure 2.2 (right), FOMs have a tendency to circle around the solution [87], which contributes to their difficulty obtaining highly accurate solutions. The PDLP method incorporated several enhancements, such as an adaptive restarting strategy, that ultimately made it competitive with existing LP solvers [11]. A more complete description of the PDLP algorithm is provided in Chapter 5.

From a GPU parallelization perspective, a notable advantage of PDLP is that it does not require the sparse linear algebra that made parallelized simplex and interior-point methods challenging. On the other hand, sparse matrix-vector multiplication is a significant strength of GPUs. The ease of parallelization was noted by Applegate et al. [11], and shortly after publication, a GPU-accelerated version of the algorithm named `cuPDLP.jl` was published [89]. As with the previously mentioned GPU-accelerated versions of simplex and interior-point methods, `cuPDLP.jl` was designed predominantly for large-scale LPs and experienced its strongest performance on the largest tested examples [89]. GPU-accelerated variants of PDLP were also quickly incorporated into commercial solvers such as Gurobi [25], reflecting the capability and value of the method.

For the purposes of this work, LP solution methods are useful insofar as they can be used as a component of a global optimization solver. The use of LPs for this purpose is described in greater detail in Section 2.6. For more complete descriptions of the simplex and interior-point methods, as well as the use of LPs more generally, Vanderbei [144] is an excellent resource. The PDLP method and its associated variants are still in active development with a number of papers devoted to its description and analysis [10, 88–91].

2.5 Interval Arithmetic and Interval Analysis

As a final preparatory section before delving into global optimization itself, this section will introduce an extension of real-valued arithmetic known as *interval arithmetic* and discuss some associated applications of this arithmetic via *interval analysis*. As with other sections, more full descriptions of interval analysis are covered by a number of resources including that of Moore [102]. At a high level, interval arithmetic is an approach designed to account for error. Measurements of real phenomena may have measurement error or uncertainty, and calculations on computers are subject to round-off error due to their finite-precision representations. Interval arithmetic accounts for these phenomena by utilizing an object called an interval, stylized as $[a, b]$ with $a, b \in \mathbb{R}$, which is effectively treated as a new kind of number [102]. By using intervals and interval arithmetic to evaluate a mathematical expression instead of real-valued numbers and standard arithmetic, the resulting interval can be guaranteed to rigorously bound the true result to infinite precision [102].

To expand upon the notation as it will be used in the remainder of this work, a nonempty compact set $A = \left[\mathbf{a}^L, \mathbf{a}^U\right]$ represents an n-dimensional interval defined as $A = \left\{\mathbf{a} \in \mathbb{R}^n : \mathbf{a}^L \leq \mathbf{a} \leq \mathbf{a}^U\right\}$, where $\mathbf{a}^L, \mathbf{a}^U \in \mathbb{R}^n$ with $\mathbf{a}^L \leq \mathbf{a}^U$ element-wise, with $\mathbf{a}^L$ and $\mathbf{a}^U$ being the lower and upper bounds of the interval, respectively. Additionally and for completeness, let $\mathbb{IR}^n$ be the set of all n-dimensional intervals, and for any $D \subset \mathbb{R}^n$, $\mathbb{I}D = \{X \in \mathbb{IR}^n : X \in D\}$ is the set of all interval subsets of D. We also describe the interval $[a, a]$ as being *degenerate*, effectively representing the real-valued number a. Through the notion of degenerate intervals, we can observe that real-valued arithmetic is a special case of interval arithmetic [102].

The most straightforward way to apply interval arithmetic is via so-called *natural interval extensions* [35]. By *interval extension*, we refer to the following definition:

Definition 2.5.1 (Interval Extension [102])**.** Let f be a real-valued function of n real variables $x_1, \ldots, x_n$. An *interval extension* of f is an interval-valued function F of n interval variables $X_1, \ldots, X_n$ with the property:

$$F(x_1, \ldots, x_n) = f(x_1, \ldots, x_n), \quad \text{for real arguments.}$$

From this definition, an interval extension of f is an interval-valued function which has a real value coinciding with f when its inputs are degenerate intervals representing reals [102]. For any scalar

function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$, its natural interval extension is the mapping $F(X)$ defined by:

$$F(X) = \text{hull}(\text{range}(f|X),$$

where the range$(f|X)$ denotes the range of f over the set X and hull(X) denotes the tightest member of $\mathbb{IR}^n$ that contains X [35]. As representative examples, the natural interval extensions of the basic arithmetic operations are as follows:

$$[a,b] + [c,d] = [a+c, b+d]$$

$$[a,b] - [c,d] = [a-d, b-c]$$

$$[a,b] \times [c,d] = [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)]$$

$$[a,b] \,/\, [c,d] = \begin{cases} \emptyset & \text{if } c = d = 0 \\ [a/d, b/c] & \text{if } 0 < a, b, c, d \\ [a/c, b/c] & \text{if } a < 0 < b, c, d \\ [a/c, b/d] & \text{if } a, b < 0 < c, d \\ [b/d, a/c] & \text{if } c, d < 0 < a, b \\ [b/d, a/d] & \text{if } a, c, d < 0 < b \\ [b/c, a/d] & \text{if } a, b, c, d < 0 \\ [-\infty, \infty] & \text{if } c < 0 < d \\ [a/d, \infty] & \text{if } c = 0, 0 < a, b, \\ [-\infty, \infty] & \text{if } c = 0, a < 0 < b \\ [-\infty, b/d] & \text{if } c = 0, a, b < 0 \\ [-\infty, a/c] & \text{if } d = 0, 0 < a, b \\ [-\infty, \infty] & \text{if } d = 0, a < 0 < b \\ [b/c, \infty] & \text{if } d = 0, a, b < 0. \end{cases}$$

Rules such as these are incorporated by software packages such as `IntervalArithmetic.jl` [120] in the Julia language. Further rules are defined for other standard math operations $(\log, \exp, \sin, \ldots)$, but are typically not explicitly defined for more complex expressions with multiple operators. In-

stead, for any given arbitrarily complicated math expression, the natural interval extensions of the component operations can be chained together, much like how real-valued arithmetic is performed, to obtain an interval extension of the original expression. Natural interval extensions calculated in this way are *inclusion monotonic*:

Definition 2.5.2 (Inclusion Monotonic [102])**.** An interval valued function F of the interval variables $X_1, \ldots, X_n$ is *inclusion monotonic* if

$$Y_i \subseteq X_i, \quad i = 1, \ldots, n,$$

implies

$$F(Y_1, \ldots, Y_n) \subseteq F(X_1, \ldots, X_n).$$

We can denote the image of $D \subset \mathbb{R}^n$ under the mapping $\mathbf{f}$ as $\hat{\mathbf{f}}(D)$, and then we reach the *Fundamental Theorem of Interval Analysis* [102, Theorem 3.1]:

Theorem 2.5.3 ([102])**.** *If F is an inclusion monotonic interval extension of f, then* $\hat{\mathbf{f}}(X_1, \ldots, X_n) \subseteq F(X_1, \ldots, X_n)$.

Recalling that real-valued arithmetic is a special case of interval arithmetic—when the interval is degenerate—we have the consequence that the real-valued result of the mapping $\mathbf{f}$ applied to any value $\mathbf{a} \in A$ is necessarily contained in $F(A)$.

Intervals calculated using natural interval extensions are straightforward to calculate, but they can be overly conservative in practice [160]. For example, natural interval extensions suffer from the *dependency problem*, which arises when multiple instances of the same variable in an expression become decorrelated with one another [27]. E.g., given an interval $X = [a, b]$, the expression $F(X) = X - X$ evaluates to $[a-b, b-a]$ rather than $[0, 0]$ [27]. One alternative to natural interval extensions is the *mean value form*, which is evaluated as:

$$F(X) := f(\mathrm{m}(X)) + \langle \nabla F(X), X - \mathrm{m}(X) \rangle,$$

where $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$, $\mathrm{m}(X)$ is the midpoint of the interval X, $\nabla F(X)$ is the natural interval extension of $\nabla f(\mathbf{x})$ over $X \subset \mathbb{I}D$, and $\langle \cdot \rangle$ is an inner product [160]. The mean value form uses natural interval extensions as part of its definition and is therefore also an inclusion monotonic interval extension, but it tends to produce tighter interval bounds in practice and exhibits a higher convergence order than natural interval extensions [160].

From an optimization perspective, interval arithmetic is useful in that it can provide rigorous information about the images of the objective function and constraints over an interval region of interest. This information can be used in a branch-and-bound scheme for global optimization purposes, as covered in Section 2.6, and it can also be used for *constraint propagation*. Constraint propagation routines attempt to use constraint information to shrink the initial enclosure of the domain of an optimization problem [151]. As an example, consider the trivial problem:

$$\begin{aligned} \min_{x,y} \; & x \\ \text{s.t. } & x + y \geq 3 \\ & x \in [0, 2] \\ & y \in [0, 2]. \end{aligned} \tag{2.6}$$

Much like how interval rules for math operations have been defined in the *forward* direction, rules have also been devised for propagating intervals in the *reverse* direction [151]. In (2.6), forward propagation of the interval domains of x and y for the constraint $x + y \geq 3$ yields $[0, 2] + [0, 2] = [0, 4] \geq 3$. Visually, it is possible that x or y could be tightened, as only a portion of $[0, 4]$ is above 3. Constraint propagation proceeds by intersecting the interval extension of the constraint with the interval domain enforced by the constraint, i.e., $[0, 4] \cap [3, \infty] = [3, 4]$. The intersected interval can then be propagated backwards through the addition operation to yield tighter intervals $x' = [1, 2]$ and $y' = [1, 2]$, effectively eliminating a region of infeasibility before any solving has taken place. While this example is trivial, constraint propagation can be of great benefit in many problems, sometimes reducing the initial domain substantially. Constraint propagation may also be used to eliminate problem domains that would cause numerical difficulties, such as by eliminating the negative portion of a domain in a problem utilizing the `sqrt` function.

Much more can be said about intervals and interval arithmetic, but for brevity, further study is left to the reader. Valuable resources include the textbook by Moore [102], as well as the IEEE-1788 Standard for Interval Arithmetic [35].

2.6 Deterministic Global Optimization

With the preliminaries thus laid out, we may turn our attention to the main focus of this work: deterministic global optimization (DGO). Briefly, this work seeks to address NLPs, which are challenging because their potential nonconvexity means they may have suboptimal local solutions. For

chemical engineers in particular, problems of practical interest are frequently nonconvex as a consequence of representing physical phenomena. While in some cases local optima may suffice, there are several contexts for which a globally optimal solution is required. One such task is model verification [127], in which an engineer investigating a process model seeks to determine whether the model is incorrect. If the model is calibrated against measured data and does not provide a strong fit, the model could be structurally insufficient, or alternatively the calibration could have resulted in a suboptimal local minimum. By applying global optimization, the engineer can verify that the calibration is as strong as possible, such that if there is still a mismatch between the data and the model, the model must be inadequate in some way. Other important applications of DGO include safety [134] and robust control [85]. For example, an engineer may want rigorous mathematical certainty that a given reactor with a specified control scheme will never exceed some predetermined concentration, temperature, or pressure thresholds that could create hazards or create an inferior product.

DGO methods are a class of approaches that solves optimization problems—typically NLPs or MINLPs—to guaranteed ϵ-global optimality. These methods are *guaranteed* in the sense that they are mathematically rigorous and provide proof of ϵ-global optimality, and the term *ϵ-global* is used because solutions are guaranteed to within some *convergence tolerance*, ϵ¿0. I.e., any solution obtained by a DGO algorithm is guaranteed to have an objective value within ϵ of a true global optimum. Such methods rely on the ability to generate global information about a problem, such as through interval methods, which allows one to compute bounds of the objective function [65]. The most popular DGO method for NLPs and MINLPs is *spatial branch-and-bound* (B&B), a variant of which is used by most major global optimizers [104].

2.6.1 Spatial Branch-and-Bound

Consider a nonconvex NLP:

$$f^* = f(\mathbf{x}^*) = \min_x f(\mathbf{x}) \qquad (2.7)$$
$$\text{s.t. } \mathbf{x} \in \mathcal{F} \subset \mathbb{R}^n,$$

where $f : D \to \mathbb{R}$ is the objective function and $\mathcal{F} \subset D \subset \mathbb{R}^n$ is the *feasible set*. Assuming that f^* exists and it is possible to compute suitable lower bounds for the minimum of the image of f (i.e., $\min \hat{f}(M)$) for at least some sets $M \subset D$, a B&B method can be applied [65]. As laid out by Horst

Algorithm 1 Spatial Branch-and-Bound (B&B)

1. Start with a relaxed feasible set $M^{(0)} \supset \mathcal{F}$, such that $M^{(0)} \in \mathbb{I}D$ and partition $M^{(0)}$ into finitely many subsets $M^{(i)}, i = 1, \ldots, n_p$.

2. For each i, determine lower and (if possible) upper bounds $LBD^{(i)}$ and $UBD^{(i)}$, respectively, satisfying:
$$LBD^{(i)} \leq \inf \hat{f}(M^{(i)} \cap \mathcal{F}) \leq UBD^{(i)}.$$
Then $LBD := \min_i LBD^{(i)}$, $UBD := \min_i UBD^{(i)}$ are *global* bounds, meaning that
$$LBD \leq \min \hat{f}(\mathcal{F}) \leq UBD.$$

3. If $UBD = LBD$ (or, for some prescribed $\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}} > 0$, we have $UBD - LBD \leq \varepsilon_{\text{abs}}$ or $|UBD - LBD|/\max(|UBD|, |LBD|) \leq \varepsilon_{\text{rel}}$), then stop.

4. Otherwise, select some of the subsets $M^{(i)}$ and partition these chosen subsets to obtain a refined partition of $M^{(0)}$. Return to Step (2) to determine new, hopefully tighter global bounds using the refined partition.

and Tuy [65], the basic idea of the spatial B&B framework is summarized in Algorithm 1.

Conceptually, and as it is typically implemented, the algorithm begins (Step 1) by considering a domain $M^{(0)}$ that contains $\mathcal{F}$, such as $\mathbb{IR}^n$ intersected with the variable bounds in (2.7) and not accounting for any nontrivial constraints. (Step 2) We then proceed by attempting to find *lower and upper bounds*, $LBD^{(i)}i$ and $UBD^{(i)}$, respectively, which contain the infimum of the image of f over the feasible portion of $M^{(0)}$. One way that $LBD^{(i)}$ might be obtained is through interval arithmetic, as $\hat{f}(M^{(0)} \cap \mathcal{F}) \subseteq F(M^{(0)} \cap \mathcal{F})$ (Thm 2.5.3) implies that $\mathbf{f}^L(M^{(0)} \cap \mathcal{F}) \leq \inf \hat{f}(M^{(0)} \cap \mathcal{F})$. It is possible to use interval arithmetic to calculate $UBD^{(i)}$ as well, but this would be overly conservative as we only seek to bound $\inf \hat{f}(M^{(0)} \cap \mathcal{F})$, not $\hat{f}(M^{(0)} \cap \mathcal{F})$ itself. In fact, finding $UBD^{(i)}$ can be fairly trivial, as $\inf \hat{f}(M^{(0)} \cap \mathcal{F}) \leq f(x) \ \ \forall x \in M^{(0)} \cap \mathcal{F}$. That is, evaluating f at any feasible point in $M^{(0)}$ provides a sufficient upper bound, whereas the lower bound required more mathematical effort. Step 2 continues by identifying *global* lower and upper bounds, LBD and UBD, respectively, which are simply the minimum-identified lower and upper bounds. (Step 3) If $UBD - LBD$ are equal or within some predefined tolerance, then we have identified $\min \hat{f}(\mathcal{F})$ to guaranteed ϵ-global optimality.

If the B&B algorithm has not terminated, the algorithm continues with Step 4, which involves refining $M^{(0)}$ into a partition. Individual elements of the partition are typically stored together in what is known as a *stack*, or equivalently, a *B&B stack*. The algorithm then returns to Step 2 where, typically, each iteration sees one element $M^{(i)}$ selected from the partition to be processed in the bounding step. As Algorithm 1 proceeds, the partition becomes further refined, which—

under some assumptions to be discussed shortly—leads the algorithm to converge to a solution, if one exists. As a note, the partitioning in Step 4 is a process referred to as *branching*. Branching is most commonly accomplished by bisecting the "current" partition element in one dimension of the decision variables (i.e., the variable is "branched on"). This partitioning strategy is commonly visualized as an undirected graph and as a B&B *tree* whose *nodes* are elements of the partition. When an element of the current partition is branched, the depth of the B&B tree increases with the addition of two child nodes. All terminal nodes are referred to as "leaves" and collectively comprise a partition of $M^{(0)}$. If it can be determined that $LBD^{(i)}$ for a given node $M^{(i)}$ is greater than UBD, the leaf can be discarded from the partition in a process known as *fathoming*. Nodes can also be fathomed if, for example, $M^{(i)} \cap \mathcal{F} = \emptyset$. If every node in the tree is fathomed in this way, the problem is said to be *infeasible*.

The reader is directed to Wilhelm and Stuber [155, Alg. 3.1] for a detailed B&B algorithm as it is implemented in the EAGO solver. Summarily, EAGO follows the framework described by Algorithm 1 by selecting one partition element per iteration, chosen based on which node maintains the current global lower bound, and performing lower- and upper-bounding operations on that node before branching and performing a fathoming operation. For the lower bound, EAGO uses McCormick relaxations (Sec. 2.6.2) to generate relaxed LP subproblems (Sec. 2.6.3), which are then passed to a dedicated LP solver. For the upper bound, EAGO follows a heuristic wherein nodes are occasionally sent as subproblems to a dedicated NLP solver, with the result becoming the upper bound for that node, or if the subproblem is not passed along, the midpoint of the node is evaluated for feasibility. If the midpoint is feasible, the value of the objective function at that point is used as the upper bound for that node.

It is valuable here, as much of the work in this thesis makes use EAGO, to show that EAGO exhibits finite convergence. I.e., the B&B algorithm implemented in EAGO is guaranteed to converge in some finite amount of time. To begin, we note that in the spatial B&B algorithm, one or more variables are continuous, and thus it is possible successively refine partition elements infinitely, thereby making the B&B procedure in EAGO infinite [65, Cor. IV.1]. To show that this infinite procedure converges, we first show that the bounding operation in EAGO is *consistent* [65]:

Definition 2.6.1 (Consistent Bounding Operations [65])**.** A bounding operation is called *consistent* if at every step any unfathomed partition element can be further refined, and if any infinitely

decreasing sequence $\{M_q^{(i)}\}$ of successively refined partition elements satisfies

$$\lim_{q\to\infty}(\min\{UBD_k^{(i)}\}_{k=1}^{q} - LBD_q^{(i)}) = 0.$$

The bounding operation in EAGO is consistent because the McCormick relaxations (Sec. 2.6.2) used in its lower-bounding procedure have pointwise convergence of order at least as high as the natural interval extensions used, which are known to have linear convergence [20]. This ensures that for any infinitely decreasing sequence $\{M_q^{(i)}\}$, $LBD_q^{(i)}$ is a nondecreasing sequence. Additionally, $\min\{UBD_k^{(i)}\}_{k=1}^{q}$ is guaranteed to be nonincreasing, and if $M^{(i)} \cap \mathcal{F} \neq \emptyset$, then EAGO's upper bounding routines, which occasionally call an NLP solver or otherwise evaluate a point within each node to identify feasible points with an improved upper bound, will ensure that $UBD^{(i)}$ will, at some finite iteration, obtain a finite value. We then assert that the node selection operation in EAGO is *bound improving* [65]:

Definition 2.6.2 (Bound Improving Operation [65])**.** A selection operation is said to be *bound improving* if, at least each time after a finite number of steps, the elements of the partition selected for further processing $\mathcal{P}_k$ satisfies the relation

$$\mathcal{P}_k \cap \operatorname{argmin}\{LBD : M \in \mathcal{R}_k\} \neq \emptyset,$$

where $\mathcal{R}_k$ is the set of unfathomed partition elements. I.e., at least one partition element where the global lower bound is attained is selected for further partitioning in iteration k of the algorithm.

In its bound selection operation, EAGO always selects a node with the global lower bound, thereby making it a bound improving operation. Because we have a consistent bounding operation and a bound improving selection operation, EAGO is convergent by Theorem IV.3 in Horst and Tuy [65].

As an important note regarding the B&B algorithm, the procedure relies on further partitioning existing elements of the partition along typically one dimension. This method is therefore particularly susceptible to the *curse of dimensionality*. That is, as the problem dimensionality grows, the search space becomes exponentially larger. In the worst case, exponentially more branches are required to obtain small nodes and, consequently, bounds that are tight enough to either converge or become fathomed. Because of the curse of dimensionality, global optimization is limited in practice to relatively small problems, with solvable problems typically featuring fewer than 20 variables. The convergence rate of the bounding information is known to affect the practical convergence time of the algorithm: if the convergence rate is too low, B&B methods may encounter the so-called *cluster*

effect [40, 150], in which large numbers of nodes are created near global minima. Because each of these nodes must be processed to obtain bounding information, the exponential number of these nodes that are created can greatly extend the solution time.

2.6.2 McCormick Relaxations

With the main spatial B&B algorithm described in Section 2.6.1, we may now begin to more specifically address the lower-bounding routine, which is the most expensive component in B&B implementations more broadly [30, 147]. Although it is possible to use interval arithmetic to obtain lower bounds (Sec. 2.5), the linear convergence rate of natural interval extensions makes them impractical for use in spatial B&B due to the cluster effect [40, 150]. One notable aspect of interval arithmetic that suggests the potential for improvement is that the intervals $F(X)$ are defined over a set X but do not take into account the structure of $f(x)$ at individual points $x \in X$. A logical extension of this idea is described in Definition 2.6.3.

Definition 2.6.3 (Convex and Concave Relaxations [99])**.** Given a convex set $Z \subset \mathbb{R}^n$ and a function $f : Z \to \mathbb{R}$, a convex function $f^{cv} : Z \to \mathbb{R}$ is a *convex relaxation* of f on Z if $f^{cv}(\mathbf{z}) \leq f(\mathbf{z})$ for every $\mathbf{z} \in Z$. A concave function $f^{cc} : Z \to \mathbb{R}$ is a *concave relaxation* of f on Z if $f^{cc}(\mathbf{z}) \geq f(\mathbf{z})$ for every $\mathbf{z} \in Z$.

While interval extensions guaranteed that $\hat{f}(X) \subseteq F(X)$, a convex relaxation is defined such that $f^{cv}(x) \leq f(x) \quad \forall x \in X$, and analogously for concave relaxations. Relaxations are therefore more flexible than interval extensions, and we can guarantee that they are at least as tight as intervals simply by intersecting $[f^{cv}, f^{cc}]$ with F. Additionally, it is important that these relaxations are convex and concave. Recall that one of the challenges with solving NLPs is that they may be nonconvex, and may therefore have suboptimal local minima. If we can generate a convex relaxation of the objective function of (2.7), then because it is convex by definition, the minimum of the convex relaxation is its global minimum. And, because it is a relaxation as defined by Definition 2.6.3, we can guarantee that at $\mathbf{x}^* = \min_{\mathbf{x}} f^{cv}(\mathbf{x})$, $f^{cv}(\mathbf{x}^*) \leq \hat{f}(X)$. Thus, if we can create convex and concave relaxations of (2.7), we can calculate tighter lower bounds than what is possible through natural interval extensions alone. Note that convex and concave relaxations of vector-valued functions $\mathbf{f} : Z \to \mathbb{R}^m$ are defined by applying the inequalities in Definition 2.6.3 componentwise [133].

The remainder of this section will describe the style of relaxations known as *McCormick relaxations*, for which a more formal definition will follow. This style of relaxation is named after McCormick [95], though that work has been expanded upon by a number of authors, as will be

detailed. To proceed toward a definition of McCormick relaxations, we will start by covering several other important terms. First, while infinitely many convex and concave relaxations may exist, we can more specifically define the tightest such relaxations:

Definition 2.6.4 (Convex and Concave Envelope [157])**.** Let $f : Z \to \mathbb{R}$ where $Z \subset \mathbb{R}^n$ is a nonempty convex set. The *convex envelope* of f on Z is the convex relaxation $f^{cv,env} : Z \to \mathbb{R}$ such that $f^{cv}(\mathbf{z}) \leq f^{cv,env}(\mathbf{z})$ holds for all $\mathbf{z} \in Z$ and every convex relaxation f^{cv} of f on Z. Similarly, the *concave envelope* of f on Z is the concave relaxation $f^{cc,env} : Z \to \mathbb{R}$ such that $f^{cc}(\mathbf{z}) \geq f^{cc,env}(\mathbf{z})$ holds for all $\mathbf{z} \in Z$ and every concave relaxation f^{cc} of f on Z.

Convex and concave envelopes are the most valuable relaxations that can be obtained for optimization purposes, but in practice, the mathematical forms of envelopes are only known for a handful of simple expressions. For additional context, although McCormick [95] originally demonstrated that relaxations of arbitrarily complicated expressions can be generated through composition of relaxations with similar terms, envelopes of some functions were required. Mitsos et al. [99] then formalized this process within the context of automatic/algorithmic differentiation to decompose these expressions into component parts for which convex and concave relaxations are known, without restriction to envelopes. The following definitions describe the manner in which expressions are decomposed.

Definition 2.6.5 (Univariate Intrinsic Function [121])**.** The function $u : B \subset \mathbb{R} \to \mathbb{R}$ is a *univariate intrinsic function* if, for any $A \in \mathbb{I}B$, the following are known and can be evaluated computationally:

- an inclusion monotonic interval extension of u on A,
- a convex relaxation of u on A,
- a concave relaxation of u on A.

Definition 2.6.6 (Factorable Function [121])**.** A function $\mathscr{F} : Z \subset \mathbb{R}^n \to \mathbb{R}$ is *factorable* if it can be expressed in terms of a finite number of factors $v_1, \ldots, v_m$, such that given $\mathbf{z} \in Z$, $v_i = z_i$ for $i = 1, \ldots, n$, and v_k is defined for $n < k < m$ as either

- $v_k = v_i + v_j$, with $i, j < k$, or
- $v_k = v_i v_j$, with $i, j < k$, or
- $v_k = u_k(v_i)$, with $i < k$, where $u_k : B_k \to \mathbb{R}$ is a univariate intrinsic function,

and $\mathscr{F}(\mathbf{z}) = v_m(\mathbf{z})$ for every $\mathbf{z} \in Z$. A vector-valued function is factorable if each of its components is a factorable function.

Definition 2.6.7 (Cumulative Mapping [121])**.** Let the *cumulative mapping* v_k be the mapping $v_k : Z \to \mathbb{R}$ defined for each $\mathbf{z} \in Z$ by the value $v_k(\mathbf{z})$ when the factors of $\mathscr{F}$ are computed recursively, as per Definition 2.6.6, beginning from $\mathbf{z}$.

Within the context of this work, we focus specifically on NLPs with their objective and constraints being factorable functions. This is a broad characterization covering a large majority of typically encountered expressions. Using Definitions 2.6.6 and 2.6.7, even if a complex expression of interest does not have known relaxations, as long as the expression is factorable, it can be expressed as a cumulative mapping of simpler factors for which relaxations are known. The relaxations of these factors may then be integrated together to develop relaxations of the more complex expression through the notion of *composite relaxations*.

Definition 2.6.8 (Composite Relaxations [133])**.** Let $D \subset \mathbb{R}^n$, $Z \in \mathbb{I}D$, and $P \in \mathbb{IR}^{n_p}$. Define the mapping $\mathcal{G} : D \times P \to \mathbb{R}^{n_\mathcal{G}}$. The functions $\mathbf{u}_\mathcal{G}, \mathbf{o}_\mathcal{G} : \mathbb{R}^n \times \mathbb{R}^n \times P \to \mathbb{R}^{n_\mathcal{G}}$ are called *composite relaxations* of $\mathcal{G}$ on $Z \times P$ if for $\boldsymbol{\psi}^{cv}, \boldsymbol{\psi}^{cc} : P \to \mathbb{R}^n$, the functions $\mathbf{u}_\mathcal{G}(\boldsymbol{\psi}^{cv}(\cdot), \boldsymbol{\psi}^{cc}(\cdot), \cdot)$ and $\mathbf{o}_\mathcal{G}(\boldsymbol{\psi}^{cv}(\cdot), \boldsymbol{\psi}^{cc}(\cdot), \cdot)$ are, respectively, convex and concave relaxations of $\mathcal{G}(\mathbf{q}(\cdot), \cdot)$ on P for any function $\mathbf{q} : P \to Z$ and any pair of convex and concave relaxations $(\boldsymbol{\psi}^{cv}, \boldsymbol{\psi}^{cc})$ of $\mathbf{q}$ on P.

Definition 2.6.8 is somewhat nuanced, but it is useful to formalize the notion of constructing convex and concave relaxations of composite functions on the domain of the inner function. Particularly, this makes explicit the relationship of convex and concave relaxations of an outer function on its domain, with respect to convex and concave relaxations of an inner function on its domain. Clarifying this relationship is important because the convex and concave relaxations of an inner function on its domain may potentially be known a priori, or calculated by some other algorithmic procedure. In that case, through the notion of the composite relaxation, those relaxations can simply be treated as convex and concave relaxations of a cumulative mapping on its domain, which can then be used to define convex and concave relaxations of a composite function involving the cumulative mapping. As a consequence, this definition enables algorithmic procedures to exist that can calculate relaxations for any factorable function. To hopefully add clarity with an example, consider the function $h(x) = \sin(\log(x))$. Despite convex and concave envelopes of $h(x)$ not being known, Definition 2.6.8 allows us to use any convex and concave relaxations of $\log(x)$ and substitute them into the calculation of convex and concave relaxations of $\sin(x)$ to ultimately calculate relaxations of

$h(x)$. I.e., it is not a requirement that convex and concave relaxations are calculated for the entirety of $h(x)$ all at once.

We continue by defining another concept important for the use of relaxations in optimization:

Definition 2.6.9 (Subgradients [133])**.** Let $Z \subset \mathbb{R}^n$ be a nonempty convex set, $f^{cv} : Z \to \mathbb{R}$ be convex, and $f^{cc} : Z \to \mathbb{R}$ be concave. A vector-valued function $\boldsymbol{\sigma}_f^{cv} : Z \to \mathbb{R}^n$ is called a subgradient (function) of f^{cv} on Z if for every $\bar{\mathbf{z}} \in Z$, $f^{cv}(\mathbf{z}) \geq f^{cv}(\bar{\mathbf{z}}) + (\boldsymbol{\sigma}_f^{cv}(\bar{\mathbf{z}}))^{\mathrm{T}}(\mathbf{z} - \bar{\mathbf{z}})$, $\forall \mathbf{z} \in Z$. Likewise, a vector-valued function $\boldsymbol{\sigma}_f^{cc} : Z \to \mathbb{R}^n$ is called a subgradient (function) of f^{cc} on Z if for every $\bar{\mathbf{z}} \in Z$, $f^{cc}(\mathbf{z}) \leq f^{cc}(\bar{\mathbf{z}}) + (\boldsymbol{\sigma}_f^{cc}(\bar{\mathbf{z}}))^{\mathrm{T}}(\mathbf{z} - \bar{\mathbf{z}})$, $\forall \mathbf{z} \in Z$.

As a point of clarification, by Definition 2.6.9, we consider $\boldsymbol{\sigma}_{[\cdot]}$ to be a *function* that is evaluated at a respective point to yield a corresponding subgradient. Within this work, we will include the evaluation point as an argument when it cannot be determined from context and, for simplicity, we will refer to the function $\boldsymbol{\sigma}_{[\cdot]}$ simply as a subgradient throughout this work. Subgradients are not unique in general, and they provide useful information to optimizers even in cases where relaxations are nondifferentiable. It is common for optimizers to use evaluations of relaxations and their subgradients at points of interest $\bar{\mathbf{z}}_i \in Z_{\mathrm{ref}} \subset Z$, where Z_{ref} is a nonempty discrete set, to construct supporting hyperplanes of the form [104]:

$$f_i^{cv,\mathrm{aff}}(\mathbf{z}) = f^{cv}(\bar{\mathbf{z}}_i) + (\boldsymbol{\sigma}_f^{cv}(\bar{\mathbf{z}}_i))^{\mathrm{T}}(\mathbf{z} - \bar{\mathbf{z}}_i).$$

From Definition 2.6.9, these hyperplanes are guaranteed to be below f^{cv} on Z, which makes them valuable for calculating a valid lower bound on the minimum of f^{cv} and, consequently, f. The notion of hyperplanes is utilized more directly in Section 2.6.3. Similar to the concept of relaxations, subgradients of relaxations may be calculated for individual factors and composed together to calculate relaxation subgradients for any factorable function using the notion of *composite subgradients* [99].

Definition 2.6.10 (Composite Subgradients [133])**.** Let $D \subset \mathbb{R}^n$, $P \in \mathbb{IR}^{n_p}$, and $Z \in \mathbb{I}D$. Let $\mathbf{q} : P \to Z$ and $\mathcal{G} : D \times P \to \mathbb{R}^{n_\mathcal{G}}$. Let $\mathbf{u}_\mathcal{G}$, $\mathbf{o}_\mathcal{G}$ be composite relaxations of $\mathcal{G}$ on $Z \times P$. The functions $\mathcal{S}_{u_\mathcal{G}}$, $\mathcal{S}_{o_\mathcal{G}} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^{n_p \times n} \times \mathbb{R}^{n_p \times n} \times P \to \mathbb{R}^{n_p \times n}$ are called *composite subgradients* of $\mathbf{u}_\mathcal{G}$ and $\mathbf{o}_\mathcal{G}$ on $Z \times P$, respectively, if for any $\boldsymbol{\psi}^{cv}$, $\boldsymbol{\psi}^{cc} : P \to \mathbb{R}^n$ and $\boldsymbol{\sigma}_\psi^{cv}, \boldsymbol{\sigma}_\psi^{cc} : P \to \mathbb{R}^{n_p \times n}$, the functions $\mathcal{S}_{u_\mathcal{G}}(\boldsymbol{\psi}^{cv}(\cdot), \boldsymbol{\psi}^{cc}(\cdot), \boldsymbol{\sigma}_\psi^{cv}(\cdot), \boldsymbol{\sigma}_\psi^{cc}(\cdot), \cdot)$, and $\mathcal{S}_{o_\mathcal{G}}(\boldsymbol{\psi}^{cv}(\cdot), \boldsymbol{\psi}^{cc}(\cdot), \boldsymbol{\sigma}_\psi^{cv}(\cdot), \boldsymbol{\sigma}_\psi^{cc}(\cdot), \cdot)$ are, respectively, subgradients of $\mathbf{u}_\mathcal{G}(\boldsymbol{\psi}^{cv}(\cdot), \boldsymbol{\psi}^{cc}(\cdot), \cdot)$ and $\mathbf{o}_\mathcal{G}(\boldsymbol{\psi}^{cv}(\cdot), \boldsymbol{\psi}^{cc}(\cdot), \cdot)$, provided $\boldsymbol{\psi}^{cv}$ and $\boldsymbol{\psi}^{cc}$ are, respectively, convex and concave relaxations of $\mathbf{q}$ on P and $\boldsymbol{\sigma}_\psi^{cv}$ and $\boldsymbol{\sigma}_\psi^{cc}$ are, respectively, subgradients of $\boldsymbol{\psi}^{cv}$ and $\boldsymbol{\psi}^{cc}$ on P.

As with the definition of composite relaxations (Definition 2.6.8), this formalized notion of composite subgradients enables the automatic computation of relaxation subgradients for arbitrarily complicated factorable functions.

With these more general preliminaries covered, we may now proceed to more explicitly describe McCormick relaxations. We begin by creating a *McCormick object*, which is a useful construct for holding multiple related data used in the calculation of relaxations.

Definition 2.6.11 (McCormick Object)**.** Let $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$. At any single $\mathbf{x} \in X \in \mathbb{IR}^n$, convex and concave relaxations of $\mathbf{f}$ on X, denoted by $\mathbf{f}^{cv}(\mathbf{x})$ and $\mathbf{f}^{cc}(\mathbf{x})$, and the lower and upper bounds on $\mathbf{f}(\mathbf{x})$ for all $\mathbf{x} \in X$, denoted by $\mathbf{f}^L$ and $\mathbf{f}^U$, can be collectively represented by the *McCormick object* $\mathbf{f}^\circ(\mathbf{x}) = (\mathbf{f}^{cv}(\mathbf{x}), \mathbf{f}^{cc}(\mathbf{x}), \mathbf{f}^L, \mathbf{f}^U)$. The space of McCormick objects is given by:

$$\mathbb{MR}^n \equiv \left\{(\mathbf{x}^{cv}, \mathbf{x}^{cc}, \mathbf{x}^L, \mathbf{x}^U) \in \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n : \mathbf{x}^L \leq \mathbf{x}^U\right\}$$

Moreover, for $D \subset \mathbb{R}^n$, define $\mathbb{M}D \equiv \left\{(\mathbf{x}^{cv}, \mathbf{x}^{cc}, \mathbf{x}^L, \mathbf{x}^U) \in \mathbb{MR}^n : [\mathbf{x}^L, \mathbf{x}^U] \subset D\right\}$.

The McCormick object we define by Definition 2.6.11 is functionally equivalent to that defined by Ye and Scott [158], despite differences in organization. The construction of the McCormick object is natural due to the interrelationships between its component elements, and moreover, it is convenient to think of the calculation of McCormick objects of expressions as opposed to individual elements of the McCormick object. We note that it is typically (and implicitly) assumed that $\mathbf{f}(\mathbf{x})^{cv} \leq \mathbf{f}(\mathbf{x})^{cc}$ and $[\mathbf{f}^L, \mathbf{f}^U] \cap [\mathbf{f}(\mathbf{x})^{cv}, \mathbf{f}(\mathbf{x})^{cc}] \neq \emptyset$ for all $\mathbf{x} \in X$. However, Ye and Scott [158] developed an approach to handle cases when this assumption is violated, which provides information about subproblem infeasibility, which we do not cover in this work. With this object defined, we are now prepared to define McCormick relaxations.

Definition 2.6.12 (Mid Function [99])**.** Given three numbers $\alpha, \beta, \gamma \in \mathbb{R}$, the mid function is defined as

$$\text{mid}\{\alpha, \beta, \gamma\} = \begin{cases} \alpha & \text{if } \beta \leq \alpha \leq \gamma \text{ or } \gamma \leq \alpha \leq \beta, \\ \beta & \text{if } \alpha \leq \beta \leq \gamma \text{ or } \gamma \leq \beta \leq \alpha, \\ \gamma & \text{if } \alpha \leq \gamma \leq \beta \text{ or } \beta \leq \gamma \leq \alpha. \end{cases}$$

Definition 2.6.13 ($S_{\mathbf{\Phi}}$, $\mathbf{\Phi}$, $\mathscr{V}$, $\widetilde{\mathbf{\Phi}}$ [121])**.** Let $S_{\mathbf{\Phi}} \subset \mathbb{R}^n$, let $\mathbf{\Phi} \subset S_{\mathbf{\Phi}}$, and let $\mathscr{V}$ be a collection of m factors such that, given any $\boldsymbol{\phi} \in \mathbf{\Phi}$, $v_k = \phi_k$ for each $v_k \in \mathscr{V}$ with $k = 1, \ldots, n$, and for each k such that $n < k \leq m$, $v_k \in \mathscr{V}$ is defined by one of the three rules in Definition 2.6.6. Moreover, define

the following set:

$$\widetilde{\mathbf{\Phi}} \equiv \left\{ \mathbf{y}^\circ \in \mathbb{MR}^n : [y_1^L, y_1^U] \times \cdots \times [y_n^L, y_n^U] \subset \mathbf{\Phi} \right\}.$$

Definition 2.6.14 (Generalized McCormick Relaxations [121])**.** Define the *Generalized McCormick relaxations* of the factors $\mathscr{V}$ on $\mathbf{\Phi}$ as the functions $\mathscr{U}, \mathscr{O} : \widetilde{\mathbf{\Phi}} \to \mathbb{R}$ where, for each $\mathbf{y}^\circ \in \widetilde{\mathbf{\Phi}}$, $\mathscr{U}(\mathbf{y}^\circ)$ and $\mathscr{O}(\mathbf{y}^\circ)$ are defined by the following procedure:

1. Set $v_i^\circ = y_i^\circ$ for all $i = 1, \ldots, n$, and denote $V_i = [v_i^L, v_i^U]$.

2. Set $k = n + 1$.

3. Compute elements of $v_k^\circ = \{v_k^{cv}, v_k^{cc}, v_k^L, v_k^U\}$ according to the definition of v_k:

   (a) If $v_k = v_i + v_j$, with $i, j < k$:

$$\begin{aligned}
\bar{v}_k^{cv} &= v_i^{cv} + v_i^{cv} \\
\bar{v}_k^{cc} &= v_i^{cc} + v_i^{cc} \\
v_k^L &= v_i^L + v_i^L \\
v_k^U &= v_i^U + v_i^U
\end{aligned}$$

   (b) If $v_k = v_i v_j$, with $i, j < k$:

$$\begin{aligned}
\bar{v}_k^{cv} &= \max(\alpha_i + \alpha_j - v_j^L v_i^L,\ \beta_i + \beta_j - v_j^U v_i^U) \\
\bar{v}_k^{cc} &= \min(\gamma_i + \gamma_j - v_j^L v_i^U,\ \delta_i + \delta_j - v_j^U v_i^L) \\
v_k^L &= \min(v_i^L v_j^L,\ v_i^L v_j^U,\ v_i^U v_j^L,\ v_i^U v_j^U) \\
v_k^U &= \max(v_i^L v_j^L,\ v_i^L v_j^U,\ v_i^U v_j^L,\ v_i^U v_j^U),
\end{aligned}$$

   where:

$$\begin{aligned}
\alpha_i &= \min(v_j^L v_i^{cv},\ v_j^L v_i^{cc}), \quad & \alpha_j &= \min(v_i^L v_j^{cv},\ v_i^L v_j^{cc}) \\
\beta_i &= \min(v_j^U v_i^{cv},\ v_j^U v_i^{cc}), \quad & \beta_j &= \min(v_i^U v_j^{cv},\ v_i^U v_j^{cc}) \\
\gamma_i &= \max(v_j^L v_i^{cv},\ v_j^L v_i^{cc}), \quad & \gamma_j &= \max(v_i^U v_j^{cv},\ v_i^U v_j^{cc}) \\
\delta_i &= \max(v_j^U v_i^{cv},\ v_j^U v_i^{cc}), \quad & \delta_j &= \max(v_i^L v_j^{cv},\ v_i^L v_j^{cc})
\end{aligned}$$

(c) Otherwise, with v_k defined by a univariate intrinsic function u_k:

$$\begin{aligned}\bar{v}_k^{cv} &= e_{u_k}(h_k^{\min})\\ \bar{v}_k^{cc} &= E_{u_k}(h_k^{\max})\\ v_k^L &= u_k^L(v_i^L, v_i^U)\\ v_k^U &= u_k^U(v_i^L, v_i^U),\end{aligned}$$

where $[u_k^L, u_k^U]$ represents the domain of the univariate intrinsic function u_k, $e_{u_k}, E_{u_k} : V_i \to \mathbb{R}$ are, respectively, convex and concave relaxations of u_k on V_i, $i < k$, and

$$h_k^{\min} = \text{mid}(v_i^{cv}, v_i^{cc}, z_k^{\min}) \quad \text{and} \quad h_k^{\max} = \text{mid}(v_i^{cv}, v_i^{cc}, z_k^{\max}),$$

where $z_k^{\min}$ is a minimum of e_{u_k} on V_i and $z_k^{\max}$ is a maximum of E_{u_k} on V_i.

4. Compute v_k^{cv} and v_k^{cc} as $v_k^{cv} = \text{mid}(v_k^L, v_k^U, \bar{v}_k^{cv})$ and $v_k^{cc} = \text{mid}(v_k^L, v_k^U, \bar{v}_k^{cc})$.

5. If $k = m$, go to Step 6. Otherwise, assign $k := k + 1$ and go to Step 3.

6. Set $\mathscr{U}(\mathbf{y}^\circ) = v_m^{cv}(\mathbf{y}^\circ)$ and $\mathscr{O}(\mathbf{y}^\circ) = v_m^{cc}(\mathbf{y}^\circ)$.

Given a function $\mathscr{F} : Z \subset \mathbb{R}^{n_p} \to \mathbb{R}$, one could obtain McCormick relaxations on a set $P = [\mathbf{p}^L, \mathbf{p}^U] \subset Z$ using Definition 2.6.14 by letting $\mathscr{V}$ be the set of factors describing $\mathscr{F}$, letting $n = n_p$ and $\mathbf{\Phi} = P$, and for every $\mathbf{p} \in P = \mathbf{\Phi}$, considering only those $\mathbf{y}^\circ \in \widetilde{\mathbf{\Phi}}$ such that $y_i^\circ = (p_i, p_i, p_i^L, p_i^U)$, $i = 1, \ldots, n_p$.

The McCormick relaxations defined by Definition 2.6.14 are *generalized* in the sense that the initial McCormick objects v_i° for $i = 1, \ldots, n_f$ are set equal to arguments y_i° that are separate from the value of $\mathbf{p}$ [121]. Further, the use of the sets defined by Definition 2.6.13 are necessary for the generalization of McCormick relaxations, as they allow for the formulation of relaxations with respect to a generic collection of factors with a corresponding domain [121]. Effectively, this allows Definition 2.6.14 to be used with previously constructed McCormick objects and thereby allows one to evaluate intermediate McCormick objects v_k° before composing them with later factors [121]. For simplicity, throughout this work we will refer to generalized McCormick relaxations as *McCormick relaxations*. The concept of calculating and utilizing intermediate results is directly used in Chapters 3 and 4, and additional details are presented by Scott et al. [121] on the distinction between *generalized* McCormick relaxations and the special case of *standard* McCormick relaxations. We also note that, as presented, the McCormick relaxation calculation procedure represents a "forward-mode"

calculation in the context of automatic differentiation. A "reverse-mode" calculation procedure was developed by Wechsung et al. [151], which could improve the tightness of these relaxations. However, a forward-mode calculation is necessary to first calculate valid set-valued enclosures on the range of the function of interest. Additionally, Beckers et al. [13] developed a procedure for calculating subgradients using an adjoint method, following the construction of McCormick relaxations using a forward-mode approach. This work will only focus on the forward-mode approach for relaxation and subgradient calculations, and reverse-mode approaches to tightening relaxations and calculating subgradients are left as future research directions.

When introducing the concept of McCormick relaxations, it is important to distinguish between the methods employed in this work and the well-studied *auxiliary variable method* (AVM) [104] and αBB approaches [5, 6, 9, 64, 73]. AVM, utilized by solvers such as BARON [119, 161] and ANTIGONE [98], is conceptually similar to the McCormick relaxation method, except that rather than calculate $\mathscr{U}$ and $\mathscr{O}$, the factors v_k are kept as additional problem variables. The AVM approach allows for smooth and potentially tighter relaxations than what can be obtained using McCormick relaxations [137], at the expense of a higher problem dimensionality. Due to the lower dimensionality of the McCormick relaxation method as compared to AVM, the McCormick-based approach is sometimes referred to as a *reduced-space method* [21]. A more thorough comparison between the McCormick and AVM approaches is provided by Tsoukalas and Mitsos [141]. Another way of relaxing mathematical expressions is to use the αBB methodology, which generates underestimators of general nonconvex terms by superimposing them with sums of univariate quadratics of sufficient magnitudes given by parameters α_i [73]. Both McCormick relaxations and the αBB methodology possess quadratic convergence rates [20, 73], which as stated previously is a minimum requirement to avoid the clustering problem in B&B [150]. αBB-style underestimators are employed for some expression types by, for example, the DGO solver ANTIGONE [98]. While this work focuses on McCormick-based relaxations, the αBB methodology may be amenable to a similar style of parallelization used in Chapter 4.

In addition to the McCormick relaxation rules given in Definition 2.6.14, there are corresponding rules that may be used to calculate relaxation subgradients as follows. Analogous to how Definition 2.6.8 enabled the calculation of relaxations of factors v_k, which enabled the procedure in Definition 2.6.14, Definition 2.6.10 enables the calculation of relaxation subgradients of factors v_k. Therefore, Definitions 2.6.15, 2.6.16, and 2.6.17 may be employed in parallel with the procedure in Definition 2.6.14 to calculate not only relaxations of arbitrarily complex factorable functions, but also their subgradients. Additional information about subgradient rules and calculations can be found in

[99, 133].

Definition 2.6.15 (Addition Rule for Subgradients [99])**.** Suppose that Step 3a from Definition 2.6.14 has been applied to obtain relaxations of a factor $v_k = v_i + v_j$, $i, j < k$. Let $\boldsymbol{\sigma}_{v_i}^{cv}$, $\boldsymbol{\sigma}_{v_i}^{cc}$, $\boldsymbol{\sigma}_{v_j}^{cv}$, and $\boldsymbol{\sigma}_{v_j}^{cc}$ be the subgradients of v_i^{cv}, v_i^{cc}, v_j^{cv}, and v_j^{cc}, respectively. Then, $\boldsymbol{\sigma}_{v_i}^{cv} + \boldsymbol{\sigma}_{v_j}^{cv}$ is a subgradient of v_k^{cv} and $\boldsymbol{\sigma}_{v_i}^{cc} + \boldsymbol{\sigma}_{v_j}^{cc}$ is a subgradient of v_k^{cc}.

Definition 2.6.16 (Multiplication Rule for Subgradients [99])**.** Suppose that Step 3b from Definition 2.6.14 has been applied to obtain relaxations of a factor $v_k = v_i v_j$, $i, j < k$. Subgradients of $\alpha_i, \alpha_j, \beta_i, \beta_j, \gamma_i, \gamma_j, \delta_i, \delta_j$ at $\mathbf{y}^\circ$ are given by:

$$\boldsymbol{\sigma}_{\alpha_i} = \begin{cases} v_j^L \boldsymbol{\sigma}_{v_i}^{cv} & \text{if } v_j^L \geq 0, \\ v_j^L \boldsymbol{\sigma}_{v_i}^{cc} & \text{otherwise,} \end{cases} \qquad \boldsymbol{\sigma}_{\alpha_j} = \begin{cases} v_i^L \boldsymbol{\sigma}_{v_j}^{cv} & \text{if } v_i^L \geq 0, \\ v_i^L \boldsymbol{\sigma}_{v_j}^{cc} & \text{otherwise,} \end{cases}$$

$$\boldsymbol{\sigma}_{\beta_i} = \begin{cases} v_j^U \boldsymbol{\sigma}_{v_i}^{cv} & \text{if } v_j^U \geq 0, \\ v_j^U \boldsymbol{\sigma}_{v_i}^{cc} & \text{otherwise,} \end{cases} \qquad \boldsymbol{\sigma}_{\beta_j} = \begin{cases} v_i^U \boldsymbol{\sigma}_{v_j}^{cv} & \text{if } v_i^U \geq 0, \\ v_i^U \boldsymbol{\sigma}_{v_j}^{cc} & \text{otherwise,} \end{cases}$$

$$\boldsymbol{\sigma}_{\gamma_i} = \begin{cases} v_j^L \boldsymbol{\sigma}_{v_i}^{cc} & \text{if } v_j^L \geq 0, \\ v_j^L \boldsymbol{\sigma}_{v_i}^{cv} & \text{otherwise,} \end{cases} \qquad \boldsymbol{\sigma}_{\gamma_j} = \begin{cases} v_i^U \boldsymbol{\sigma}_{v_j}^{cc} & \text{if } v_i^U \geq 0, \\ v_i^U \boldsymbol{\sigma}_{v_j}^{cv} & \text{otherwise,} \end{cases}$$

$$\boldsymbol{\sigma}_{\delta_i} = \begin{cases} v_j^U \boldsymbol{\sigma}_{v_i}^{cc} & \text{if } v_j^U \geq 0, \\ v_j^U \boldsymbol{\sigma}_{v_i}^{cv} & \text{otherwise,} \end{cases} \qquad \boldsymbol{\sigma}_{\delta_j} = \begin{cases} v_i^L \boldsymbol{\sigma}_{v_j}^{cc} & \text{if } v_i^L \geq 0, \\ v_i^L \boldsymbol{\sigma}_{v_j}^{cv} & \text{otherwise.} \end{cases}$$

And subgradients of v_k are given by:

$$\boldsymbol{\sigma}_{v_k}^{cv} = \begin{cases} \boldsymbol{\sigma}_{\alpha_i} + \boldsymbol{\sigma}_{\alpha_j} & \text{if } \alpha_i + \alpha_j - v_i^L v_j^L \geq \beta_i + \beta_j - v_i^U v_j^U, \\ \boldsymbol{\sigma}_{\beta_i} + \boldsymbol{\sigma}_{\beta_j} & \text{if } \alpha_i + \alpha_j - v_i^L v_j^L \leq \beta_i + \beta_j - v_i^U v_j^U \end{cases}$$

$$\boldsymbol{\sigma}_{v_k}^{cc} = \begin{cases} \boldsymbol{\sigma}_{\gamma_i} + \boldsymbol{\sigma}_{\gamma_j} & \text{if } \gamma_i + \gamma_j - v_i^U v_j^L \leq \delta_i + \delta_j - v_i^L v_j^U, \\ \boldsymbol{\sigma}_{\delta_i} + \boldsymbol{\sigma}_{\delta_j} & \text{if } \gamma_i + \gamma_j - v_i^U v_j^L \geq \delta_i + \delta_j - v_i^L v_j^U. \end{cases}$$

Definition 2.6.17 (Subgradients from McCormick's Composition [99])**.** Suppose that Step 3c of Definition 2.6.14 has been applied to obtain the relaxations of a factor $v_k = u_k(v_i)$, $i < k$. Further, suppose that $\sigma_e(v_i^{cv})$ is a subgradient of e_{u_k} at v_i^{cv}, $\sigma_e(v_i^{cc})$ is a subgradient of e_{u_k} at v_i^{cc}, $\sigma_E(v_i^{cv})$ is a subgradient of E_{u_k} at v_i^{cv}, and $\sigma_E(v_i^{cc})$ is a subgradient of E_{u_k} at v_i^{cc}, and let $\boldsymbol{\sigma}_{v_i}^{cv}$, $\boldsymbol{\sigma}_{v_i}^{cc}$, $\boldsymbol{\sigma}_{v_j}^{cv}$, and $\boldsymbol{\sigma}_{v_j}^{cc}$ be the subgradients of v_i^{cv}, v_i^{cc}, v_j^{cv}, and v_j^{cc}, respectively. Then, a subgradient of v_k^{cv} is given

by:

1. $\mathbf{0}$ if $v_i^{cv} \leq z_k^{\min} \leq v_i^{cc}$,
2. $\sigma_e(v_i^{cc})\boldsymbol{\sigma}_{v_i}^{cc}$ if $v_i^{cc} < z_k^{\min}$,
3. $\sigma_e(v_i^{cv})\boldsymbol{\sigma}_{v_i}^{cv}$ if $z_k^{\min} < v_i^{cv}$,

and a subgradient of v_k^{cc} is given by:

1. $\mathbf{0}$ if $v_i^{cv} \leq z_k^{\max} \leq v_i^{cc}$,
2. $\sigma_E(v_i^{cc})\boldsymbol{\sigma}_{v_i}^{cc}$ if $v_i^{cc} < z_k^{\max}$,
3. $\sigma_E(v_i^{cv})\boldsymbol{\sigma}_{v_i}^{cv}$ if $z_k^{\max} < v_i^{cv}$.

2.6.3 Relaxed Linear Programs

In Section 2.6.2, an extensive discussion was provided on how McCormick relaxations and their subgradients may be calculated. Here, we will demonstrate how those relaxations and subgradients may be used to generate a valid lower bound for the B&B algorithm. Consider a nonconvex NLP of the form:

$$\begin{aligned} f^* = \min_{\mathbf{x}}\ & f(\mathbf{x}) \\ \text{s.t.}\ & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{x} \in X \subset \mathbb{R}^n. \end{aligned} \tag{2.8}$$

A standard approach to calculate a valid lower bound of (2.8) using relaxations and their subgradients involves solving an easier *relaxed* problem. In particular, the relaxed problem is typically a related LP or convex NLP whose solution is theoretically guaranteed to be a lower bound of (2.8). Due to the convexity of these relaxed problems, local optimization solvers can be used.

To create a relaxed LP of (2.8)—also referred to as an LP relaxation—it is necessary to convert all nonlinear expressions into affine underestimating expressions that are referred to as *affine relaxations*. An affine relaxation may be generated by, for example, linearizing a McCormick relaxation of an expression of interest using its subgradient. The resulting LP for a given subdomain $X' \subset X$ takes

the form:

$$\begin{aligned} f^*_{\text{aff}} = \min_{\mathbf{x}} \ & \zeta \\ \text{s.t.} \quad & f_{\text{aff}}(\mathbf{x}) \leq \zeta \\ & \mathbf{g}_{\text{aff}}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{x} \in X' \subset X \subset \mathbb{R}^n, \end{aligned} \tag{2.9}$$

where $\zeta \in \mathbb{R}$ is the epigraph variable and f_{aff} and $\mathbf{g}_{\text{aff}}$ are affine relaxations of f and $\mathbf{g}$, respectively. Typically, the linearization of McCormick relaxations occurs at one or more linearization points. The choice to use additional linearization points may be beneficial, as the resulting lower bounds may be tighter than if fewer linearization points are used, though this adds computational expense in terms of calculating more relaxations and solving potentially multiple LPs. The method of choosing linearization points is largely based on heuristics, and is discussed in greater detail in Chapter 6. Program (2.9) forms a polyhedral outer approximation of the feasible set of (2.8), which guarantees that $f^*_{\text{aff}} \leq f^*$. This process can be visualized in Figure 2.3. Through this process, the task of obtaining a lower bound for (2.8) on a subdomain X' is reduced to the creation and solution of the LP given by (2.9). This approach is favored over directly using nonlinear relaxations in an NLP formulation due to the numerical reliability and the low computational costs of solving LPs [104] and is utilized by many established global solvers such as BARON [76, 119, 161], ANTIGONE [98], MAiNGO [22], SCIP [19, 145], and EAGO [155].

2.6.4 Parallelization Challenges

Now that the standard techniques used within global optimization have been described, they may be connected with topic of GPU parallelization that is central to this work. To recontextualize some points made previously, the spatial B&B algorithm is limited in practice to relatively small problems due to the vast computational expense it entails, particularly with regard to the lower-bounding problem. On the computing side, GPUs have become the predominant and mainstream method of carrying out calculation- and data-intensive tasks, being more cost- and energy-efficient than CPUs for large workloads. These points suggest that GPUs may be both useful and beneficial for accelerating the speed and efficiency of DGO methods. Despite the potential benefits, however, most mature implementations of DGO solvers continue to rely exclusively on CPU hardware. While DGO implementations have made use of different forms of CPU parallelization [68, 72, 124, 125],

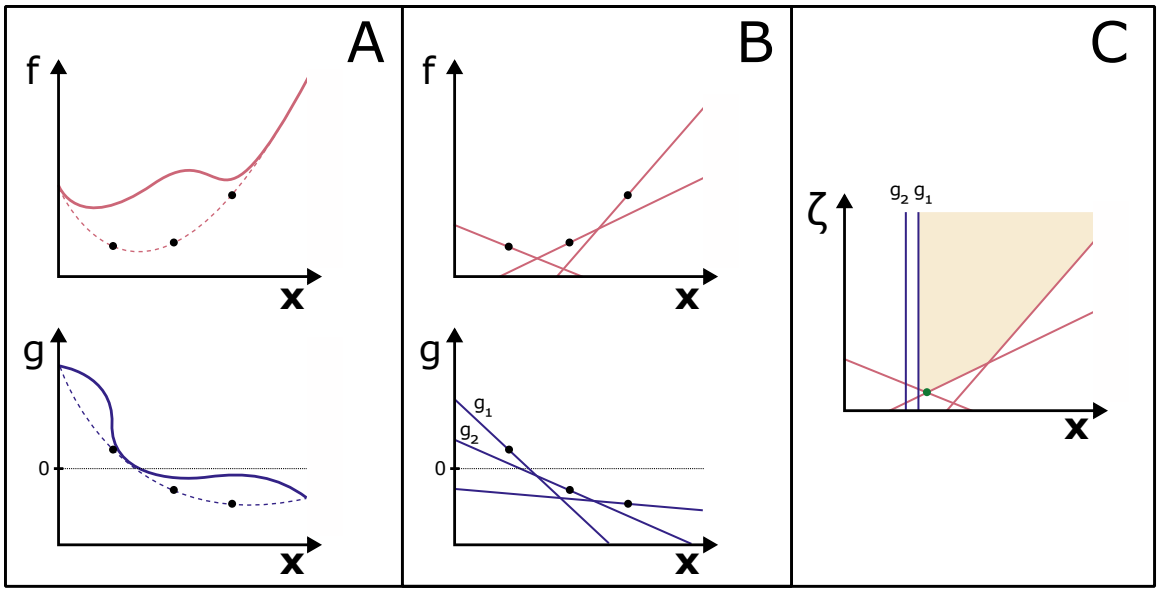

Figure 2.3: A standard approach to obtain a lower bound of a nonconvex objective function f with a nonconvex inequality constraint $g \leq 0$. (A) Given f (red, solid) and g (blue, solid), and a method of calculating convex relaxations (dashed), several linearization points are selected (black). (B) Evaluating the convex relaxations and their subgradients at the linearization points yields sets of affine relaxations for f (red) and g (blue). (C) Affine relaxations are composed together as constraints in a single linear program (LP) (red and blue) minimizing an epigraph variable ζ, with constraints g_1 and g_2 (blue) appearing at the zero level set of the affine relaxations of g_1 and g_2 in panel B. The feasible region of the LP is shaded in orange, and the value of ζ at the solution of the LP (green dot) is a valid lower bound of f constrained by $g \leq 0$. Figure adapted from [51]

to the best of my knowledge only a small number of GPU-based implementations exist, comprising the present work and a recent development by Zhang et al. [160].

Accelerating DGO using GPUs is challenging for a number of reasons, mainly stemming from the SIMD/SIMT architecture that requires the same instruction to be applied to multiple portions of data. This design makes GPUs most effective when handling extremely regular workloads. One of the first roadblocks with DGO is that the B&B tree search is problem-dependent and dynamic [29, 147]. I.e., the structure of the tree cannot be known a priori. This makes workload scheduling a challenge, and it also potentially leads to problems with load balancing, where different GPUs, or different portions of individual GPUs, do not get utilized to their full potential because the effort required is unpredictable and highly variable. Depending on the specific GPU implementation, this irregularity may also lead to warp divergence that would severely degrade performance. Within the present work, the dynamic nature of the tree search does not have a substantial impact as the number of nodes available within the B&B stack typically far exceeds the number of tasks being run in parallel, and the entire stack is stored in the memory of one CPU. This issue becomes more of a

challenge when dealing with distributed systems, as it can be expensive to transfer memory between disparate computing elements. E.g., if two separate stacks are stored and operated on independently, the unpredictable nature of the B&B tree may cause one stack to deplete considerably faster than the other, thereby leaving some computing resources idle unless the remaining stack can be split and transferred. While this challenge is not a main focus of this work, some further discussion is included in Chapters 3–6.

A second essential challenge for GPU parallelization of DGO is that although the standard lower-bounding approach—comprising relaxations and creating and solving an LP—requires substantial background information to define, it is computationally cheap and largely serial. The calculation of McCormick relaxations for a given constraint, for example, requires a serial calculation following the cumulative mapping that defines the expression. For particularly complex expressions it may be possible to parallelize the calculation of their relaxations—such as, e.g., a parallel reduction approach to evaluate the product of many terms—but in most instances this is unlikely to provide much benefit as the calculation may only involve several floating-point calculations. Additionally, McCormick relaxations have the challenges of conditional checks and possibly calculation loops [156], each of which would directly lead to warp divergence and consequent poor performance [86]. The low calculation volume, serial nature, and presence of conditional checks and calculation loops suggest that GPU parallelization of relaxations would be ineffective in practice. In this work, however, a different approach is taken wherein many relaxations are calculated in parallel, rather than parallelizing the calculation of a single relaxation. This alternate approach for relaxation calculations is described in Chapters 3 and 4.

Similar to the challenges associated with relaxations, it is challenging to address the use of GPUs to solve relaxed LPs. Specifically, GPU implementations of LP solvers exist, but as noted in Section 2.4, they typically only become performant once the problem size exceeds roughly one thousand variables. This is in stark contrast to the typical problem sizes of global optimization problems, which in most cases are unlikely to exceed one hundred variables. Because the relaxed LPs are derived from the nonconvex NLPs as described in Section 2.6.3, the sizes of the LPs are generally comparable to the sizes of the NLPs. As a result, most GPU-based LP solvers would likely be less effective on relaxed LPs than traditional serial CPU-based LP solvers. In this work, a less conventional style of a GPU-based LP solver is considered which, much like the approach taken for relaxations, proceeds by addressing multiple LPs in parallel rather than individual LPs. This approach is described in greater detail in Chapters 5 and 6.

In summary, not counting the technical difficulties associated with their individual develop-

ment, GPU-accelerated components of the B&B lower-bounding problem are not expected to be performant due to their associated challenges. This work aims to show that despite the apparent roadblocks to efficient parallelization, the main components of the lower-bounding problem can be made individually efficient (Chapters 3–5) and ultimately fast and effective in combination (Chapter 6).

Chapter 3

Automatic Source Code Generation for Deterministic Global Optimization with Parallel Architectures

Trends over the past two decades indicate that much of the performance gains of commercial optimization solvers is due to improvements in x86 hardware. To continue making progress, it is critical to consider alternative/specialized massively parallel computing architectures. In this chapter, we detail the development of an open-source source code transformation approach built using `Symbolics.jl` to construct McCormick-based relaxations of functions that enables their effective parallelized evaluation. We then apply this approach in a novel parallelized branch-and-bound routine that offloads lower- and upper-bounding problems to a GPU. The effectiveness of this new approach is demonstrated on three nonconvex problems of interest, where it yields convergence time improvements of 22–118x compared to an equivalent serial CPU implementation and in two cases outperforms vanilla branch-and-bound versions of existing state-of-the-art solvers that use tighter bounding techniques. The work in this chapter exemplifies how deterministic global optimizers using alternative hardware architectures can compete with—or eventually outclass—even the most powerful serial CPU implementations, and to the best of the authors' knowledge, represents the first successful demonstration of deterministic global optimization using a GPU. This chapter is

published separately as reference [58].

3.1 Introduction and Motivation

Optimization research over the past several decades has led to an increase in the number and types of solvable problems, but as recently observed by Koch et al. [80], much of the perceived performance improvements of mixed-integer programming solvers were the result of advances in CPU hardware [80]. However, with the death of Moore's Law [116, 152], CPU hardware is not improving at the same high rate as in past decades. This performance slump is especially hard-hitting for deterministic global optimization solvers which are, to the best of our knowledge, exclusively designed to run on CPUs. Other application areas of process systems engineering that were traditionally designed for CPU operation, such as data analytics, simulation, and control, have found improved performance by utilizing alternative hardware architectures, including graphics processing units (GPUs) and application-specific integrated circuits (ASICs) [14, 15]. GPUs in particular are a common, cost-effective, and scalable way to gain massive speedups for specialized computations through parallel computing. The successes of parallelization in a variety of fields are well documented [47], yet to the best of our knowledge, there has been no publication documenting the use of GPUs for deterministic global optimization prior to the submission of this work for publication. With earlier preliminary results from this chapter first reported at the AIChE Annual Meeting 2022 [52] and FOCAPO-CPC 2023 [57], this chapter aims to detail the full development of the method, software implementation, and benchmarking for the first successful implementation, to the best of our knowledge, of a deterministic global optimization algorithm designed to function on a GPU for massive parallelization.

Modern deterministic global optimization solvers (BARON [119], ANTIGONE [98], SCIP [19, 145], MAiNGO [22], EAGO [155], etc.) rely on some variation of spatial branch-and-bound (B&B), where, for each spatial subdomain, a mathematically rigorous lower bound of the objective function must be obtained. One way to obtain these lower bounds is to apply the rules laid out by McCormick [95]. These rules were later formalized for use in a *forward mode* to operate on generic algorithms by Mitsos et al. [99], generalized by Scott et al. [121], extended to implicit functions by Stuber et al. [133], developed for use in a *reverse mode* by Wechsung et al. [151], and modified to ensure differentiability by Khan et al. [77, 78], among other recent developments. Through the McCormick composition theorem provided in Section 3.2, the McCormick-based approaches enable convex and concave relaxations—and therefore lower and upper bounds, respectively—to be generated for ar-

bitrarily complicated expressions. As observed by Mitsos et al. [99], the automatic construction of convex/concave relaxations via these rules is similar to the concept of automatic differentiation (AD). Due to this similarity, the implementation of automatic McCormick relaxations can be accomplished using the same techniques as AD; namely: operator overloading (OO) and source code transformation (SCT) [60].

Briefly, OO for AD involves the definition of a variable type for floating-point numbers and their associated gradients. Operations such as those for elementary arithmetic can be defined for these variable types to compute the result of the operation applied to the floating-point values as well as their associated gradients [18]. SCT, on the other hand, involves the automated rewriting or generation of source code that computes the derivative of an expression as a subroutine call [18].

AD is critically important for a variety of numerical computing tasks, including machine learning [12], computational fluid dynamics [75], and quantum chemistry [18]. In machine learning, for example, backpropagation and gradient-based optimization are ubiquitous for model training, and consequently several AD software tools have been developed specifically for use in machine learning applications [12]. In this application space, OO-based AD is more common (e.g., autograd [93], PyTorch [113], among others), but recent SCT implementations exist as well (e.g., Tangent [142, 143], etc.). As it pertains to the present work, there have been several successes in parallelizing these methods on GPU hardware that have accelerated the speed of machine learning model training and development [34, 81, 103, 139], although those developments did not directly guide this work. Generally, OO approaches are simpler to implement and can be flexibly used, but incur a runtime overhead time cost, whereas SCT approaches have much higher implementation complexity but can facilitate compiler optimizations that result in faster execution of the generated code [18].

For deterministic global optimization, all existing implementations of the rules of McCormick that we are aware of (e.g., `MC++` [28], `McCormick.jl` [153]) rely on OO, or the analogous multiple dispatch in the Julia language. In `McCormick.jl`, for example, the `MC` structure is defined, and new methods are created for the arithmetic operators and a library of univariate intrinsic functions that apply the McCormick calculation rules when they are dispatched on `MC` objects. Such an approach is adequate for cases where the relaxation calculations do not account for a high proportion of runtime requirements [99], but if relaxations are to be evaluated thousands or millions of times, as may happen in comparatively large global optimization problems, the combined overhead times may be substantial. Additionally, approaches such as `McCormick.jl` are incompatible with general-purpose computing on GPUs (GPGPU). Any implementation of GPGPU of a deterministic global optimizer that utilizes McCormick-based relaxations would therefore need an alternative method of evaluating

those relaxations.

In this chapter, we employ the SCT approach to interpret mathematical expressions and automatically generate static functions that represent convex underestimators, concave overestimators, and interval extensions of the original expressions. By design, these generated functions are compatible with GPU operation, enabling the parallelized evaluation of convex relaxations at multiple points. Additionally, a custom B&B routine has been developed to exploit this parallelism to solve lower-bounding problems of the B&B framework entirely on a GPU.

The remainder of this chapter is structured as follows. Section 3.2 provides a small amount of background information not previously covered in Chapter 2. In Section 3.3, we describe the implementation of the SCT approach capable of generating convex evaluator functions and describe how this approach can be used within deterministic global optimization contexts. Subsequently, in Section 3.4, we present the numerical results arising from a benchmark comparison between the SCT-based product of this work and the multiple dispatch approach of `McCormick.jl`, and then provide several numerical examples of optimization problems solved using this approach. Lastly, in Section 3.6, we reflect on the technical challenges associated with this new method and software implementation, and we present our plans for future improvements.

3.2 Background

Much of the relevant background information for this chapter is provided in Chapter 2, specifically in Sections 2.6.1–2.6.2. Particularly, we emphasize that one common way to obtain LBD_i for $\min \hat{f}(\mathcal{F} \cap M^{(i)})$ or $\min \hat{f}(M^{(i)})$ in Algorithm 1 is by minimizing a suitable convex relaxation of f on $M^{(i)}$, as described in Definition 2.6.3. Existing implementations of B&B operate by selecting an individual element of the partition $M^{(i)}$, creating and solving a convex subproblem to obtain LBD_i, and further partitioning element $M^{(i)}$ before repeating the process. As indicated in the process laid out by Horst and Tuy [65] in Section 2.6.1, in principle, multiple elements of the partition $M^{(i)}$ may be selected in each iteration. However, in practice, multiple elements are not addressed simultaneously, and Steps 2 through 4 are iterated on for each i. One of the novel contributions of this chapter is a B&B algorithm capable of handling multiple partition elements simultaneously by exploiting computing hardware designed to accelerate such parallelized processes.

3.3 Software Development

One key product of this work is a GPU-compatible SCT approach to construct McCormick-based relaxations. These relaxations are then used in a novel B&B routine to *process* (i.e., solve lower- and upper-bounding problems) multiple partition elements (nodes) in parallel, which is described at the end of this section. Previous methods of creating relaxations by applying McCormick arithmetic include `MC++` [28], written in C++ and used in the MAiNGO solver [22], and `McCormick.jl` [153], written in Julia and used in the EAGO solver [155]. Both of these approaches comprise intrinsic function libraries and an OO-like scheme to construct relaxations in a functionally and mathematically equivalent way, just in different programming languages.

Similar to the ways that forward-mode AD is accomplished, it is also possible to implement the rules of McCormick via SCT. A first implementation of this new approach is currently, at the time of writing, under active development within the `SourceCodeMcCormick.jl` (`SCMC`) package, which is available publicly on GitHub [53]. As a brief summary of its operation: `SCMC` uses the Julia package `Symbolics.jl` [59] to decompose mathematical expressions, applies the generalized McCormick relaxation rules [121, 153] and interval bounding rules based on interval arithmetic and natural interval extensions [102], creates source code that computes McCormick relaxations and inclusion monotonic interval extensions associated with the original expression, and then compiles the source code into functions that return inclusion monotonic interval extensions and pointwise values of convex and concave relaxations of the original input expression. The intended use for these evaluator functions is to evaluate relaxations at a large number of points on potentially different domains within a single function call, and thereby exploit the parallel computing capabilities of GPUs.

`SCMC` relies on several features of the `Symbolics.jl` package to operate. Primarily, `Symbolics.jl` is used as a symbolic algebra system that enables McCormick relaxations to be expressed symbolically. These relaxations are then composed together to create expressions representing composite relaxations using the `Symbolics.substitute` function, and the final expressions are converted into callable functions using the `Symbolics.build_function` routine. Further details of how these utilities are employed by `SCMC` are presented in the remainder of this section.

A truncated version of `convex_evaluator`, one of the main user-facing utilities in `SCMC`, is presented in Listing 3.1. This utility creates an evaluator function for the convex relaxation of a math expression. The following subsections will detail the mechanism of this function to describe how an input expression is converted to an evaluator function.

Listing 3.1: A truncated version of the `SCMC` `convex_evaluator` function is presented. `convex_evaluator` accepts a mathematical expression composed of `Symbolics.jl`-type variables and returns a function that evaluates a convex relaxation of the input mathematical expression, as well as an ordered list of variables to inform the user as to the input format of the generated function.

```
1  function convex_evaluator(term::Symbolics.Num)
2     step_1 = apply_transform(McCormickIntervalTransform(), term)
3     step_2 = shrink_eqs(step_1)
4     ordered_vars = pull_vars(step_2)
5     @eval new_func = $(build_function(step_2[3].rhs, ordered_vars...))
6     return new_func, ordered_vars
7  end
```

3.3.1 Factorization and Computational Graph Generation

The `SCMC` package exploits the factorability of functions of interest to construct McCormick-based relaxations. As such, we must make the practical and generally non-restricting assumption that all functions of interest are factorable. When the user inputs a mathematical expression into an evaluator generation function such as `convex_evaluator` shown in Listing 3.1, the expression is automatically factored into binary and univariate terms to generate a primal trace of the expression, from which a computational graph can be inferred. The generation of a primal trace can be seen for a representative expression in Listing 3.2, with the resulting primal trace and its corresponding computational graph shown in Figure 3.1 (left and center, respectively). Elements of the primal trace of the expression are represented in the graph as nodes, which are joined by labeled arcs that represent binary addition, multiplication, or division (i.e., inversion of the denominator multiplied by the numerator [99, 121, 153, 158]), or are individually operated on by a univariate intrinsic function. This is analogous to the auxiliary variable method of generating relaxations in that each node can be thought of as an auxiliary variable representing an intermediate term in the calculation of the original expression. However, unlike in the auxiliary variable method, when McCormick-based relaxations of the original expression are ultimately generated, these auxiliary variables are eliminated as intermediates and do not appear in the final calculation. Internally, `SCMC` represents the trace as a vector with elements of type `Symbolics.Equation`, where the left-hand sides are the auxiliary variables and the right-hand sides are the corresponding factored expressions. The left- and right-hand sides of these `Equation`s are maintained as type `SymbolicUtils.BasicSymbolic{Real}`, where `SymbolicUtils.jl` is the foundation for `Symbolics.jl`.

For ease of organizing the necessary data and computed values, we define the following data structure, which is analogous to the tuples that are often defined in AD libraries (e.g., $(f(x), f'(x))$).

Definition 3.3.1 (McCormick Tuple)**.** Consider a cumulative mapping $v_k : Z \in \mathbb{IR}^n \to \mathbb{R}$. A

Listing 3.2: An example showing how SCMC can be used to create a primal trace of a mathematical expression. Note that this step is for demonstration and analysis purposes and is not necessary for standard use cases of SCMC.

```
using SourceCodeMcCormick, Symbolics
@variables x, y
expr = exp(x/y) - (x*y^2)/(y+1)
trace = factor(expr)
```

McCormick tuple will be the collection of four symbolic elements: `(vk_cv, vk_cc, vk_lo, vk_hi)` that represents $\{v_k^{cv}(\cdot), v_k^{cc}(\cdot), v_k^L(Z), v_k^U(Z)\}$. A McCormick tuple of a base variable `z` will be `(z_cv, z_cc, z_lo, z_hi)`.

Note that this definition of a McCormick tuple is very closely related to the septuple defined by Mitsos et al. [99] and the McCormick *object* defined by Ye and Scott [158] with a few exceptions. First, the septuple of Mitsos et al. [99] includes the cumulative mapping v_k and subgradient information. Subgradients are currently not included in the proposed approach, as discussed in Section 3.5. The McCormick objects of Ye and Scott [158] are functionally equivalent (despite differences in organization) to Definition 3.3.1 with the nuance that McCormick objects are defined with real-valued components (i.e., pointwise with respect to the independent variable for McCormick relaxations), whereas the McCormick tuple of Definition 3.3.1 contains the actual functions representing McCormick relaxations.

Once a function is factored (a computational graph is prepared), SCMC proceeds by creating a McCormick tuple of each base variable (e.g., `x`). If a base variable is being branched on in a B&B scheme, its interval bounds will be the lower and upper bounds of the B&B node for this variable, and its convex and concave relaxations will simply take the value `z` where the relaxation of the primal expression is desired (i.e., `x_cv = x_cc = z`) [121, Def. 9, Step 2]. The computational graph is then processed in a forward mode as follows. Starting from the children of the root nodes, each node is extended into its McCormick tuple by applying the McCormick rule(s) corresponding to the operation that fed into the node, with the McCormick tuple(s) of the parent node(s) as inputs. An example of the process of extending base variables and factors into their McCormick tuples is shown in Figure 3.1. While a basic computational graph that represents the primal trace of a function has $m - n$ nodes (not including the base variable nodes), where each has at most 2 inputs, the result of the application of McCormick rules is a larger computational graph with $4(m - n)$ nodes, each having at most 8 inputs. In this new graph, each node represents one part of its origin node's McCormick tuple.

The process described so far, including the factorization step and the application of McCormick

$$
\begin{aligned}
v_1 &= x \\
v_2 &= y \\
v_3 &= v_1 / v_2 \\
v_4 &= \exp(v_3) \\
v_5 &= v_2^2 \\
v_6 &= v_1 v_5 \\
v_7 &= -v_6 \\
v_8 &= v_2 + 1.0 \\
v_9 &= v_7 / v_8 \\
v_{10} &= v_4 + v_9
\end{aligned}
$$

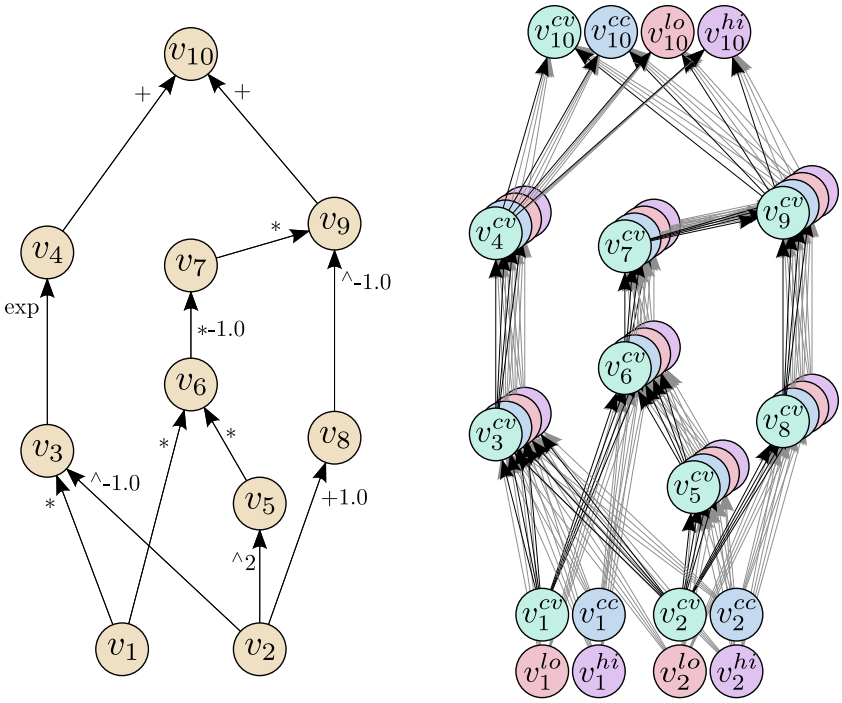

Figure 3.1: (left) The primal trace of the expression $\exp(x/y) - xy^2/(y+1)$, (center) the associated computational graph, and (right) the expanded computational graph. The nodes in the original computational graph are split into their McCormick tuples, creating the expanded computational graph. For nodes that were connected in the original graph, their McCormick tuples are connected in the expanded graph. Although the density of interrelationships between nodes in the expanded graph is high, the overall structure is comparable to the original computational graph.

rules, occurs in Line 2 of Listing 3.1. Specifically, the `apply_transform` function combines the factorization and McCormick rule application steps to output a vector with elements of type `Equation`. In this vector, the left-hand sides are variables of type `BasicSymbolic{Real}`, representing individual elements of the McCormick tuples of the auxiliary variables, and the right-hand sides are the symbolic representations of the appropriate McCormick rule(s), stored as type `BasicSymbolic{Real}`. A trivial example of this structure is displayed in Listing 3.3, which shows the vector of `Equations` that result from applying the `apply_transform` function to the expression $x + y$, where x and y are `Symbolics.jl` variables (wrapped in the `Symbolics.Num` type).

Following the application of McCormick rules, the computational graph is collapsed via edge contraction to eliminate auxiliary variables until only the four deepest nodes—the McCormick tuple of the original full expression—remain. This step occurs in Line 3 of Listing 3.1, using the utility `shrink_eqs`. The function works by selecting the first `Equation` in the input vector and replacing all instances of the `Equation`'s left-hand side with its right-hand side for all other `Equations` in the vector using the `SymbolicUtils.substitute` subroutine. That is, it replaces instances of ele-

Listing 3.3: Output from the `apply_transform` function applied to the expression $x + y$. The variables x and y are extended to their McCormick tuples, and elements of the McCormick tuple of the resulting expression are labeled as `result_*`. The tilde character (`~`) is used to represent equality in `Symbolics.Equation`s.

```
1 julia> apply_transform(McCormickIntervalTransform(), x+y)
2    4-element Vector{Equation}:
3       result_lo ~ x_lo + y_lo
4       result_hi ~ x_hi + y_hi
5       result_cv ~ x_cv + y_cv
6       result_cc ~ x_cc + y_cc
```

ments of McCormick tuples of auxiliary variables with the expressions representing their associated McCormick rule(s). This process is repeated iteratively to eliminate the McCormick tuples of all auxiliary variables. The end result is a vector with four components of type `Equation`, where the left-hand sides are `Symbolics.jl` variable representations of the McCormick tuple of the original expression, and the right-hand sides are the symbolic compositions of all the McCormick rules that made up the original expression. By eliminating the auxiliary variables and composing rules starting from the base variables, these four expressions are only functions of the McCormick tuples of the base variables.

One key property of McCormick relaxations, as illustrated by Definition 2.6.14, is that the mathematical structure of $f^{cv}(\mathbf{x})$ depends not only on the expression of f, but also on the bounds of the domain of $\mathbf{x}$ (i.e., $X = [\mathbf{x}^L, \mathbf{x}^U]$). As an aside, the development of the generalized McCormick relaxation [121] made this dependence explicit. `McCormick.jl` handles this type of bounds dependence through the use of control flow in its OO-like scheme. That is, different software functions will be called depending on logic statements based on the bounds of the input variable(s). In the novel source code generation approach of `SCMC`, this bounds dependence is accounted for using the Julia `ifelse` function. By deeply nesting these `ifelse` functions within one another, all possible mathematical structures of a relaxation that arise from composing McCormick rules together can be contained within a single code expression, where the output of the expression depends on the same type of logic statements used in typical OO schemes.

An example of how the `ifelse` function is used within `SCMC` can be seen in Listing 3.4. This listing shows a simplified version of the McCormick rule that `SCMC` applies for the multiplication of two terms, truncated to show only the convex relaxation part of the function. By using `ifelse` functions, this expression captures all possible realizations of the convex relaxation arising from a multiplication step within a single `Equation`. Because the input McCormick tuples are treated symbolically, during the edge contraction step, these symbolic variables can be replaced by full

Listing 3.4: A simplified and truncated version of the `transform_rule` for the multiplication of two symbolic variables, `x` and `y`. The `apply_transform` function automatically extends `x` and `y` into their McCormick tuples and then applies the appropriate `transform_rule` with those tuples as inputs, such as this version of the function for multiplication. This `transform_rule` function returns `Equation`s where the left-hand sides (first argument) are the symbolic variables corresponding to the convex and concave relaxations of the product, and the right-hand sides (second argument) are the symbolic expressions of the applied McCormick rule.

```
function transform_rule(::McCormickTransform, ::typeof(*), zL, zU, zcv, zcc,
                                        xL, xU, xcv, xcc, yL, yU, ycv, ycc)
  rcv = Equation(zcv,
      ifelse(xL >= 0.0,
          ifelse(yL >= 0.0, max(-xU*yU + xU*ycv + xcv*yU,
                                -xL*yL + xL*ycv + xcv*yL),
              ifelse(yU <= 0.0, -min(xU*yU - xU*ycv - xcc*yU,
                                     xL*yL - xL*ycv - xcc*yL),
                  max(-xU*yU + xU*ycv + xcv*yU,
                      -xL*yL + xL*ycv + xcc*yL))),
          ifelse(xU <= 0.0,
              ifelse(yL >= 0.0, -min(xL*yL - xL*ycc - xcv*yL,
                                     xU*yU - xU*ycc - xcv*yU),
                  ifelse(yU <= 0.0, max(-xL*yL + xL*ycc + xcc*yL,
                                        -xU*yU + xU*ycc + xcc*yU),
                      -min(xL*yL - xL*ycc - xcc*yL,
                           xU*yU - xU*ycc - xcv*yU))),
              ifelse(yL >= 0.0, max(-xU*yU + xU*ycv + xcv*yU,
                                    -xL*yL + xL*ycc + xcv*yL),
                  ifelse(yU <= 0.0, -min(xL*yL - xL*ycc - xcc*yL,
                                         xU*yU - xU*ycv - xcc*yU),
                      max(-xU*yU + xU*ycv + xcv*yU,
                          -xL*yL + xL*ycc + xcc*yL)))))
  [...]
```

symbolic expressions of the McCormick tuples of the inputs thereby resulting in a more deeply nested `ifelse` expression that encodes the combined rules of multiple mathematical operations.

A key benefit of this approach is that, because `ifelse` is a function, `SCMC` can continue to work in the environment of the `Symbolics.jl` symbolic algebra system, which is important for the source code generation process, while still utilizing the control decisions necessary for building McCormick relaxations. As discussed in Section 3.3.4, in practice, the complexity of representable factorable functions that are represented in this manner is limited due to the rapidly increasing length of such expressions. That is, the output from the `shrink_eqs` function contains four right-hand-side terms that must capture all possible dependencies of the final McCormick tuple on the dependent variables' domains.

3.3.2 Source Code Generation

By defining and applying the McCormick rules as single-line code expressions, the deepest nodes of the expanded (and then edge-contracted) computational graph are already in a form that, if written out, could be interpreted by Julia as source code expressions that are the McCormick tuples

of the original function. For example, if the right-hand sides of the four `Equation`s output by `shrink_eqs` were pasted into a separate Julia file, with McCormick tuples of the base variables assigned numerical values, the pasted expressions would return evaluations of the McCormick tuple of the original mathematical expression. However, by remaining within the `Symbolics.jl` algebra system throughout the application of McCormick-based relaxation rules, `SCMC` can further automate the step of interpreting the source code expressions and substituting in numerical values through the use of `Symbolics.build_function`.

When a symbolic expression is passed to `Symbolics.build_function`, it generates code for a numeric function that allows the substitution of numerical input values in place of the variables of type `BasicSymbolic{Real}`. `build_function` is used in Line 5 of Listing 3.1, where the third element of the vector output from `shrink_eqs` (i.e., the term representing the convex relaxation of the original expression) is passed as an argument, along with a sorted list of `BasicSymbolic{Real}`-typed variables used in the relaxation, generated in Line 4 using the `SourceCodeMcCormick.pull_vars` function. The code output from `build_function` is then wrapped in an `@eval` statement to compile the code into a callable Julia function. Although not utilized by `SCMC`, `build_function` is also capable of building functions compatible with other languages such as C, Stan, and MATLAB [59]. This function, and the ordered list of `BasicSymbolic{Real}`-typed variables to use as a reference, are the outputs of the `convex_evaluator` function in Listing 3.1. An additional user-facing function, `all_evaluators`, returns four functions representing the elements of the McCormick tuple as well as the ordered variable list. In sum, because the McCormick rules were applied within the `Symbolics.jl` algebra system, we are allowed to use `build_function` to construct static evaluation functions that return inclusion monotonic interval extensions and convex and concave relaxation values of the original function at respective input values.

One important distinction between using `build_function` and simply pasting the top-level node expressions into a separate file is the fate of the `ifelse` function. Although this function is useful for remaining in the `Symbolics.jl` algebra system, if used as-is, the resulting code will perform very poorly. The `ifelse` function works by evaluating both conditional paths and then returning the appropriate result based on the conditional statement. This operation would waste significant computing time, particularly because the nature of McCormick-based relaxations means that there would be many potential conditional branches. The `build_function` utility, however, internally substitutes `ifelse` with normal if-else control flow, which only evaluates the branch corresponding to the conditionally selected outcome. This type of substitution does not occur for functions such as `min` or `max`, for which both arguments must be evaluated and compared to return the correct result.

The distinction between `ifelse` and `min` or `max` can be seen more clearly by referring again to Listing 3.4. As described in Step 3b of Definition 2.6.14 (Generalized McCormick Relaxations) [121], the relaxations of a product are comprised of `min`/`max` functions for the calculations of both intermediate terms and the relaxations themselves. If these rules were implemented using only `min` and `max`, every multiplication operation in the intermediate terms would need to be evaluated prior to performing the relaxation evaluations. Instead, we recognize that the decisions in the intermediate functions can be simplified to checks on the bounds of g_1 and g_2. For example, for α_1, since $g_1^{cv}(\mathbf{z}) \leq g_1^{cc}(\mathbf{z})$ by definition, the correct output can be ascertained by checking if $g_2^L \geq 0$. In this case, replacing `min` with `ifelse` eliminates the need to perform both multiplication operations of $g_2^L g_1^{cv}(\mathbf{z})$ and $g_2^L g_1^{cc}(\mathbf{z})$, and also eliminates the need to evaluate a relaxation of g_1 that is not ultimately chosen, which may be expensive if the expression is deeply nested. A similar substitution of `min`/`max` for `ifelse` cannot easily be made for the calculation of g^{cv} or g^{cc} since a simpler conditional check is not readily available. Thus, the implementation of multiplication in `SCMC`, as shown in Listing 3.4, proceeds by checking the bounds of the factors using `ifelse` conditional checks with each branch terminating in a `min` or `max` function with the correct results of the intermediate functions interpolated. It is worth noting that, by this construction, the only code branches required for multiplication are those that check the signs of the bounds of the factors—`min` and `max` do not result in branches in the code since both arguments are always evaluated to determine the output, regardless of the result.

It is also important to address the complexity of these source-code-generated functions. The aim of `SCMC` is to create single compiled functions that produce McCormick relaxations on any interval domain, and, by design, the mathematical structure of McCormick relaxations depends on the domain. Therefore, any function that tries to encapsulate all possible domains will necessarily have many different potential calculations to perform, separated in `SCMC`-generated functions by `if-else` statements. Table 3.1 shows the number of conditional branches contained in convex relaxations for a set of common mathematical expressions. There are more branches than might be expected from a naïve implementation of the rules in, e.g., Step 3b of Definition 2.6.14 due to an added intersection between relaxations and the inclusion monotonic interval extensions. This operation is realized in [121, Def. 9, Step 6] and referred to as the *cut* operation, which is formalized in Ye and Scott [158, Def. 11]. Due to how `SCMC` composes McCormick-based relaxations together, the total number of conditional branches grows rapidly, even for visually simple expressions. Although the compiled versions of these functions return evaluations quickly (see Subsection 3.3.4), combining more than a few variables together, such as for multiplication or division, results in lengthy expressions that can

Table 3.1: The total numbers of conditional branches that are required to express a convex relaxation equation using the SCT approach are reported for commonly encountered multi-term/multi-factor equation forms (e.g, sums and products). Cases involving summations exhibit a number of conditional branches that are independent of the number of terms. Cases involving products (and divisions) exhibit a combinatorial dependence of the number of conditional branches on the number of factors in the operation. Dashes in the table indicate expressions that were too long to evaluate.

Equation Form	Cardinality of Terms/Factors			
	2	**3**	**4**	**5**
$\sum x_i$	0	0	0	0
$\exp(\sum x_i)$	6	6	6	6
$(\sum x_i)^2$	6	6	6	6
$\prod x_i$	16	736	20,608	—
$\exp(\prod x_i)$	262	10,390	281,014	—
$(\prod x_i)^2$	310	11,206	294,118	—
$x_1/(\prod_{i=2}^{m} x_i)$	314	32,426	321,578	—

consume significant memory resources of the computing system during compilation. If expressions that are too complicated are passed through `SCMC`, it may cause Julia to stall in the function creation step. For this reason, a warning system is implemented in `SCMC` that attempts to determine a priori whether an expression will be too long to handle in a reasonable time frame. If so, `SCMC` will abort the calculation and inform the user.

3.3.3 Utilities

In this section, details are provided on how to use two main `SCMC` utilities to evaluate the bounds and relaxations of mathematical expressions of interest: `convex_evaluator` and `all_evaluators`.

The `SCMC` function `convex_evaluator` returns a source-code-generated function that provides evaluations of convex relaxations of the mathematical expression of interest. This utility is designed for situations where only convex relaxations of an expression are desired, which may be the case with relatively simple constraints or an objective function in an optimization problem. Listing 3.5 provides an example of how `convex_evaluator` can be used to generate an evaluator function, as well as how the generated function can be used. Specifically, the generated function takes numerical values corresponding to the McCormick tuples of the dependent variables, grouped by tuple and sorted alphabetically according to the `var_order` output, and returns a convex relaxation of the original expression on the desired domain, evaluated at the desired point in that domain. Similarly, the generated function can be broadcast over vectors of numerical values to return evaluations of a convex relaxation of the original expression at multiple locations, with no requirement that the domains of the dependent variables be similar for different elements of the input vectors.

The function `all_evaluators` returns four functions corresponding to the McCormick tuple of the

Listing 3.5: An example is provided to demonstrate how the `convex_evaluator` function can be used to obtain convex relaxations of a desired expression. The variables in the `var_order` vector, with the McCormick tuples sorted alphabetically, are shown in the order in which they should be passed into the evaluator function. Note that the created `expr_cv_eval` function may also be broadcast over vectors of values to obtain relaxations of multiple points simultaneously.

```
using SourceCodeMcCormick, Symbolics
@variables x, y
expr = exp(x/y) - (x*y^2)/(y+1)
expr_cv_eval, var_order = convex_evaluator(expr)
xcv, xcc, xlo, xhi = 1.0, 1.0, 0.5, 3.0
ycv, ycc, ylo, yhi = 0.7, 0.7, 0.1, 2.0

julia> @show var_order;
  var_order = Num[x_cv, x_cc, x_lo, x_hi, y_cv, y_cc, y_lo, y_hi]

julia> convex_relaxation = expr_cv_eval(xcv, xcc, xlo, xhi, ycv, ycc, ylo, yhi)
  0.22836802303235837
```

input expression, i.e. a convex relaxation, a concave relaxation, and the lower and upper bounds of an inclusion monotonic interval extension. Having access to all elements of the McCormick tuple may be useful for analysis purposes, and when dealing with cumulative mappings such as when the original expression is too complicated to handle in a single `convex_evaluator` call due to memory limitations. Listing 3.6 shows an example of how `all_evaluators` can be used to calculate relaxations for a complicated expression that could not otherwise be handled in one step. This approach exploits the notion that, when a computational graph is contracted, we are applying the concepts of composite relaxations (Def. 2.6.8) and cumulative mappings (Def. 2.6.7). Namely, observe that all nodes with index $k > n$ (i.e., factors) only rely on the McCormick tuples corresponding to the connected nodes immediately prior in the graph. For example, in Figure 3.1, the source nodes correspond to factors v_1 and v_2, which were defined to be the independent (base) variables. However, for more complicated expressions, v_1 and v_2 could instead represent cumulative mappings, with their corresponding relaxations in the McCormick tuple calculated as composite relaxations. Although the process shown in Listing 3.6 currently requires user-defined functions, future work will explore the automation of this step so that complicated expressions can be passed seamlessly to `SCMC` without any external combination steps such as described.

3.3.4 GPU Compatibility

A key design objective of `SCMC` was compatibility with GPGPU. This was a goal primarily due to the potential for substantial speed benefits if the software can exploit the massive parallelization of GPUs, but also because a successful implementation would allow the speed of the associated global optimization algorithm to scale based on available hardware resources rather than CPU clock speed.

Listing 3.6: An example of how an expression that is too complicated for SCMC to handle in a single function evaluation can be separated into intermediate expressions and recombined with the use of the all_evaluators function in user-defined code. Arbitrarily complicated expressions can be handled in this manner, exploiting the notions of composite relaxations (Def. 2.6.8) and cumulative mappings (Def. 2.6.7).

```
# Expression: f = (x*y*z - 3*(x*y) + 4*(z/y)) / (x^2 + y^2 + z^2)
using SourceCodeMcCormick, Symbolics

@variables x, y, z, temp1, temp2

num_cv, num_cc, num_lo, num_hi, _ = all_evaluators(x*y*z - 3*(x*y) + 4*(z/y))
den_cv, den_cc, den_lo, den_hi, _ = all_evaluators(x^2 + y^2 + z^2)
f_cv, _ = convex_evaluator(temp1/temp2)

function div_cv(xcv, xcc, xlo, xhi, ycv, ycc, ylo, yhi, zcv, zcc, zlo, zhi)
  input = [xcv, xcc, xlo, xhi, ycv, ycc, ylo, yhi, zcv, zcc, zlo, zhi]
  temp1_cv = num_cv.(input...)
  temp1_cc = num_cc.(input...)
  temp1_lo = num_lo.(input...)
  temp1_hi = num_hi.(input...)
  temp2_cv = den_cv.(input...)
  temp2_cc = den_cc.(input...)
  temp2_lo = den_lo.(input...)
  temp2_hi = den_hi.(input...)
  division_cv = f_cv.(temp1_cv, temp1_cc, temp1_lo, temp1_hi,
                      temp2_cv, temp2_cc, temp2_lo, temp2_hi)
  return division_cv
end
```

An example is presented in Listing 3.7 to illustrate how the generated SCMC functions can be used to perform evaluations of relaxations on a GPU. Thanks to the development of CUDA.jl by Besard et al. [16] (a Julia wrapper for the CUDA language [92] for GPGPU on NVIDIA GPUs), it is simple to accelerate these functions by broadcasting them over CUDA arrays instead of standard arrays. SCMC functions are designed to only employ functions internally that are compatible with GPGPU, and thus no additional considerations are required in terms of problem formulation or use of SCMC function generators.

Benchmarking results are presented in Table 3.2 for convex evaluator functions created for a set of commonly encountered multi-term equation forms. Given that the intent of SCMC is to make use of massive parallelization, the benchmark times in this table represent the time to calculate one million relaxation values for randomly selected points from the variable domain of $X = [0.9, 1.1]^n$. Three times are shown for each cell, representing the time to perform the calculations using the multiple dispatch approach of McCormick.jl, the time using a SCMC-generated function using one core of an Intel Xeon W-2195 CPU (2.3 GHz/4.3 GHz base/turbo), and the time using a SCMC-generated function using an NVIDIA Quadro GV100 GPU. Due to memory limitations, evaluator functions were only generated for expressions with fewer than 10^5 conditional branches (see Table 3.1). It should also be noted that the McCormick.jl method, by using the MC type and multiple

Listing 3.7: An example of how relaxation evaluator functions can be used with CUDA arrays to calculate relaxations on a GPU. In this example, the domain of x is left unchanged for all points for simplicity, but this is not a requirement. Points on any domain may be passed within the same function evaluation, all of which will be evaluated in parallel. Note also that double-precision floating-point numbers must be specified, as CUDA arrays will default to single-precision floating-point values.

```
using SourceCodeMcCormick, Symbolics, CUDA
@variables x
expr = (x-0.5)^2+x
f_cv, order = convex_evaluator(expr)

xcv_GPU = CUDA.rand(Float64, 1000000)
xcc_GPU = copy(xcv_GPU)
xlo_GPU = CUDA.zeros(Float64, 1000000)
xhi_GPU = CUDA.ones(Float64, 1000000)
convex_relaxations = f_cv.(xcv_GPU, xcc_GPU, xlo_GPU, xhi_GPU)
```

dispatch, simultaneously calculates the following for the desired expression: convex and concave relaxations, inclusion monotonic interval extensions, and corresponding subgradients. Notably, the `SCMC`-generated functions evaluated using a CPU are considerably faster than the `McCormick.jl` approach in all cases, by a factor of 1.1–25.4x. Using a GPU, however, the `SCMC` functions are consistently nearly three orders of magnitude faster than the `McCormick.jl` approach. This result showcases the speed that can be gained by utilizing a parallel computing approach.

Designing an algorithm to be GPU compatible requires acknowledgment of the limitations of GPU architectures. While CPUs are built with a small number of fully independent cores that can execute independent tasks simultaneously, GPUs are designed with a much larger number of cores that, in general, operate in parallel by following a single instruction that is applied to multiple data points (SIMD). For tasks where the same instructions are executed a large number of times, SIMD operation can be significantly faster than operation on x86 hardware, by virtue of the greater number of cores in typical GPUs as compared to typical CPUs. This concept is illustrated in Figure 3.2 for the evaluation of a convex relaxation $f^{cv} : X \to \mathbb{R}$ at several different points on a single core of a CPU versus a GPU. However, if the instructions differ depending on the data, there can be significant time penalties that impact the overall computation time.

Consider, for example, GPUs produced by NVIDIA. To execute a GPU function—called a kernel—on a collection of data points, the task will be handled by a collection of threads, which can be thought of as individual execution units. NVIDIA GPUs issue instructions to *warps*, which are groups of 32 threads, as illustrated in Figure 3.2. In the ideal case, where the same instruction applies to every data point, each thread in a warp will operate in parallel to compute the total of 32 results. However, if the instructions differ between threads in a warp—a phenomenon called

Table 3.2: The total times to evaluate convex relaxations of the equation of the listed form with one million randomly selected points in the domain $X = [0.9, 1.1]^n$ are tabulated for three different methods: using the `McCormick.jl` approach (MC), the `SCMC` approach operating on one core of an Intel Xeon W-2195 CPU (SCMC (CPU)), and the `SCMC` approach on an NVIDIA Quadro GV100 GPU (SCMC (GPU)). The `SCMC` method, when performed on the CPU, calculates relaxations considerably faster than the `McCormick.jl` method in every case. When the `SCMC` approach is paired with a GPU, the evaluation times are nearly three orders of magnitude shorter than the `McCormick.jl` method. Note that the `McCormick.jl` method, by design, is also computing a concave relaxation, inclusion monotonic interval extension, and subgradients of the relaxations.

Equation Form	McCormick Style	Unit	Cardinality of Terms/Factors **2**	**3**	**4**	**5**
$\sum x_i$	MC	ms	15.923	28.457	42.387	61.937
	SCMC (CPU)	ms	0.851	1.292	1.861	2.434
	SCMC (GPU)	ms	0.049	0.063	0.076	0.091
$\exp(\sum x_i)$	MC	ms	79.753	92.130	99.210	117.020
	SCMC (CPU)	ms	6.817	9.528	12.755	30.842
	SCMC (GPU)	ms	0.082	0.113	0.142	0.171
$(\sum x_i)^2$	MC	ms	53.276	67.735	76.703	98.066
	SCMC (CPU)	ms	9.941	14.412	22.479	28.354
	SCMC (GPU)	ms	0.109	0.152	0.194	0.236
$\prod x_i$	MC	ms	33.154	65.497	93.542	127.982
	SCMC (CPU)	ms	7.666	21.990	77.212	—
	SCMC (GPU)	ms	0.110	0.154	0.211	—
$\exp(\prod x_i)$	MC	ms	97.459	132.791	161.305	201.233
	SCMC (CPU)	ms	25.075	90.600	—	—
	SCMC (GPU)	ms	0.113	0.209	—	—
$(\prod x_i)^2$	MC	ms	77.450	109.112	141.369	179.004
	SCMC (CPU)	ms	26.583	96.794	—	—
	SCMC (GPU)	ms	0.113	0.216	—	—
$x_1/(\prod_{i=2}^m x_i)$	MC	ms	83.475	121.933	154.147	198.200
	SCMC (CPU)	ms	9.837	41.361	—	—
	SCMC (GPU)	ms	0.113	0.167	—	—

warp divergence—the GPU will be unable to issue the different instructions to different threads simultaneously. One way GPUs can handle warp divergence is by first activating only those threads in the warp that will execute one set of instructions, then activating the threads that will execute an alternate set of instructions, and so on. This is inherently slower than the ideal case because the massive parallelization made possible by the GPU architecture is not being used to the fullest extent. In the worst case, where each thread in a warp executes a different instruction, the total calculation time may increase by a factor of approximately 32. On the other hand, if there would be significant warp divergence but the data points can be sorted prior to calculation to minimize the number of unique instructions per warp, the computation time can be greatly reduced.

This architecture limitation is critical to understand and account for when using `SCMC`, since the relaxation evaluator functions are composed of deeply nested `if-else` statements. Particularly, if worst-case GPU performance comes from warp divergence resulting in 32 unique instructions per

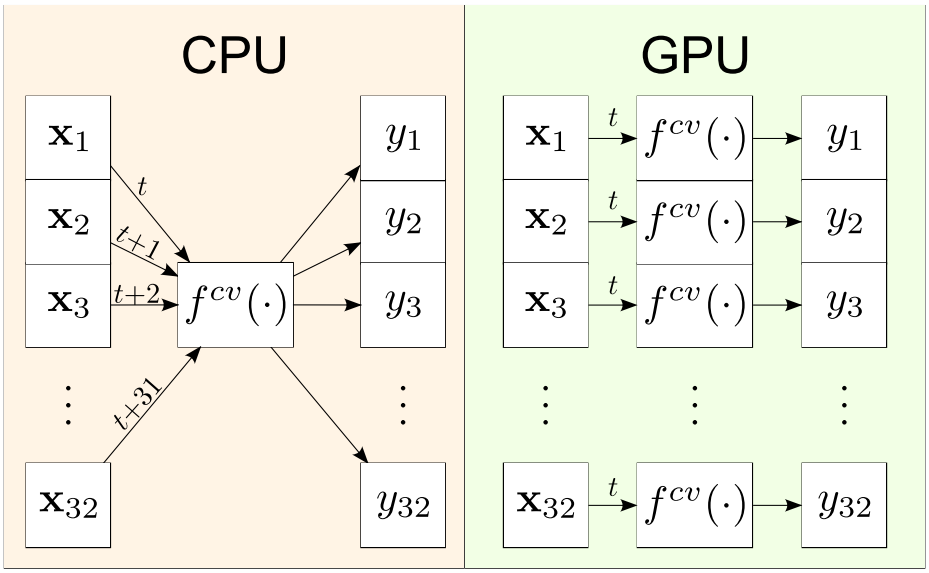

Figure 3.2: This figure illustrates a high-level simplification of the computing differences between CPU and GPU architectures. A convex relaxation $f^{cv} : X \to \mathbb{R}$ is evaluated at 32 points with reference to an instruction set sequence time index t. A CPU executes the instruction sets on the data points sequentially (ignoring multithreading). In contrast, the architecture of the GPU enables instruction sets to be executed on many data points simultaneously. 32 points $\mathbf{x}_i \in X$ and function evaluations are illustrated to coincide with a single warp on the GPU.

warp, and the `SCMC` functions have hundreds to tens of thousands of unique branches, intuitively these functions should practically always result in very poor computational performance. Fortunately, there are three important factors that work to the benefit of `SCMC`:

1. The number of conditional branches is large because every possible control-flow case must be considered. However, conditional branches are not equally likely to be encountered in practice. For example, 57 paths from the 314 conditional branches in the expression representing a convex relaxation of x/y stem from checks that the division operation is defined on the given domain. If the user is working on variable domains where division always returns a real value, these conditional branches will never be encountered.

2. The substitution process of `SCMC` results in nested `ifelse` statements but does not currently check for conflicting conditional statements within the tree. Consequently, many conditional branches may be unreachable.

3. Functions such as x/y fall victim to warp divergence and performance degradation in the case where threads in the same warp try to access different conditional branches. For McCormick relaxations, different conditional branches might be accessed when inputs or intermediate values switch from positive to negative, or vice versa. In a typical use case of B&B, which is the

expected application space of `SCMC`, the SCT-generated functions will likely be used to evaluate multiple points within individual B&B nodes, which in most cases represent comparatively small regions of the overall search space. Within a single node, it is unlikely that multiple conditional branches will need to be accessed, since the points to evaluate will be numerically similar. If points from multiple nodes are to be evaluated and the nodes are far from one another, it is possible that different conditional branches will need to be accessed for points associated with the two nodes, but this is far from the worst-case warp divergence scenario of 32 unique branch paths. This point is further helped by the problem scale. As detailed in Subsection 3.3.5, the current application method requires the evaluation of $2n + 1$ points per node, where n is the problem dimensionality. For problems involving higher dimensions, larger numbers of numerically similar points will be grouped together, which will further reduce the incidence of warp divergence.

In summary, although these functions have a large number of conditional branches, the existence of these conditional branches is not necessarily problematic. The key factor is warp divergence, which is likely to be low in practice within typical optimization problems that will be solved using B&B.

One critical aspect to address is how the speed of the evaluator functions is affected by warp divergence and how the functions can be expected to perform in a spatial B&B algorithm. To address the large potential impact of warp divergence, the speeds of `SCMC` convex evaluator functions for several forms of mathematical expressions are presented in Table 3.3 for a set of potential computational scenarios. These range from the most ideal scenario, in which all the input points and domains are numerically similar—such as the case in Table 3.2—to the worst-case scenario, where the input points and domains are numerically dissimilar and randomized. There are additional intermediate cases meant to represent how these functions may be expected to behave in a B&B implementation. Notably, in optimization problems where the branching variables are constrained to numerically similar values. For example, positive values over a range that is not expected to change the sign of the resulting calculation, the `SCMC` functions should operate at speeds close to the ideal case, since there is expected to be low warp divergence. This is also true for multiplication, which is implemented to only have warp divergence when the signs of the operands change (see discussion in Section 3.3.2). In more complicated cases, such as optimization problems where the variables are all close to zero and can take on positive or negative values, the `SCMC` functions perform slower than the ideal case, but still substantially faster than the worst case. Generally, these results indicate that although warp divergence is a critical factor affecting performance, the degree of warp divergence is

Table 3.3: The total times to evaluate convex relaxations of the equation of the listed form with $\mathbf{x} \in X \in \mathbb{IR}^3$ are tabulated. These timings are for 350k evaluation points—equivalent to 50k B&B nodes using the blackbox sampling method of Song et al. [131]—using `SCMC` source-code-generated functions on a GV100 GPU. The Ideal Case represents a potentially unrealistic scenario with numerically similar inputs, where all evaluation points are randomly selected from a small, identical domain of $X = [0.9, 1.1]^3$. The Typical B&B Case represents a commonly encountered B&B scenario in which we evaluate 7 points per node and the 50k nodes are randomly selected non-overlapping subdomains of $X = [0, 3]^3$. The Complicated B&B Case represents a B&B scenario where we evaluate 7 points per node and the 50k nodes are randomly selected non-overlapping subdomains of $X = [-1.5, 1.5]^3$ (i.e., spanning negative and positive values). In the two B&B cases, the order of the nodes is randomized to emulate a more realistic B&B situation. The Worst Case represents a situation where we evaluate 1 point per node and the 350k nodes are randomly selected (possibly overlapping) subdomains of $X = [-1.5, 1.5]^3$. In the complicated B&B and worst case scenarios, the $\exp(\Pi x_i)$, $(\prod x_i)^2$, and division formulations exhibit significant time penalties from warp divergence.

Equation Form	Unit	Ideal Case	Typical B&B Case	Complicated B&B Case	Worst Case
$\sum x_i$	ms	0.029	0.029	0.029	0.029
$\exp(\sum x_i)$	ms	0.047	0.047	0.047	0.047
$(\sum x_i)^2$	ms	0.060	0.060	0.060	0.060
$\prod x_i$	ms	0.063	0.063	0.239	0.713
$\exp(\prod x_i)$	ms	0.093	0.094	3.801	12.665
$(\prod x_i)^2$	ms	0.097	0.099	3.696	15.776
$x_1/(\prod_{i=2}^{m} x_i)$	ms	0.072	0.090	1.823	2.078

likely to be low in typical problems, and thus the generated functions should perform similarly to their ideal speeds in practice.

3.3.5 Global Optimization with SourceCodeMcCormick.jl

So far, we have focused on the details of how the `SCMC` approach can be used to quickly obtain a large number of pointwise evaluations of convex relaxations of mathematical expressions. In this section, we develop a novel parallelized B&B framework that can exploit these fast pointwise evaluations of relaxations in parallel. Since this parallel pointwise evaluation approach does not align naturally with how global optimizers like EAGO normally function, careful considerations are required to effectively exploit this approach. Specifically, EAGO and global optimizers like EAGO most commonly assume that only one B&B node is evaluated at a time. Since the `SCMC` approach makes the best use of parallelization when a large number of points are evaluated, and because it is capable of evaluating different variable domains (nodes) simultaneously, a B&B method that uses `SCMC` should have the capability to solve lower- and upper-bounding problems for multiple B&B nodes in parallel rather than in series. Additionally, modern global optimization routines that utilize McCormick-based relaxations also typically use subgradients of their corresponding convex/concave relaxations for domain reduction and accelerating lower-bounding problems, which are currently not

computed by SCMC. Although the inclusion of subgradients could be beneficial, solvers like EAGO typically use them to generate subtangent hyperplanes used for LP lower-bounding problems, which are then sent to a separate LP solver to obtain a lower bound. If subgradients are to be incorporated into SCMC, additional developments are necessary to obtain batched solutions to large numbers of LPs on a GPU so that this process does not become the bottleneck for parallelization. This is a future area of research.

Instead, for the choice of lower-bounding method, recent work of Song et al. [131] provided a method for calculating a mathematically rigorous lower bound of a convex function via pointwise evaluations. This method starts by considering an interval $X \in \mathbb{IR}^n$ and a convex function $f : X \to \mathbb{R}$. We define the midpoint $\mathbf{w}^{(0)}$ of X, and for each index $i \in \{1, \ldots, n\}$, we define two vectors $\mathbf{w}^{(\pm i)}$, as follows:

$$\mathbf{w}^{(0)} := \frac{1}{2}(\mathbf{x}^L + \mathbf{x}^U),$$
$$\mathbf{w}^{(\pm i)} := \mathbf{w}^{(0)} \pm \frac{\alpha_i}{2}(x_i^U - x_i^L)\mathbf{e}^{(i)},$$

with $\mathbf{e}^{(i)}$ as the i-th unit coordinate vector in $\mathbb{R}^n$ and with a predetermined step length $\alpha_i \in (0, 1]$. We continue by obtaining the values of the convex function at each of these points (i.e., $y_0 := f(\mathbf{w}^{(0)})$ and $y_{\pm i} := f(\mathbf{w}^{(\pm i)})$) and, using the evaluations at these points, we calculate a scalar lower bound as:

$$f^{LBD} := y_0 - \sum_{i=1}^{n} \left(\frac{\max(y_{+i}, y_{-i}) - y_0}{\alpha_i} \right).$$

The full details of the proof may be found in Song et al. [131].

This method is especially convenient for the SCMC approach because a known number of pointwise evaluations $(2n + 1)$ is required regardless of the complexity of the underlying convex function. Furthermore, since the locations of these points can be determined a priori based on the domain of a B&B node, it is simple to determine which points the convex relaxations must be evaluated at for multiple nodes before any calculations are completed. In effect, by applying this method, the points and domains that will need to be evaluated can be gathered prior to calculation, and then all the pointwise evaluations that are needed can be calculated in a single step for an arbitrary number of nodes, limited mainly by VRAM capacity.

To address the new parallel node evaluation paradigm, an extension of the EAGO solver was created to handle multiple nodes simultaneously. Pseudocode showing the overall algorithm is provided by Algorithm 2, which is also illustrated in Figure 3.3. An important distinction between

this algorithm and that of the default EAGO solver is how upper-bounding problems are handled. In the default EAGO B&B method, the upper-bounding problem is solved for every node shallower in the tree than a predetermined threshold, and then it is solved depending on a depth-based probability distribution for deeper nodes. This approach works well when nodes are individually evaluated, as the number of nodes after several iterations is still low, on the order of 10^1. With this small number of nodes, solvers such as IPOPT [148], which may take $10^1 - 10^2$ ms to run, are only called a small number of times. In Algorithm 2, by the time the first iteration begins, the problem domain has already been partitioned into $n_\sigma = q$ subdomains/nodes by the preprocessing step. Depending on the machine and user settings, q may be large: preliminary testing of some of the numerical examples in Section 3.4 found that $q = 2^{13} = 8192$ worked well. Although IPOPT is a highly performant solver, running it on all $n_\sigma = q$ nodes would be detrimental to the speed of the overall algorithm, and choosing a subset of the n_σ nodes on which to use IPOPT may be no better than random guessing. For this reason, IPOPT is only used once, at the root node before the initial partitioning step, to obtain a reasonable starting upper bound for global optimization. Instead, for each iteration of the main algorithm, we note that pointwise evaluations of the objective function can indeed function as valid upper bounds. The main power of `SCMC` lies in its speed of performing parallelized pointwise evaluations, so the addition of one point per node to function as a valid upper bound is computationally inexpensive.

Although the lower bounds generated for each individual node could be made tighter by utilizing subgradient information, this is compensated for by the much greater node throughput and the consequently smaller subdomains that these calculations are being performed on for each iteration of the algorithm. One important consequence of this algorithm, however, is related to the large number of nodes generated in the main B&B stack. While the base implementation of B&B would require 10^5 iterations to reach a stack size of 10^5 nodes (without fathoming), with $n_\sigma = q$ set to 2^{13}, this number of nodes is reached after just twelve iterations of Algorithm 2. Storing such a large number of nodes may be highly memory intensive. This issue is exacerbated by the decreased tightness associated with each node due to the choice of lower-bounding method, which results in fewer nodes being fathomed than if the same nodes were evaluated using a tighter method. Consequently, while many more nodes can be processed using this algorithm than the more typical single-node implementations, the abundant use of memory quickly becomes a limiting factor in the overall algorithm speed. However, if subgradient calculations can be incorporated into the `SCMC` functionality in the future, this issue may be ameliorated to a large extent.

Algorithm 2 Parallel-Evaluation B&B (ParBB)

1. Solve the original program to local optimality at the root node on the domain $X \in \mathbb{IR}^n$ using IPOPT (or some other NLP solver)
2. Partition X into $n_\sigma := q$ similar-sized intervals $X^{(i)}$ such that $X = \cup_{i=1}^{q} X^{(i)}$, and push them to the main B&B stack
3. While the problem is not converged (or the stack is nonempty):
   (a) Perform fathoming (pop and delete d nodes from the stack if their lower bounds preclude them from containing a problem solution). $n_\sigma := n_\sigma - d$
   (b) Pop the $n_\sigma^{\text{sub}} := \min\{n_\sigma, q\}$ nodes with the lowest lower bounds in the main B&B stack to the parallel evaluation substack
   (c) Perform in parallel, for each node in the substack:
      i. Calculate the values of the $2n + 1$ points to be evaluated and add them to the evaluation queue
      ii. Calculate the node's midpoint and add it to the evaluation queue; evaluating the objective function at this point returns an upper bound
      iii. Obtain convex relaxation values and a pointwise evaluation of the objective function by passing the evaluation queue to the `SCMC` convex evaluator for the objective function
      iv. Apply the blackbox sampling method to calculate the node lower bound based on the $2n + 1$ evaluation points
      v. Apply the lower bound result to the node in the substack
      vi. Apply the upper bound result to the node in the substack
   (d) Branch each node in the substack and push all new partitions to the main B&B stack. $n_\sigma := n_\sigma + n_\sigma^{\text{sub}}$
   (e) Check for problem convergence

3.4 Numerical Examples

While the parallel B&B algorithm (ParBB), Algorithm 2, may be applied to a range of optimization problems, in this section we will explore the application of the new methods and software to the optimization of a machine learning model (Sec. 3.4.1) and parameter estimation problems (Sec. 3.4.2–3.4.3). The machine learning problem of Section 3.4.1 seeks to determine optimal inputs to a trained artificial neural network (ANN), formulated generally as

$$\min_{\mathbf{x} \in X \in \mathbb{IR}^n} f^{\text{ANN}}(\mathbf{x}), \tag{3.1}$$

where f^{ANN} represents a scalar output of the ANN that relates to some system performance metric. The parameter estimation problems of interest are formulated generally as:

$$\mathbf{p}^* \in \arg \min_{\mathbf{p} \in P \subset \mathbb{R}^{n_p}} \sum_{i=1}^{n_d} (y_i(\mathbf{p}) - y_i^{\text{data}})^2 \tag{3.2}$$

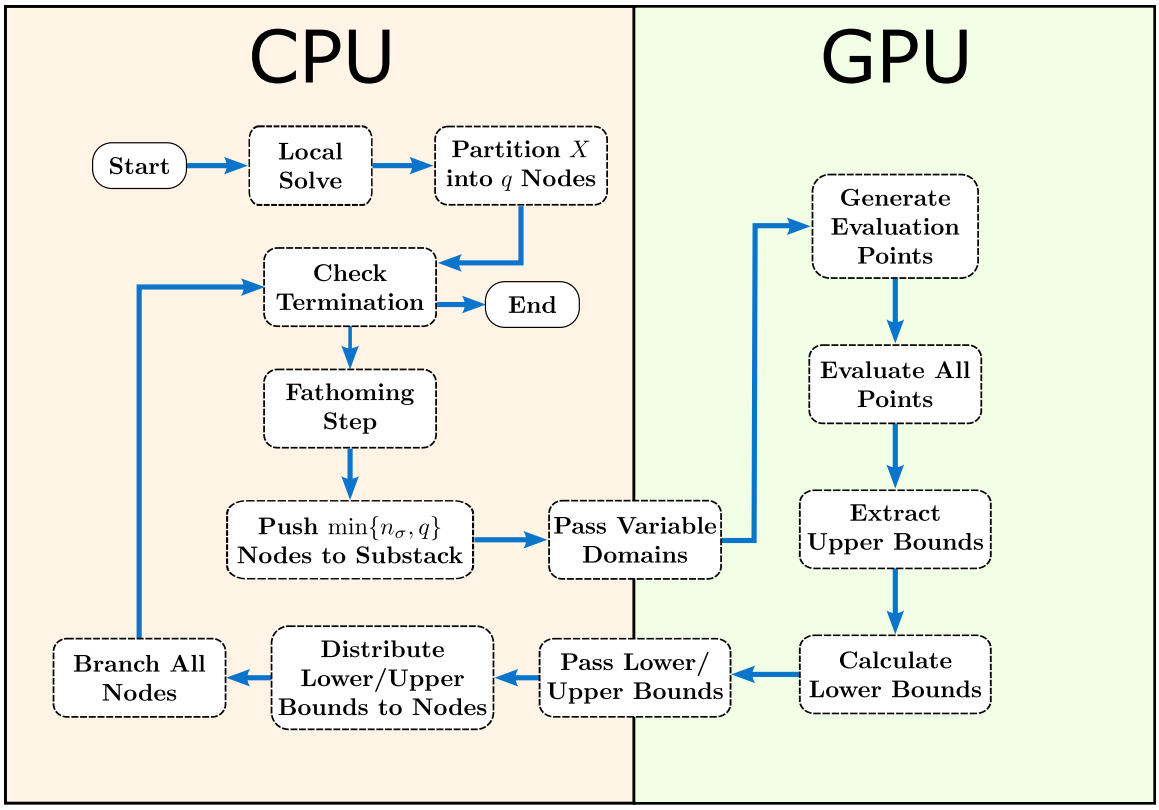

Figure 3.3: A block flow diagram representation of the parallel B&B algorithm (Algorithm 2), indicating the hardware on which the algorithm steps are executed. Operations corresponding to node storage and management are kept on the CPU, whereas numerical operations are offloaded to the GPU. Steps involving the transfer of information between the CPU and GPU are located at the intersection of the CPU and GPU regions.

The objective function in (3.2) is the sum of squared error (SSE) between the predictions of a model of interest $\mathbf{y}(\mathbf{p})$ and a set of experimental data $\mathbf{y}^{\text{data}}$, where $\mathbf{p}$ is the uncertain parameter vector and n_d is the number of experimental data points. In the following sections, two problems of this form will be used for numerical experiments to benchmark and demonstrate the use of the proposed approaches.

An important point to note is that the ParBB algorithm is, at the time of writing, an early-stage implementation to demonstrate the utility of `SCMC` in a GPU-based deterministic global optimization algorithm. ParBB by itself is not yet a mature solver and lacks bounds tightening techniques and features that are common across existing solvers, such as constraint propagation, feasibility-based bounds tightening, and optimization-based bounds tightening, among others. For this reason, in order to showcase a more direct comparison between how ParBB and the other solvers handle the following numerical examples, the other solvers (BARON [119], ANTIGONE [98], SCIP [19, 145],

Table 3.4: Parameters used to obtain "vanilla" versions of common deterministic global optimizers. These versions are meant to represent B&B approaches that do not make use of pre-processing or post-processing techniques, relying instead on lower- and upper-bounding techniques within each B&B node.

Solver	Parameters

BARON (Vanilla)	`LBTTDo = 0`
	`MDo = 0`
	`NumLoc = 0`
	`OBTTDo = 0`
	`PDo = 0`
	`TDo = 0`

ANTIGONE (Vanilla)	`fbbt_improvement_bound = 0` `use_obbt = 0`
SCIP (Vanilla)	`presolving/maxrounds = 0` `propagating/maxrounds = 0`

EAGO (Vanilla)	`cp_depth = 0`
	`dbbt_depth = 0`
	`fbbt_lp_depth = 0`
	`obbt_depth = 0`

EAGO (Blackbox Sampling)	`cp_depth = 0`
	`dbbt_depth = 0`
	`fbbt_lp_depth = 0`
	`obbt_depth = 0`

and EAGO [155]) will have their pre- and post-processing techniques deactivated, using the settings presented in Table 3.4. These versions of the solvers, which rely only on their branching and bounding routines, are referred to in this section as "vanilla" versions of these solvers. For each of the examples, reactivating these processes greatly enhances the ability of the solvers to address the problems. Additionally, to better demonstrate the differences between subgradient-based methods and subgradient-free methods, such as what is used in the ParBB algorithm, a version of EAGO is included in the examples that uses the same "vanilla" settings as previously described in addition to the blackbox sampling method of Song et al. [131] as the lower-bounding method.

All numerical experiments in this work were run on a single thread of an Intel Xeon W-2195 2.30/4.30 GHz (base/turbo) processor with 64 GB of RAM running a Windows 11 Enterprise 22H2 operating system and an NVIDIA Quadro GV100 GPU with 32 GB of HBM2 VRAM (driver v552.22, CUDA v12.4). Julia v1.10.3 was used in conjunction with `BenchmarkTools.jl` v1.5.0 [33], `CUDA.jl` v5.3.3 [16], `EAGO.jl` v0.8.1 [155], `JuMP.jl` v1.21.1 [41], `McCormick.jl` v0.13.4 [153], `SourceCodeMcCormick.jl` v0.3.1, and `Symbolics.jl` v5.28.0 [59]. Vanilla EAGO is run using GLPK as the lower-bounding solver, using `GLPK.jl` v1.2.0 which is a wrapper for GLPK solver v5.0.1 [94]. Each problem was run with a time limit of two hours. For the ParBB algorithm and lower-bounding method, q was set to 2^{13} and α was set to 2×10^{-5} for each problem.

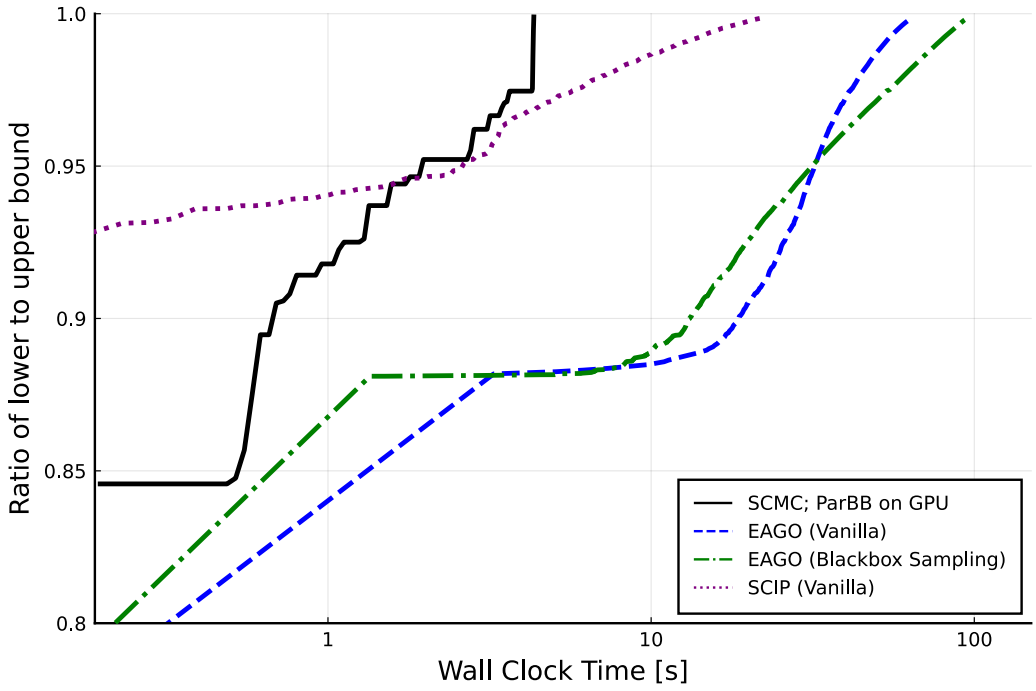

Figure 3.4: A convergence plot for the surrogate ANN model problem (Sec. 3.4.1) showing the performance of the novel ParBB algorithm (solid black) against vanilla versions of EAGO (dashed blue) and SCIP (dotted purple) and vanilla EAGO using the same lower-bounding routine as ParBB (dash-dotted green). ParBB converges the fastest in 4.3 s, followed by vanilla SCIP in 22 s, vanilla EAGO in 64 s, and EAGO with the blackbox sampling method in 95 s. As compared to the blackbox method implemented with EAGO, ParBB is roughly 22x faster, and despite not using subgradients, ParBB converges at least 5x faster than vanilla versions of the more mature solvers.

3.4.1 Surrogate ANN model

In Smith et al. [129], a surrogate ANN model of bioreactor productivity was constructed by fitting results from computationally expensive computational fluid dynamics simulations to a small ANN with one hidden layer. The surrogate model was then optimized to obtain ideal processing conditions for the bioreactor [129]. The objective is given by:

$$f^{\text{ANN}}(\mathbf{x}) = b_2 + \sum_{r=1}^{3} w_{2,r} \frac{2}{1 + \exp\left(-2\mathbf{w}_{1,r}^{\text{T}}\mathbf{x} + b_{1,r}\right)}, \tag{3.3}$$

with the weights vectors $\mathbf{w}_{1,r}$ (for $r = 1, 2, 3$) and $\mathbf{w}_2$, and biases $\mathbf{b}_1, b_2$ available in Smith et al. [129], and $\mathbf{x} \in X = [-1, 1]^8$.

The convergence plot for this problem is shown in Figure 3.4. The first and most direct comparison to make is between the ParBB algorithm and the blackbox sampling method used with the EAGO solver. Since neither solver uses pre- or post-processing techniques, and the same lower-bounding method is used, differences in the convergence profiles between these two methods are only

due to the batched node processing of ParBB and ParBB's use of GPU hardware. While EAGO with the blackbox sampling method is able to converge in 95 s, ParBB converges in 4.3 s, giving a speedup of roughly 22x. ParBB and this version of EAGO required 6.6×10^5 and 8.0×10^5 node evaluations to solve the problem to guaranteed global optimality, respectively. The discrepancy in nodes is due to differences in when nodes were branched on in batch versus serial node processing, but since ParBB is able to process nodes in parallel on faster hardware, the solution was obtained in significantly less time.

In this problem, it is interesting to note that the vanilla and blackbox sampling versions of EAGO give roughly comparable performance, converging in 64 s and 95 s, respectively. This indicates that, for this problem, the improved lower bounds that can be obtained by using subgradient information only slightly outweigh the added computational cost of generating and solving LPs. By parallelizing the blackbox sampling method on a GPU, ParBB is able to achieve considerable performance gains over the same method run on a CPU. Since the performance gained from this GPU parallelization is greater than the switch from subgradient-free to subgradient-based methods on a CPU, this results in ParBB achieving faster convergence than any of the vanilla versions of CPU-based solvers.

Comparing ParBB and the vanilla versions of SCIP and EAGO, for this problem, ParBB is able to reach convergence in only 4.3 s, outperforming the other tested solvers by more than 5x in terms of solution speed. Vanilla SCIP is the next fastest, converging in 22 s, followed by Vanilla EAGO, which solves the problem in 64 s. One curious aspect of these solution profiles is the conspicuous early plateau near 88% convergence identified by vanilla EAGO and EAGO with the blackbox sampling method, which is also present, but at a lower convergence, for ParBB. This plateau corresponds with finding a global solution and a lower bound that corresponds to that of an inclusion monotonic interval extension of the objective function on the domain. For the two versions of EAGO, since their solution methods start at the root node, an inclusion monotonic interval extension provides the first lower bound that is obtained, and improvements are only made once smaller domains with tighter relaxations are reached through branching. ParBB, on the other hand, starts by partitioning the root node into q subdomains, and then applies the blackbox sampling method, which is not guaranteed to provide results tighter than interval extensions. That is, across these subdomains, there are regions where applying the blackbox sampling method to the convex relaxation results in a lower bound that is lower than what can be obtained by applying interval arithmetic to the root node. Consequently, ParBB effectively starts at a lower plateau than what is found by both versions of EAGO, and only after eliminating these regions by further branching is ParBB able to improve the global lower bound. Still, due to its ability to use GPU hardware, ParBB is able to overcome

Table 3.5: The fitted parameters for (3.4), as obtained by Álvarez et al. [7], are tabulated for validation with the vapor-liquid equilibrium example (Sec. 3.4.2).

Parameter	Value
a_0	8.7369
a_1	27.0375
a_2	-21.4172
b_0	-2432.1378
b_1	-6955.3785
b_2	4525.9568

these differences to give overall faster performance than the vanilla versions of more mature solvers.

Finally, it should be noted that vanilla versions of ANTIGONE and BARON were unable to make progress on this problem within 2 hours, likely due to the curse of dimensionality as it applies to the higher dimensional space that these solvers operate in using the auxiliary variable method. With pre- and post-processing routines reactivated, ANTIGONE and BARON are able to solve this problem during preprocessing or at the root node in 0.13 s and 0.13 s, respectively.

3.4.2 Vapor Pressure Parameter Estimation

The second problem of interest comes from Álvarez et al. [7], in which the authors were studying the vapor-liquid equilibria of electrolyte solutions for triple-effect absorption heat pump systems. Through measurements of the vapor pressure over a range of alkaline nitrate salt concentrations and temperatures, the authors were able to compare the performances of the proposed salt mass fractions for operation in a high-temperature absorption cycle. In this work, we reexamine the experimental vapor pressure data from Álvarez et al. [7] to validate the parameters shown in Table 3.5, which were obtained by the authors to fit the polynomial expression:

$$\log(\pi^{\text{calc}}) = \sum_{i=0}^{2} a_i w^i + \frac{\sum_{i=0}^{2} b_i w^i}{T}, \tag{3.4}$$

where π is the vapor pressure in kPa, w is the total mass fraction of salts in the solution, and T is the temperature in K.

For this problem, we use a scaled least-squares method for parameter fitting with the objective function given as:

$$f(\mathbf{p}) = \sum_{i=1}^{N} \left[\frac{(\pi^{\text{calc}}(\mathbf{x}_i, \mathbf{p}) - \pi_i^{\text{exp}})}{\pi_i^{\text{exp}}} \right]^2, \tag{3.5}$$

where $\mathbf{p} = (a_0, a_1, a_2, b_0, b_1, b_2)$ is the parameter vector, π^{calc} is the vapor pressure calculated by

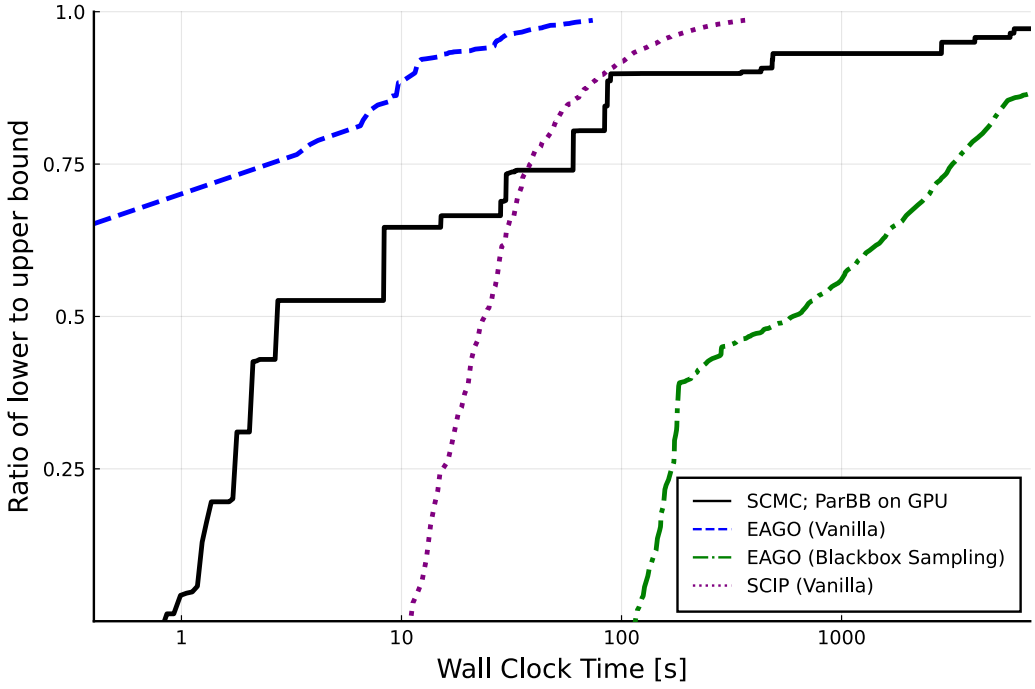

Figure 3.5: A convergence plot for the vapor pressure parameter estimation problem (Sec. 3.4.2) showing the performance of the novel ParBB algorithm (solid black) against vanilla versions of EAGO (dashed blue) and SCIP (dotted purple) and vanilla EAGO using the same lower-bounding routine as ParBB (dash-dotted green). While vanilla SCIP and EAGO are able to solve this problem within several hundred seconds due to their tighter relaxation techniques, the ParBB algorithm is unable to solve the problem due to the clustering problem, reaching a final convergence of 97.2% after 7200 s of runtime. The direct-comparison algorithm, EAGO with the blackbox sampling lower-bounding method, is also unable to solve the problem, reaching a final convergence of 86.5% after 7200 s of runtime. This level of convergence is reached by ParBB in 86 s, for a speed improvement of 84x.

(3.4) at a given condition $\mathbf{x}_i = (w_i, T_i)$, π_i^{exp} is the experimental vapor pressure data provided by Álvarez et al. [7, Tab. 1] corresponding to the condition $\mathbf{x}_i$, and N is the number of data points used for fitting, which is 50 in this example. Due to the presence of an exponential term with a complex argument used in the calculation of the objective function, this optimization problem is nonconvex, and with six parameters to fit, this problem is challenging for a B&B approach due to the curse of dimensionality. Bounds tightening techniques are particularly important for problems such as this with nested equation forms and multiple problem dimensions, since significant branching is required to obtain nodes that are sufficiently small in each dimension to provide tight relaxations. Parameter bounds for this problem were selected to be a box of diameter 0.2, centered around the parameters in Table 3.5.

As shown in Figure 3.5, vanilla implementations of EAGO and SCIP are able to converge to within the problem tolerance in 74 s and 360 s, respectively, while the methods that rely on the blackbox sampling method, including ParBB, are unable to converge within 7200 s. Notably, the

convergence profile of ParBB shows the apparent behavior of the clustering problem [40, 150], in which an undesirably large number of nodes in the vicinity of a global minimizer must be visited before a solution can be obtained. This problem results from the behavior of objective functions near their global minimizers [40], with the number of nodes having exponential dependence on the problem dimensionality [106]. In particular, and as observed by Wechsung et al. [150], improving the tightness of the relaxations—and by extension improving the tightness of the resulting node lower bounds—is critical to mitigating the clustering problem [150]. While the blackbox sampling method used by ParBB is capable of providing fast lower bounds, the decreased tightness of the technique as compared to subgradient-based lower-bounding methods makes ParBB more susceptible to the clustering problem than solvers that utilize subgradients. This is most visible towards the end of the convergence profile whereby ParBB achieves roughly 90% convergence in 89 s, after which only marginal progress is made for the remaining 7111 s of the problem runtime.

Comparing ParBB to EAGO with the same blackbox sampling method, EAGO's final convergence of 86.5% is reached by ParBB in 86 s, for a direct-comparison performance increase of 84x. Both ParBB and this version of EAGO are outpaced by the vanilla EAGO solver, which is able to exploit subgradient information to calculate tighter lower bounds and thereby mitigate the clustering problem. As a comparison to demonstrate the reduced node requirements due to its tighter lower-bounding method, vanilla EAGO took roughly 5.1×10^4 iterations (5.1×10^4 nodes explored) to reach its solution, whereas ParBB completed 2.4×10^4 iterations (2.0×10^8 nodes explored—over 3 orders of magnitude more than vanilla EAGO) in two hours and only achieved a convergence of 97.2%.

As in the previous example, vanilla versions of ANTIGONE and BARON were unable to make notable progress on this problem within two hours, likely due to the high dimensionality of their auxiliary variable method approaches in addition to their dependence on domain reduction techniques. With pre- and post-processing techniques activated, ANTIGONE and BARON are able to solve this problem in under 5 s each (4.0 s and 3.3 s, respectively).

3.4.3 Transient Absorption Kinetics Model

The final example of interest, originally described by Taylor [138], concerns the time-evolving concentrations of several chemical species after an initial laser flash pyrolysis. The problem consists of

the following system of ODEs:

$$\begin{aligned}
\frac{dx_A}{dt} =& k_1 x_Z x_Y - c_{O_2}(k_{2f} + k_{3f})x_A + \frac{k_{2f}}{K_2}x_D + \frac{k_{3f}}{K_3}x_B - k_5 x_A^2, \\
\frac{dx_B}{dt} =& c_{O_2} k_{3f} x_A - \left(\frac{k_{3f}}{K_3} + k_4\right) x_B, \\
\frac{dx_D}{dt} =& c_{O_2} k_{2f} x_A - \frac{k_{2f}}{K_2} x_D, \\
\frac{dx_Y}{dt} =& - k_{1s} x_Z x_Y, \\
\frac{dx_Z}{dt} =& - k_1 x_Z x_Y, \\
& x_A(0) = x_B(0) = x_D(0) = 0,\ x_Y(0) = 0.4,\ x_Z(0) = 140.
\end{aligned} \tag{3.6}$$

Here, x_j is the concentration of species $j \in \{A, B, D, Y, Z\}$, and there are known constants of $T = 273$, $K_2 = 46 \exp(6500/T - 18)$, $K_3 = 2K_2$, $k_1 = 53$, $k_{1s} = k_1 \times 10^{-6}$, $k_5 = 1.2 \times 10^{-3}$, and $c_{O_2} = 2 \times 10^{-3}$. The uncertain model parameters are $\mathbf{p} = (k_{2f}, k_{3f}, k_4)$ with $k_{2f} \in [10, 1200]$, $k_{3f} \in [10, 1200]$, and $k_4 \in [0.001, 40]$, and experimental data is given in terms of intensity, which has a known dependency on concentrations as $I^{\text{calc}} = x_A + \frac{2}{21}x_B + \frac{2}{21}x_D$, which originates from the Beer-Lambert law for relating measured absorbance to concentration with a correction for multiple species [126]. Thus, the objective function is formulated as:

$$f(\mathbf{p}) = \sum_{i=0}^{N} \left(I^{\text{calc}}(\mathbf{x}_i, \mathbf{p}) - I_i^{\text{exp}}\right)^2,$$

where $\mathbf{x}_i = (x_{A,i}, x_{B,i}, x_{D,i})$ represents the relevant discrete-time states at time i at which the intensity function is evaluated.

This problem was also addressed by Mitsos et al. [99], in which the ODE system was first discretized using the explicit Euler method to match the times of the 200 experimental data points, and then a McCormick relaxation library was applied to calculate convex and concave relaxations of the discrete-time state variables on the parameter domain. These relaxations were propagated forward through all time points to obtain a relaxation of the objective function. An alternative method of solving this problem using an implicit Euler discretization was utilized by Stuber et al. [133] and Wilhelm et al. [154], which was then solved using the global optimization of implicit functions approach. In this work, we follow the explicit Euler approach of Mitsos et al. [99]. The ODE system in (3.6) is discretized using 200 time points, and a convex relaxation of the objective function is obtained by propagating forward McCormick relaxations through the discrete-time states and comparing the resulting ODE trajectories with the experimental data.

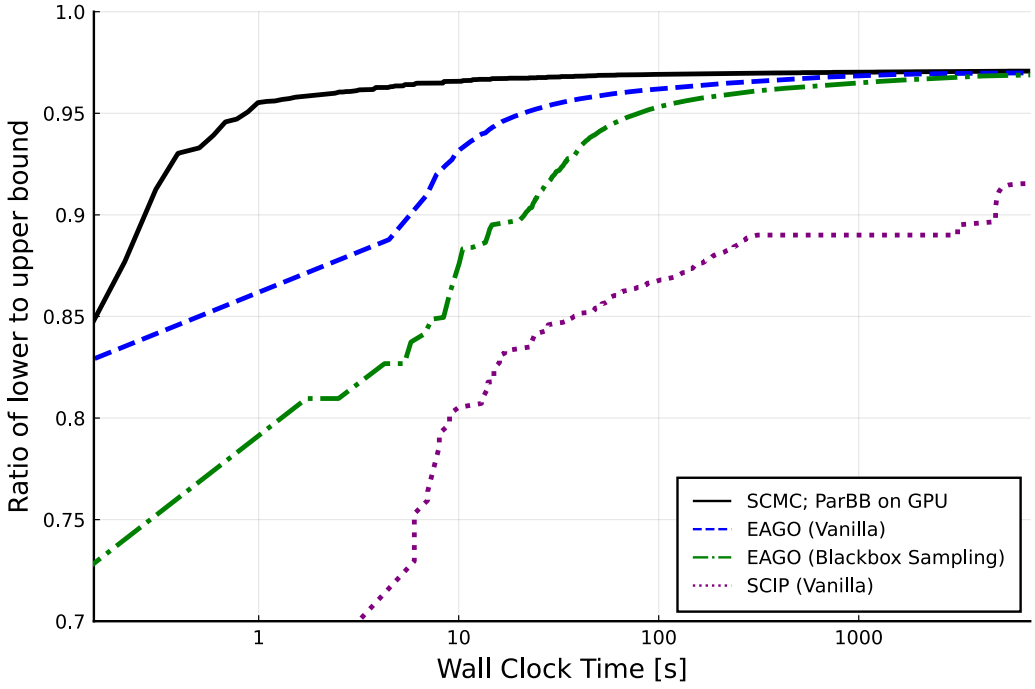

Figure 3.6: A convergence plot for the transient absorption kinetics problem (Sec. 3.4.3) showing the performance of the novel ParBB algorithm (solid black) against vanilla versions of EAGO (dashed blue) and SCIP (dotted purple) and vanilla EAGO using the same lower-bounding routine as ParBB (dash-dotted green). The ParBB algorithm shows the strongest performance out of any of the vanilla versions of more mature solvers, reaching 95% convergence in 0.88 s—roughly 24x faster than vanilla EAGO and roughly 90x faster than EAGO with the blackbox sampling method. At the two-hour time limit, EAGO with the blackbox sampling method reached a convergence of 96.9%, which was passed by ParBB in 61 s for a speedup of 118x. ParBB owes its performance to its ability to offload expensive objective function calculations to the GPU, as well as its ability to access a far greater number of nodes than traditional serial solvers.

Figure 3.6 shows a convergence plot that compares the performance results between the solvers in Table 3.4 on this problem. The most apparent aspect of this plot is that the ParBB algorithm converges faster than all other tested solvers. Although none of the solvers were able to converge fully within two hours due to the clustering problem, ParBB reached the highest level of convergence out of any solver, and reached 95% convergence in 0.88 s. This is 24x as fast as vanilla EAGO (21 s), and roughly 90x as fast as EAGO with the blackbox sampling method (79 s). Vanilla SCIP was unable to reach 95% convergence within the two hours of runtime. As another comparison point, EAGO with the blackbox sampling method eventually reached a convergence of 96.9% after two hours. This level of convergence was reached by ParBB in 61 s, giving a speedup of roughly 118x. This problem was too large to address using ANTIGONE or BARON, as the community license limited the number of nonlinear terms in the problem formulation.

This example represents a particularly strong use case for a parallelized B&B approach such as

ParBB because evaluating the objective function to obtain an upper bound effectively requires the calculation of a trajectory of the underlying ODE system, and obtaining a lower bound requires generating bounds on the set of all possible trajectories of the ODE system in a given parameter subdomain. These operations are computationally expensive, which makes them a good target for GPU parallelization. With ParBB, this parallelization is accomplished by grouping together calculations for different nodes into batches, and because ParBB uses the bounds on trajectories to calculate node lower bounds on the GPU, there is no overhead required to transfer calculation results between the GPU and CPU. This parallelized processing of nodes, made possible by `SCMC` and the ParBB algorithm, enable the parallelization of this and other expensive calculations that may be needed to solve complex deterministic global optimization problems.

3.5 Limitations

The new `SCMC` package, which can calculate pointwise evaluations of the interval extensions and convex and concave relaxations of math expressions in parallel on a GPU, has been shown to be fast and useful for global optimization when used in the correct conditions. However, it comes with several important limitations, some of which are briefly mentioned throughout this chapter and many of which are the targets of ongoing research efforts. The most impactful of these limitations, which will likely require significant effort to overcome, are summarized in the following.

- Due to the large number of floating-point operations required to calculate McCormick-based relaxations, it is highly recommended to use double-precision floating-point numbers, including numbers on GPUs. Since most GPUs are designed for single-precision floating-point operation, forcing double-precision will often result in a significant performance hit. GPUs designed for scientific computing, with a higher proportion of double-precision-capable cores, are recommended to obtain the best double-precision floating-point performance for use with `SCMC`.

- Due to the high branching factor for calculating McCormick-based relaxations and the possibility of warp divergence, there will likely be a performance gap between optimization problems with variables covering positive-only domains and variables with mixed domains. Additionally, more complicated expressions where the structure of a McCormick relaxation changes more frequently with respect to the bounds on its domain will likely perform worse than problems where the structure of the relaxation is more consistent. Some of this performance gap could be fixed by more intelligent ordering of B&B nodes prior to passing calculations to the GPU,

but more research is needed to investigate whether the sorting process would result in a net loss or gain of speed.

In addition to these more challenging obstacles, there are several more manageable limitations that we will seek to address in future work, summarized in the following.

- `SCMC` does yet not calculate subgradients of relaxations, which are used in many modern global optimizers. Although it would be possible to implement subgradient calculations in a similar fashion as described in this chapter, additional advances would be needed to make use of the subgradients in a performant way, such as by creating a GPU-based LP solver that can solve batches of LPs simultaneously. With developments such as this, including subgradients would allow for tighter relaxations for each node, which could increase the number of nodes fathomed in each iteration and thereby ease the memory burden of the parallelized B&B algorithm.

- Complicated expressions may cause significant compilation time if passed through `SCMC` at one time, which can currently be avoided by manually factoring expressions into a few intermediate expressions and combining the results together in a user-defined function. Automating this process with further source code generation capabilities will simplify the use and implementation of `SCMC`.

- Due to limitations in CUDA, functions created with `SCMC` may only accept 32 CUDA arrays as inputs. Thus, functions with more than 8 unique variables will need to be split/factored by the user, similar to how complex expressions are handled, in order to be accommodated. As with the expression complexity limitation, an automated factorization and function generation process would be able to overcome this hurdle by ensuring that the number of inputs for any given intermediate function does not exceed the 32-input limit.

- The current version of `SCMC` is compatible with the elementary arithmetic operations `+`, `-`, `*`, `/`, and the univariate intrinsic functions `^2` and `exp`. Although this library covers a very broad set of nonconvex optimization problems of interest to researchers and practitioners, it represents a very small fraction of the exhaustive library of `McCormick.jl` used by EAGO. Continued development to include additional functions will allow a more diverse set of functions to be used with the parallelized B&B algorithm.

While these listed issues will all positively impact the utility of `SCMC` for deterministic global optimization applications, they were not viewed as fundamental requirements to disseminate the novel methods and software. Instead, they function as near-term attainable targets for future research.

3.6 Conclusions and Future Prospects

A new method of calculating McCormick relaxations was developed that is built on a SCT approach inspired by forward-mode AD and implemented as the software package `SourceCodeMcCormick.jl` in the Julia language. This approach enables the parallel evaluation of an arbitrarily large number of interval bounds and convex/concave relaxation points, each on potentially different domains, on a GPU. A deterministic global optimization algorithm was created that makes use of this new approach to solve lower- and upper-bounding problems for a large number of B&B nodes simultaneously. The performance benefits of this new algorithm were demonstrated using GPU parallelization on three challenging nonconvex global optimization problems relevant to researchers and practitioners in the physical sciences and engineering. Future improvements to the `SourceCodeMcCormick.jl` package are underway, including the incorporation of subgradients, which promise to greatly enhance the utility of this approach within deterministic global optimization software.

Chapter 4

Automatic Generation of GPU Kernels for Evaluators of McCormick-Based Relaxations and Subgradients

This chapter builds on the work in the previous chapter to create a more effective method of evaluating McCormick relaxations on GPU hardware. Although the approach used in 3 demonstrated strong performance, it suffered from combinatorial explosion in terms of the number of conditional branches contained within each expression, which limited applicability. Further, it was missing the critical element of relaxation subgradients, which are important for creating the relaxed LPs described in Section 2.6.3. In this chapter, an entirely novel approach is used that resolves the combinatorial issue, allows for consistently faster relaxation evaluations, and incorporates subgradient calculations. As in Chapter 3, this chapter is motivated by the cost-effectiveness and energy efficiency of GPUs. At the time of writing this chapter, no existing mature solvers, to the best of our knowledge, used GPUs in any capacity. While Chapter 3 demonstrated that GPU-accelerated relaxations were potentially effective in practice, this chapter demonstrates that the generation of relaxations and relaxation subgradients can be efficient and practical for arbitrarily complicated factorable expressions. This work is accomplished using a source code generation approach for McCormick-based relaxation rules and is made available in the open-source software package `SourceCodeMcCormick.jl`. Across a

benchmark test set of problems from MINLPLib, this approach achieves speedups in the calculation of relaxations and their subgradients in the range of one-to-four orders-of-magnitude over a typical CPU-based implementation of these rules at a level of parallelization that would be useful for a GPU-based deterministic global optimizer. At the time of writing, this chapter is currently under review for publication and appears as reference [55].

4.1 Introduction

Deterministic global optimization (DGO) methods are required to solve nonconvex optimization problems to guaranteed ϵ-optimality. Typically, DGO relies on the B&B algorithm, but the worst-case exponential runtime of the algorithm limits the scale and scope of problems that can be addressed in a practical amount of time. One approach for addressing larger problem instances is to adapt the traditionally serial algorithm to operate on parallel computing architectures. When applying B&B to solve general mixed-integer programs (MIPs), for example, there have been successful adaptations using CPU-based multithreading [26] or distributed systems [97], as well as using massively parallel graphics processing units (GPUs) [23, 29], all of which have been covered by several reviews [49, 147]. The use of GPUs is of particular interest because of the fast performance that can be obtained from an efficient GPU-bound algorithm [135], as well as the greater energy efficiency of the hardware compared to CPUs [66, 100]. However, for spatial B&B, which is used to solve mixed-integer nonlinear programs (MINLPs), although mature solvers exist that can utilize multithreading [43, 61, 125] and distributed systems [124], the use of GPUs remains rare. To our knowledge, these efforts comprise interval arithmetic-based approaches by Kiel et al. [79], Eriksen and Rasmussen [42], and Zhang et al. [160], as well as the McCormick relaxation approach used by the present authors [58]. Although GPUs have demonstrated success in speeding up certain computations, it is nonintuitive how to best apply them to effectively speed up the spatial B&B algorithm.

Building an efficient GPU-accelerated DGO solver for MINLPs is challenging for several key reasons. First, the B&B tree search is problem-dependent and dynamic, which can lead to challenges with workload scheduling and balancing that can lead to poor performance [29, 147]. While this is an important aspect to consider for any parallelized solver using B&B, it is not a main focus of the present work. A second challenge, which this work seeks to address, is that the bounding operation itself is irregular and node dependent [30], especially when using relaxation techniques such as McCormick relaxations [99, 121] that require conditional checks and possibly calculation

loops [156]. This irregularity conflicts with the single-instruction multiple-thread (SIMT) execution model used by many GPUs [86] as it can lead to warp divergence and potentially poor performance as a consequence. The lower-bounding operation in B&B is of particular interest because, when applied to MIPs, nearly all of the B&B run time is spent on bounding [30, 147]. For MINLPs, which typically have a more complex relaxation procedure than MILPs, we expect this to be the case as well.

The standard lower-bounding operation in spatial B&B used by solvers such as BARON [76, 119, 161], MAiNGO [22], and EAGO [155], among others, is to use subgradients of convex relaxations to generate a linear program (LP) relaxation that is then solved to furnish a rigorous lower bound. This work addresses the first part of this bounding operation—specifically, we wish to calculate McCormick relaxations and their subgradients efficiently on GPU hardware. Our focus on relaxations and their subgradients is motivated by two considerations. First, the calculation of these objects is one of the two major steps in the standard lower-bounding method, thereby making it a critical component if the entire lower problem is sought to be parallelized. Second, preliminary profiling on our EAGO solver indicates that the calculation of relaxations and their subgradients makes up a nontrivial portion of the total runtime. Across a range of NLP benchmark problems, we observed that between 0.2–22.9% of the total time was spent performing these calculations. This value was generally higher for problems that took longer to solve, and problems that featured more complex expressions in the objective or constraints. These reasons suggest that accelerating the calculation of relaxations and their subgradients using GPU resources may lead to notable speedups for global solvers, especially on problems that take longer to solve. In this chapter, we will demonstrate that the calculation of relaxation and subgradient information through the generalized McCormick technique can be parallelized on GPU hardware to yield strong performance despite the presence of some warp divergence. This development is necessary for the development of a GPU-accelerated MINLP solver because, to the best of our knowledge, existing methods of calculating McCormick relaxations—besides our earlier work [58]—are not designed to efficiently exploit the SIMT execution of GPUs.

To effectively utilize GPU resources, this work develops a substantially different approach from other existing methods. The CPU-based `McCormick.jl` [153], for example, uses an operator overloading-like approach wherein basic math operations are effectively redefined to apply McCormick relaxation rules. Relaxations can then be computed for arbitrarily complex expressions through the automatic application of the relaxation rules for the necessary basic math operations composing the expression. An analogous GPU-based approach could be to launch a kernel (GPU subroutine) for each basic

math operation. While this operator overloading-like approach is flexible, having many such small kernels is inefficient because the volume of calculation in each kernel would be extremely small relative to the required volume of memory accesses (i.e., the kernels are memory-bound). In contrast, this work uses a source code generation approach wherein larger, unique kernels are created for each arbitrarily complex expression of interest. These larger kernels each encompass multiple McCormick-based rules and utilize thread-local memory resources for intermediate variable storage, which serve to increase the amount of calculation relative to the slower global memory accesses. To the best of our knowledge, there is one other existing source code transformation approach for McCormick relaxations by Corbett et al. [37], but the approach detailed in this chapter is explicitly designed to utilize GPU resources. The product of this chapter is, in effect, a library of GPU-compatible McCormick-based rules as well as a framework and tool that integrates them together using source code generation as needed for any expression of interest.

The developments in this chapter are available starting with version 0.5.0 of the open-source `SourceCodeMcCormick.jl` (`SCMC`) package [53]. While this work shares its goals with our previous chapter and earlier version(s) of the same software package [58], the approach taken here is entirely novel and distinct. Most notably, the methodology used in this chapter enables the use of basic programming constructs such as temporary variables and for-loops within relaxation rules, which are critical not only for describing complex rules, such as those for trigonometric functions, but also for more easily performing subgradient calculations by looping over subgradient elements.

For this chapter, background information is primarily contained within Chapter 2, Section 2.6. The remainder of this chapter is organized as follows. In Section 4.2, we will introduce some notational conventions, necessary definitions, and terminology that will be useful elsewhere in the chapter. Section 4.3 will contain descriptions of the implementation of new methods in software that enable GPU-accelerated relaxation and relaxation subgradient calculations. In Section 4.4, we will showcase the performance of this chapter's contributions in comparison with the CPU-based `McCormick.jl` [153] on a wide range of numerical examples from MINLPLib [1, 24]. Lastly, in Section 4.5, we will reflect on the current capabilities of this method compared to what is needed for a deterministic global optimizer and present our direction for future efforts.

4.2 Computational Background

The method development that follows in Section 4.3 relies on formal definitions of several computer science concepts. In particular, we wish to make clear the distinction in terminology between

standard software functions or *subroutines*, *symbolic functions*, *numeric functions*, and *kernels*, and describe how each of these may be interacted with by users and programs.

Definition 4.2.1 (Symbolic Variable)**.** A character or string of characters used to represent a mathematical object, without the assignment of a specific value, is referred to as a *symbolic variable*. The space of n distinct symbolic variables is given by $\mathbb{S}^n$.

Definition 4.2.2 (Symbolic Function)**.** An expression composed of numbers and symbolic variables that represents an exact computation is a *symbolic function*.

We introduce the notions of *symbolic variables* and *symbolic functions* to clarify our use of the `Symbolics.jl` [59] computer algebra system in the Julia programming language. Specifically, we make use of `Symbolics.jl` to store and operate on symbolic functions that represent the objective function and constraints of an optimization problem of interest. For example, by Definition 4.2.2, some objective function $f : \mathbb{R}^3 \to \mathbb{R}$ may be symbolically represented as the symbolic function $s : \mathbb{S}^3 \to \mathbb{S}$. For preciseness, we note that a mathematical *expression* is any configuration of mathematical symbols used to define a function. Within the context of this work, those functions must be consistent with Definition 2.6.6 (factorable functions).

Definition 4.2.3 (Subroutine)**.** A *subroutine* is a named block of code that contains software logic that can be called using a well-defined interface to perform a specific task.

Definition 4.2.4 (Numeric Function)**.** A *numeric function* is a subroutine that accepts numeric objects (i.e., collections of numbers) as inputs, performs specified numerical computations on its inputs and any other numerics stored internally, and returns one or more numeric objects as output.

The concept of a *subroutine* is raised to help distinguish between a function in a computer programming context versus a function in a mathematics context. Furthermore, this allows us to distinguish more precisely between *symbolic functions*, which are representations of mathematical expressions, and *numeric functions*. Although, generally, subroutines may be interfaced with in countless ways, we wish to call special attention to *numeric functions*, which apply numerical, algorithmic procedures to compute the values (i.e., outputs) of mathematical functions at given points (i.e., inputs), within machine precision. We note that subroutines are general and can be defined for both symbolic and numeric computation. Next, we further specialize and formalize the definition of a subroutine that is restricted to a specific computing architecture.

Definition 4.2.5 (Kernel)**.** A *kernel* is a subroutine that is only capable of running on graphics processing unit (GPU) computer hardware, with the instruction set only available in GPU memory.

In other words, for the purposes of this work, *kernels* are GPU-specific subroutines. In this work, we will only consider kernels that operate on numeric objects, and so we will only consider kernels to be numeric functions. Additionally, the `SourceCodeMcCormick.jl` package was developed to make use of the CUDA toolkit [107] through the `CUDA.jl` package [16] to develop NVIDIA-GPU-specific kernels. As such, we equivalently refer to kernels in this work as *CUDA kernels*.

In addition to having their instruction sets available in GPU memory, kernels are unique in their ability to operate on data stored in GPU memory. For example, an array of data may be stored in CPU memory in the form of an `Array`. To be made accessible to a kernel, this array must first be sent to GPU memory, which is accomplished with `CUDA.jl` by converting the `Array` to a `CuArray` [16]. A `CuArray` is further defined by the type of data it holds, such as a `CuArray{Float64}`, which holds double-precision floating-point values. The *size* of an `Array` or `CuArray` will be denoted as (m, n), which represents a matrix object with m rows and n columns. A program or a user can then instruct the GPU of a computer system to execute a kernel that will interact with this `CuArray`, thereby taking full advantage of the parallelization capabilities of the GPU hardware. In this chapter, the distribution of work to parallel hardware is accomplished by taking the task of calculating many McCormick-based relaxations at once and, effectively, assigning one relaxation to each GPU thread.

4.3 Method Development

In this section, we detail the developments that enable parallel relaxation and relaxation subgradient computation on GPUs. The main feature of spatial B&B that we exploit to enable this parallelized evaluation is that each node in the B&B tree is addressing fundamentally the same objective function and constraints as the original global optimization problem, with generally only the (sub)domain changing. I.e., the lower-bounding process for every node will require calculating relaxations and relaxation subgradients for a problem-specific set of expressions that are necessarily known prior to starting the B&B algorithm. `SCMC` relies on the fact that repeated relaxation evaluations are needed for a specific set of expressions to create and compile kernels, unique to those expressions, that return inclusion monotonic interval extensions and McCormick-based relaxations and their subgradients of those expressions on any domain of interest. Note that because kernels are intended to be created prior to starting the B&B algorithm based on a fixed set of expressions, this approach is less effective in cases where the set of problem expressions changes during the processing of the B&B tree. Once these kernels are compiled, they can be used throughout the B&B process to quickly obtain many interval, relaxation, and subgradient evaluations, for one node or multiple nodes, using

GPU parallelization.

The main contribution of this work is the process by which `SCMC` automatically creates kernels representing McCormick relaxations, which is described in the remainder of this section. This functionality is accessed using the `kgen` (**k**ernel **gen**erator) subroutine, which is the primary user-facing tool and the core, fundamental backbone of `SCMC`. Practically speaking, `kgen` accepts a symbolic function $s : \mathbb{S}^n \to \mathbb{S}$ as input, and it outputs a kernel that performs the mapping $\mathbf{f}_s : \mathbb{R}^{3n} \to \mathbb{MR} \times \mathbb{R}^{2n}$, where the $\mathbb{R}^{2n}$ component of the codomain holds the n-dimensional subgradients associated with the convex and concave relaxations of s. From the discussion following Definition 2.6.14, the domain of $\mathbf{f}_s$ is $\mathbb{R}^{3n}$ because, for each of the n components of the input variable vector of $s(\cdot)$, three values in total are required to define a McCormick object for that variable. The kernel that computes this mapping is created by `kgen` through an automatic source code generation process, described in Sections 4.3.2-4.3.4, that is effectively a transcription of the McCormick-based rules associated with factors of s placed in a new `.jl` file.

The main steps of `kgen` are illustrated in Figure 4.1. The remainder of this section is dedicated to explaining the individual components of `kgen`. In Section 4.3.1, the inputs and outputs of `kgen` will be discussed in detail. Sections 4.3.2-4.3.4 will then describe the process that `kgen` uses to generate its numeric kernel output.

4.3.1 Inputs and Outputs

The `kgen` subroutine is designed to be called at the initialization step of a DGO solver, prior to any iterations of B&B. At this stage, the objective and constraints of the global optimization problem are known, and the goal of `kgen` is to create kernels that can be called during the main steps of the B&B algorithm to calculate inclusion monotonic interval extensions, relaxations, and relaxation subgradients of these constraints and objective. Specifically, the `kgen` subroutine expects its first argument to be a symbolic function composed using `Symbolics.jl`-type [59] symbolic variables. [1] If relaxations are desired for an isolated expression, the symbolic function is the only required input to `kgen`. For more typical cases, where relaxation evaluations will be required for, e.g., multiple constraints in an NLP, `kgen` also optionally accepts information about all decision variables in the problem in the form of a vector of symbolic variables. When this information is provided, `kgen` will expand the dimensionality of the relaxation subgradients to account for all decision variables, rather

[1] The choice to use `Symbolics.jl` is based on legacy requirements documented in Gottlieb et al. [58] and is not of fundamental importance to the methods used in `SCMC`. Future work includes adapting to the expression format used by `MathOptInterface` [83], which is the backend for the JuMP algebraic modeling language [41]: the primary interface for Julia users looking to model/formulate and solve optimization problems.

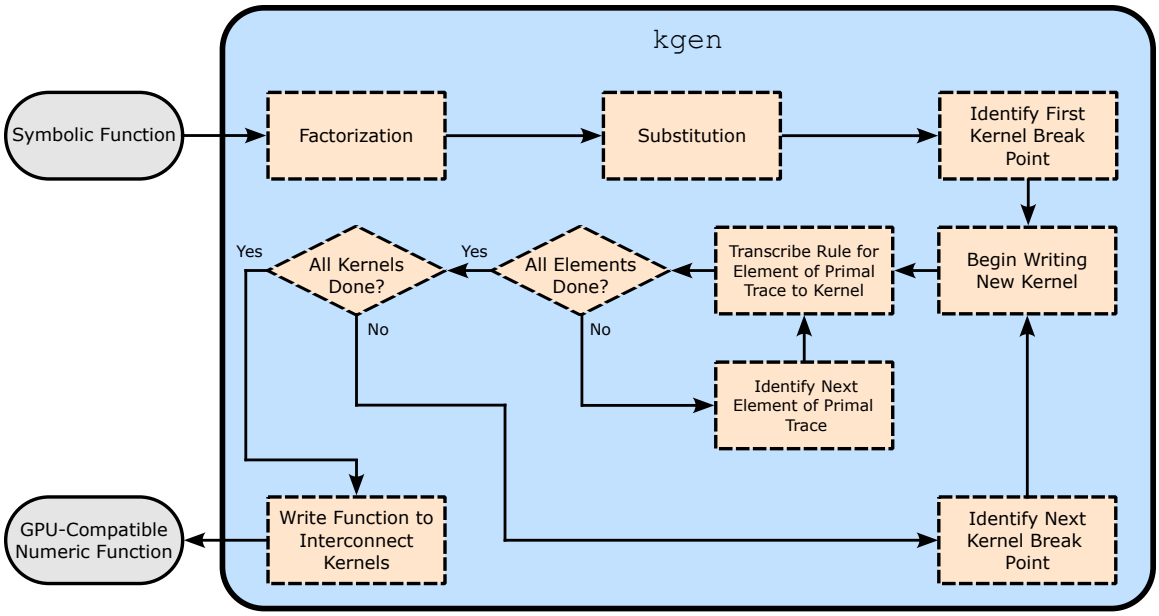

Figure 4.1: A flowchart of the main processes taking place in a call to the `kgen` subroutine. Input functions are initially factored to produce a primal trace of the expression, followed by an expression detection and substitution step. The reformulated primal trace is then parsed sequentially to automatically generate CUDA kernels that represent the original—or portions of the original—expression. A single function is returned as `kgen` output that automatically calls the generated kernel(s) in the proper order and with the proper inputs, and modifies the first input to hold relaxations, inclusion monotonic interval extensions, and convex and concave relaxation subgradients of the input expression on any provided domain.

than only those participating in the expression, and will ensure that the subgradient dimensions associated with each variable are consistent across each call to `kgen`.

The output of `kgen` is a single CPU subroutine `new_func` that acts as a wrapper for the kernel(s) corresponding to the original input symbolic function. For a symbolic function input $s : \mathbb{S}^n \to \mathbb{S}$, the newly created `new_func` accepts $n+1$ inputs and always returns no outputs; the first argument is mutated to store the output information, and the remaining arguments correspond to individual symbolic variables in alphabetical order. In practical terms, each input argument of `new_func` is required to be a 2D `CuArray{Float64}`. For the arguments associated with problem variables, numbered 2 through $n+1$, the `CuArray{Float64}` objects must be of size $(m, 3)$. Specifically, each of the m rows holds a value of a variable where convex and concave relaxations and their subgradients are requested, as well as the variable's interval bounds on a domain; the first column stores the value of this variable, and the second and third columns correspond to the lower and upper bounds of the domain of this variable, respectively. This is the minimum amount of information required to apply McCormick-based rules (see discussion following Definition 2.6.14). The first argument, which stores

Listing 4.1: A typical use-case of `kgen` is demonstrated. Once symbolic variables are created, `kgen` is called to create a numeric function as output. This numeric function accepts `CuArray{Float64}` objects representing the output and input variables and calls one or more source-code-generated kernels to compute interval extensions, relaxations, and relaxation subgradients at the points and on the sets defined by the input variables. The identities of the columns in the input and output `CuArray`s are described in Figure 4.2. In this case, `my_func` is used to return evaluations at the point $(1.5, 3.0) \in [1.0, 2.0] \times [2.0, 4.0]$.

```
using SourceCodeMcCormick, Symbolics

# Define symbolic variables
Symbolics.@variables x, y

# Call kgen to generate a numeric function "my_func"
my_func = kgen(1/(1+exp(x*y)))

# Create CuArray objects holding (point, lower_bound,
# upper_bound) for x and y, as well as a pre-allocated
# output CuArray
x_in = CuArray([1.5 1.0 2.0;])
y_in = CuArray([3.0 2.0 4.0;])
OUT = CUDA.zeros(Float64, 1, 4 + 2*(2))

# Call the numeric function
my_func(OUT, x_in, y_in)

# Convert the output to an Array for visualization,
# and show the result
display(Array(OUT))
# --> Result is:
#     1x8 Matrix{Float64}:
#   0.00669285 0.0795804  0.00033535  0.119203 \\
#  -0.0265922  -0.00664806  -0.0396225  -0.0198113
```

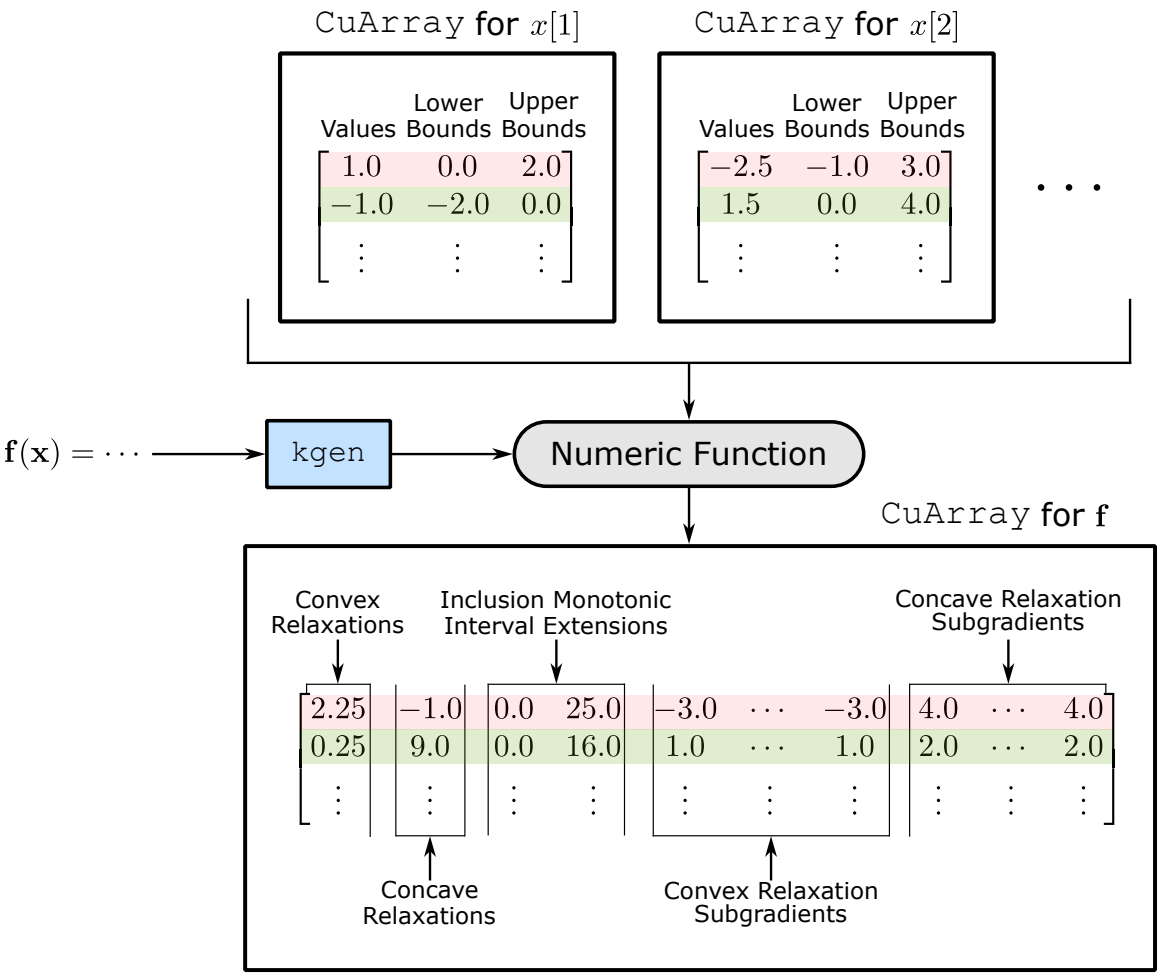

Figure 4.2: Visualization of the inputs and outputs of custom-generated kernels. In this example, $x[1]$, $x[2]$, and $\mathbf{f}(\mathbf{x})$ are represented in code as `CuArray{Float64}` objects. All calculations in the generated `new_func` are performed in custom-generated kernels which operate on the GPU data given as inputs. All calculations are row-independent, such that, for example, the numbers highlighted in red (green) in $\mathbf{f}$ correspond to the inputs highlighted in red (green).

the output, must be of size $(m, 4 + 2n)$, and will be mutated to hold in its columns, respectively, a convex relaxation, a concave relaxation, lower and upper bounds of an inclusion monotonic interval extension, a convex relaxation subgradient, and a concave relaxation subgradient. Each row of this output corresponds to pointwise evaluations of the relaxations, bounds, and subgradients calculated for the input symbolic function of `kgen`, evaluated at the points and for the sets defined in the corresponding rows of the other inputs. Listing 4.1 shows a code demonstration of how `kgen` can be used and Figure 4.2 presents a visualization of the inputs and outputs, with the "output" argument represented pictorially as leaving `new_func` to clarify the flow of information.

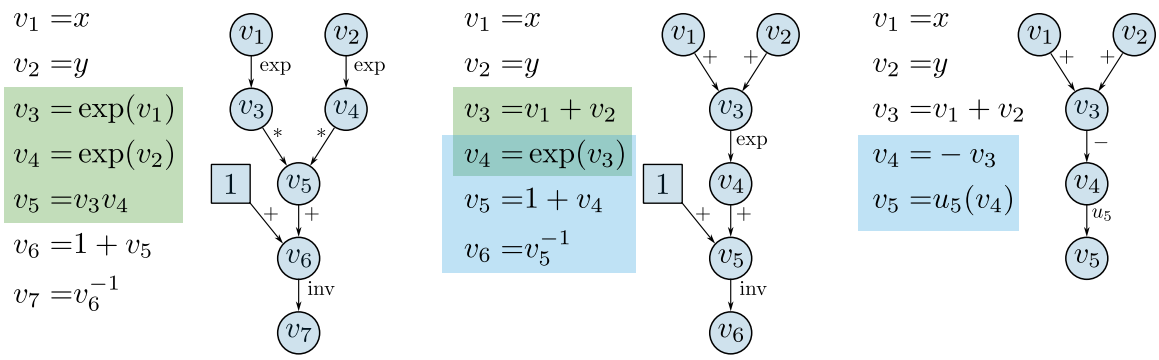

Figure 4.3: Factorization and expression detection of the function $f(x, y) = 1/(1 + \exp(x)\exp(y))$. The function is first decomposed into its primal trace and computational graph (left). Next, subexpressions that match specified forms are identified and substituted with equivalent forms, such as the algebraic rearrangement $\exp(x)\exp(y) = \exp(x + y)$ (green, center) and the substitution of a subexpression with a specialized function with known convex and concave envelopes: $u_5(x) = \text{sigmoid}(x) = 1/(1 + \exp(-x))$ (blue, right).

4.3.2 Factorization and Substitution

As a requirement for applying Definition 2.6.14 (i.e., the McCormick relaxation procedure), `kgen` assumes that all functions of interest are factorable and continuous. We note that the continuity assumption comes from the original result by McCormick [95], which can be replaced by the assumption that all functions are bounded [149]. To begin the process of creating a kernel, given a symbolic function input, `kgen` first factors the symbolic function into binary addition and multiplication and univariate terms to generate a primal trace and computational graph of the expression. Following the convention of Gottlieb et al. [58], elements of the primal trace of the expression are represented as nodes in the graph, which are joined by labeled arcs representing binary addition or multiplication, or which are individually operated on by a univariate intrinsic function. The factorization and generation of a computational graph is visualized in Figure 4.3 (left). This process is analogous to the auxiliary variable method in that each node can be thought of as an auxiliary variable representing an intermediate term in the calculation of the original expression [58].

With the input expression fully factored, `kgen` performs a substitution step in which it parses the factorization in search of several specified forms that are known to have an equivalent algebraic rearrangement that results in tighter relaxations. We note that these substitutions may not typically be performed by a compiler, as the results of these substitutions may not be identical when applying standard IEEE-754 floating-point arithmetic [36]. However, they are enabled in `SCMC` by default due to the substantial improvement in the tightness of relaxations that can be obtained. This result is related to the well-known dependency problem of interval arithmetic [69]. If the substitutions

described in this section are not desired, they can be disabled by setting `substitute=false` in the call to `kgen`. The current algebraic rearrangements utilized by `kgen` are the same as those used by EAGO [155]:

$$\exp(x)\exp(y) \Rightarrow \exp(x+y) \tag{4.1}$$

$$\log(x^y) \Rightarrow y\log(x) \tag{4.2}$$

$$(x^a)^b \Rightarrow x^{(ab)} \tag{4.3}$$

$$a^{\log(x)} \Rightarrow x^{\log(a)} \tag{4.4}$$

$$\log(xy) \Rightarrow \log(x) + \log(y) \tag{4.5}$$

$$\log(1/x) \Rightarrow -\log(x) \tag{4.6}$$

Additionally, `kgen` automatically detects and substitutes some subexpressions with their equivalent specialized forms if they have known convex and concave envelopes. One example is the sigmoid function given by:

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}. \tag{4.7}$$

As described by Wilhelm et al. [156], McCormick-based relaxations for this function are unnecessarily conservative relative to the known convex and concave envelopes of this function. In the `McCormick.jl` library [153], the envelope for the sigmoid function is invoked when the user defines their expression using the `sigmoid` subroutine. With `kgen`, the expression can be written in the expanded right-hand-side form of (4.7) without the user intervention of invoking the named subroutine: `kgen` automatically detects this special structure within the factorization—including in cases where, for example, the argument is an auxiliary variable—and replaces elements of the primal trace to use the internal `SCMC_sigmoid` subroutine rather than the full algebraic expression of (4.7). An example of this detection can be seen in Figure 4.3, where `kgen` recognizes the presence of the sigmoid form (blue, center), if an additional auxiliary variable is added to negate the argument of exp (auxiliary variable v_4, right). This detection and replacement approach is a feature of `kgen` due to its representation of the expression as a primal trace. Operator overloading-like approaches such as `McCormick.jl` are unable to detect these structures because each math operation inherently lacks the context of the greater math expression. Additional unique subexpression forms, including those for domain-specific applications, may be easily added to the expression library if provided with an appropriate method for calculating envelopes or tighter relaxations. In principle, this library could also be extended to include multivariate and parametric functions, such as multivariate McCormick

relaxations [141], which we leave as an area of future work, if an appropriate method for calculating relaxations is provided and the factorization process is adjusted to recognize the new functions. Extensions could similarly be made to include operators with auxiliary arguments, which are not currently implemented. This type of extension is recommended in general, as specialized relaxations for nontrivial subexpressions can often lead to tighter relaxations.

4.3.3 Kernel Writing

Following the substitution step, `kgen` proceeds by converting the updated factorization into one or more CUDA kernels, as described in this section. Multiple kernels may be created for an individual math expression if `kgen` determines that the code length of the desired kernel—in terms of the number of lines of written code—would exceed a predetermined threshold. Functionally, splitting occurs by creating a partition of the computational graph, storing intermediate results as temporary variables, and passing those temporary variables as inputs to subsequent kernels. This process of storing an intermediate result and passing it onward to later calculations is valid through the notion of cumulative mappings in Definition 2.6.7 and the generalized structure of Definition 2.6.14 (Generalized McCormick Relaxations) [121]. A visualization of how this splitting may occur can be seen in Figure 4.4.

Splitting an expression so that it is addressed by multiple kernels has implications for the compilation time—the time before the kernels are usable in code after being created by `kgen`—and the overall performance of the generated function. As an expression is split into an increasing number of separate kernels, the overall compilation time decreases because the kernels are simpler to process, but the performance of the generated function also decreases for two main reasons. First, adding kernels may require the use of more temporary variables in global GPU memory to hold intermediate results, the allocations for which will cost time and memory space. `kgen` is able to reuse temporary variables that hold stale data, but this reuse is not always possible, depending on the structure of the original expression. Second, there is a small overhead cost to calling kernels that arises due to the need to allocate CPU memory for kernel parameters. Increasing the number of calls to separate kernels increases this overhead cost proportional to the number of kernels being called. Due to this competition between compilation times and generated function speed, `kgen` includes an adjustable `splitting` parameter that changes the threshold number of lines for kernel splitting, with full details provided in the Supplementary Information of reference [55].

Once a partition of the computational graph is determined based on the splitting policy, `kgen`

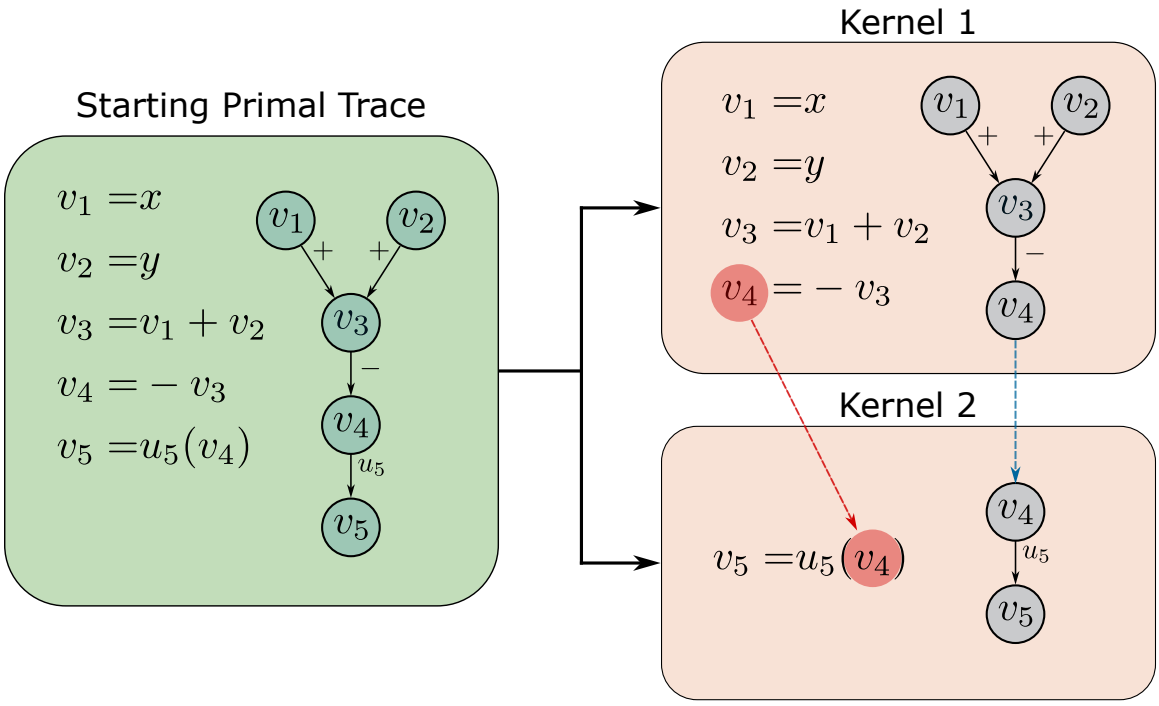

Figure 4.4: If `kgen` detects that a primal trace and its associated computational graph (left) would result in a generated kernel that is too long, it may split the overall computation into two or more kernels (right) by creating a partition of the original computational graph. Intermediate results from earlier kernels, such as v_4 in Kernel 1 (top right), are then stored as intermediate variables and may be utilized as inputs to later kernels, such as Kernel 2 (bottom right).

passes elements of the partition—effectively, portions of the original primal trace—to the internal `create_kernel!` subroutine. This subroutine then proceeds to directly compose a kernel into a newly generated `.jl` file. The kernel being written begins with a generic initialization, which includes setting up an appropriate number of temporary variables that are generally stored during calculation in registers (quick-access memory accessible to individual threads), subject to eventual compiler decisions. Next, `create_kernel!` scans through its input primal trace topologically. For each element of the trace, this subroutine identifies the math operation being performed and appends the corresponding McCormick-based rule to the end of the kernel. The operators for which rules are currently implemented include: $\{+, -, *, /, \hat{}\,, \text{sqrt}, \exp, \log, \text{abs}, \cos, \sin, \text{sigmoid}\}$.

As a more literal description, `create_kernel!` generates a kernel by combining together pre-written blocks of code that apply the correct set of rules in a correct, topological order. In effect, `SCMC` can be viewed as a library of GPU-compatible McCormick-based rules as well as the tool that integrates these rules together. Each rule in this library is written as a code block that applies Definitions 2.6.14–2.6.17 for a given operation using typical Julia syntax, including constructs such

as for-loops and temporary variable storage, where appropriate. The rules themselves are derived from the `McCormick.jl` library [153], but keeping them as code blocks enables greater flexibility. For example, `SCMC` maintains information about the input(s) and output of each rule in the context of a given expression, and by storing the rules as code blocks, `kgen` is able to interpolate specific variable references into the text of each rule as it integrates them together in sequence. Knowing the context of the expression also enables several code optimizations to be made that improve performance, such as the elimination of some conditional trees, as detailed in Section 4.3.4.

One downside to this code-block-based approach is that, because the kernels are written to a file and compiled for GPU use, it is not trivial to make use of existing Julia packages that are designed for CPU use. Specifically, for its calculation of inclusion monotonic interval extensions, `McCormick.jl` makes use of the `IntervalArithmetic.jl` package [120], which ensures that intervals are correctly rounded according to the IEEE 1788-2015 standard for interval arithmetic [35]. For `SCMC`, although kernel programming using standard Julia syntax is made possible through the `CUDA.jl` package [16], to our knowledge, specifying rounding modes on individual math operations during GPU operation is not natively supported. Consequently, all math operations in `SCMC` currently use the default rounding mode of the GPU. Although the default within `SCMC` is to use double-precision floating-point numbers, this may lead to the accumulation of small errors over many calculations. Modifications to the code-block library to enforce correct rounding of interval and relaxation calculations is left as future work, and may require substantial modification to the code blocks themselves, but these modifications will likely not require material changes to the underlying `SCMC` approach.

Once `create_kernel!` has finished appending code for each element of its input primal trace, it concludes the kernel generically by assigning the results to the memory address named by the first kernel argument and writing `end` to finalize the subroutine definition. If `kgen` had identified that multiple kernels were needed, it will continue calling `create_kernel!` to append new kernels representing other elements of the partition of the input expression into the same file as the first. Following this, `kgen` creates a final subroutine that integrates all generated kernels together. This subroutine accepts the $n+1$ inputs described in Section 4.3.1, creates temporary variables as needed to pass information between kernels, calls the generated kernels in order, and stores the final result in its (mutating) first argument. This subroutine and the generated kernels are then compiled and ready to be used. Ultimately, this final subroutine, which connects the generated kernels together internally, is the output of `kgen`.

4.3.4 Supporting Code Optimizations

One distinguishing characteristic of the rules implemented in `SCMC` versus those in `McCormick.jl` is that in `SCMC`, many of the smaller helper subroutines are inlined (i.e., the body of the would-be subroutines are written instead of subroutine calls), and they are occasionally integrated together with other portions of code. This inlining provides opportunities for code optimizations that are not possible with the generically written rules in `McCormick.jl`. In particular, `SCMC` exploits information about participating variables and the static nature of the final, composed rules to incorporate simplifications that could not be made dynamically. In this subsection, several such supporting code optimizations are described that are used by `SCMC` to improve code execution performance.

The simplest form of supporting optimization occurs when multiplying a variable by a constant. In `McCormick.jl`, the rule involves a check for the sign of the constant. `SCMC` can eliminate this check, since the value of the constant is known at the time of kernel writing, and instead only writes the correct version of the rule. While this optimization is simple, it concisely demonstrates the type of improvements that can be made to McCormick-based rules if the expressions are known ahead of time, which is the key benefit of this novel source code generation approach over multiple dispatch- or operator overloading-based schemes.

As a less trivial supporting optimization, Step 1 of Definition 2.6.14 allows Generalized McCormick relaxations to work with arbitrary relaxation and interval inputs. However, as described in the text following Definition 2.6.14, when applying the McCormick relaxation procedure to a function $\mathscr{F}$, the McCormick objects $\mathbf{y}^\circ$ used as inputs only contain three unique values. Additionally, the numerical values of subgradients associated with the relaxations of any y_i°, $i = 1, \ldots, n_p$ can be initialized to a value of one in the i-th dimension and zero in all other dimensions. One result of this simplicity is that the only input information required is the domain of each variable and the point at which an evaluation is requested, as described in Section 4.3.1. This contrasts `McCormick.jl`, which is built for generality and, therefore, requires explicit relaxation, interval, and subgradient information for every input of an expression.

At the kernel level, assumptions about $\mathbf{y}^\circ$ are directly used to improve performance through the use of custom simplified rules. For example, applying Step 3c of Definition 2.6.14 for a univariate function u_k applied to one of the inputs v_i°, $i = 1, \ldots, n_p$ requires the use of the `mid` operator (Definition 2.6.12) to determine the median value of the convex and concave relaxations of the input v_i and the minimum (maximum) of u_k on the domain of the input V_i. Without simplifying assumptions about $\mathbf{y}^\circ$, calculating `mid` requires a tree of value comparisons, the output of which plays

a role in the final convex (concave) relaxation value and a subgradient of that relaxation. However, with the *a priori* knowledge that the convex and concave relaxation values of v_i°, $i = 1, \ldots, n_p$ are the same, the tree of comparisons can be entirely removed as the result is always known.

Additionally, for the McCormick-based rules applied in general, calculations are required for every dimension of the convex and concave relaxation subgradients. In `SCMC`, information about which variables participate in each element of the primal trace (i.e., the *sparsity*) is calculated prior to writing kernels. This sparsity information is directly incorporated into written kernels to (without calculation) evaluate elements of the subgradient as zero in every dimension not associated with a participating variable. Furthermore, in cases where a variable $\mathbf{y}^\circ$ is used directly, `SCMC` applies simplified math to calculate the subgradient in the dimension of the variable since the initialized value is known to be one. These algorithmic simplifications are not possible using the generalist approach of `McCormick.jl` as there is currently no way to propagate information about participating variables, nor is it possible to apply unique rules to different elements of a subgradient.

In cases where `SCMC` does not make simplifications based on values known *a priori*, it is still able to implement some evaluation shortcuts due to the extensive inlining of rules. Specifically, rather than store the result of the `mid` operator as a temporary variable to use later, in some cases, it is beneficial to write the entirety of the resulting rule at every branch of the tree created for `mid`. Within each branch, instead of using a temporary variable to hold the median value, the identity of the median value is known and may be interpolated directly into the math. Using the rule for $\exp(x)$ as an example, the concave relaxation in `McCormick.jl` [153] is given by the following expression:

$$(\exp(x))^{cc} = \frac{\exp(x^L)(x^U - \text{midcc}) + \exp(x^U)(\text{midcc} - x^L)}{(x^U - x^L)}, \tag{4.8}$$

where midcc is $\text{mid}\{x^{cc}, x^{cv}, x^U\}$. By writing this rule in a branch of `mid` where midcc is evaluated as x^U, the simplification

$$(\exp(x))^{cc} = \exp(x^U)$$

can be made. In `SCMC`, this simplified rule is written to the kernel in place of (4.8) for the affected branches. Similarly, in branches where midcc (midcv) is evaluated as x^U (x^L), the math associated with the relaxation subgradient can be eliminated as the subgradient becomes zero in every dimension (Definition 2.6.17). The removal of this extraneous math—and the elimination of the need to store midcc or midcv temporarily—results in a modest performance boost, at the cost of a longer kernel. Creating longer kernels increases the compilation time, sometimes significantly, and it is therefore

evaluated on a case-by-case basis whether to make this optimization.

For a final form of supporting code optimization, it is worth noting that the EAGO solver has special handling for quadratic expressions. Specifically, if EAGO detects that a constraint is quadratic, it will not rely on McCormick relaxation rules, but will instead use the comparatively simpler and tighter approach given by Vigerske and Gleixner [145]. In addition to not being McCormick-style relaxations, this alternative approach is not possible using `McCormick.jl` because it lacks the context to know whether an expression overall is quadratic. In contrast, `SCMC` works by creating a computational graph of expressions of interest and therefore is able detect specific expression types. For the case of quadratic expressions, this context enables `SCMC` to construct a kernel that applies the tighter technique of Vigerske and Gleixner [145] to calculate relaxations and subgradients instead of the typical McCormick-based rules. If more traditional McCormick-style rules are desired, this functionality can be disabled by setting `affine_quadratic=false` when calling `kgen`.

4.4 Benchmarks

`SCMC` is designed to exploit the parallel processing capabilities of GPUs to simultaneously calculate relaxations, inclusion monotonic interval extensions, and relaxation subgradients at multiple evaluation points, potentially on different sets. Accordingly, benchmarks in this section are designed to reflect what would likely be a typical use case of this package in a parallelized B&B algorithm. All numerical experiments were run on a single thread of an Intel Xeon W-2195 2.30/4.30 GHz (base/turbo) processor with 64 GB of RAM running a Windows 11 Enterprise 22H2 operating system and an NVIDIA Quadro GV100 GPU with 32 GB of HBM2 VRAM (driver v32.0.15.7656, CUDA v12.9.86). A single thread was used for the CPU for simplicity of comparisons and to enable comparisons against `McCormick.jl` as it is used in our solver EAGO, which does not currently exploit multithreading. Results may be extrapolated to multithreaded implementations under the assumption of linear scaling with the number of CPU threads available. Julia v1.12.5 was used in conjunction with `MKL.jl` v0.9.1 to ensure use of the Intel MKL library, `BenchmarkTools.jl` v1.6.3 [33], `CUDA.jl` v5.11.0 [16], `McCormick.jl` v0.14.0 [153], and `SourceCodeMcCormick.jl` v0.5.2.

Benchmarking was performed using a set of problems from MINLPLib [1] satisfying the following criteria:

1. The problem type was listed as “NLP”;

2. The number of variables did not exceed 100;

3. The operators used in the NLP did not include any of the following: {tanh, log10, gamma, mod, cvpower, errorf, signpower, centropy, rpower}, where:

$$\text{errorf}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-t^2}{2}} dt; \tag{4.9}$$

$$\text{signpower}(x, y) = \text{sign}(x)|x|^y, \text{ for } y > 0; \tag{4.10}$$

$$\text{centropy}(x, y[, z]) = \begin{cases} x \cdot \ln\left(\frac{x+z}{y+z}\right) & \text{if } x > 0,\ y > 0,\ z \geq 0 \\ 0 & \text{if } x = 0,\ y > 0,\ z \geq 0 \end{cases} \tag{4.11}$$

$$\text{rpower}(x, y) = \begin{cases} x^y & \text{if } x > 0 \\ 0 & \text{if } x = 0,\ y > 0 \\ 1 & \text{if } x = 0,\ y = 0. \end{cases} \tag{4.12}$$

For each of the remaining 148 problems, the benchmarking process entailed partitioning the initial domain of the problem into 20,480 elements, with the number selected to match the parallelization capacity of the GV100 GPU being used by `SCMC`. For problems with unbounded variables, the domains were artificially limited to $\pm 10^6$ prior to partitioning. For each partition element, a point was selected from a uniform distribution within the subdomain to act as an evaluation point, overall yielding 20,480 unique evaluation points and domains, all stored using double-precision floating-point values. For `SCMC`, the `kgen` function was called on the objective function and each constraint of a given NLP, with the sum of kernel generation and compilation times across all expressions being recorded as the compilation time. Each of the `kgen`-generated kernels was then tasked with evaluating all 20,480 evaluation points for each expression, with the total time recorded as the runtime. For `McCormick.jl`, a function was created for each problem expression that accepts participating `MC` objects as inputs and outputs the result of the expression. The sum of the first-call times for each such function for every expression in a given problem is recorded as the compilation time, and the total time required to apply each function for a problem to the 20,480 evaluation points is recorded as the runtime. To determine the evaluation runtimes of `SCMC` and `McCormick.jl` we utilized the `@belapsed` macro from the `BenchmarkTools.jl` library, which evaluates the timing for a large number of samples before returning the minimum result.

In terms of data transfer, `SCMC` requires all 20,480 sets of evaluation points and domain bounds to be transferred from the CPU to the GPU, each of which includes a variable value, a lower bound, and an upper bound for each dimension. For example, a ten-dimensional example requires

transferring a total of 614,400 double-precision floating-point numbers, which takes roughly 460 μs on our system. This time scales linearly with the dimensionality of the problem, though we note that the information can be reused for the objective function and each constraint function within a given NLP and therefore only needs to be transferred once per set of evaluation points. For the final calculation results, we transfer $4 + 2n$ double-precision floating-point numbers per unique evaluation point and domain from the GPU back to the CPU, for each constraint function and the objective function. For example, for the objective function of a ten-dimensional example, a total of 491,520 double-precision floating-point numbers will be transferred, which takes roughly 410 μs on our system. We note that these transfers do not happen automatically within `SCMC`, which allows opportunities for an end-to-end solver to avoid unnecessary data transfer. For example, evaluation points may be generated on the GPU, thereby only requiring the lower and upper bounds to be moved to the GPU initially, and if later steps in a lower-bounding routine also occur on the GPU, significant portions of the data transfer off the GPU can be avoided or postponed.

It is important to note that although this benchmark method is being applied to NLPs, the problems are not being solved using a spatial B&B algorithm. Rather, since this novel source-code generation approach was created with the intention of using it within a B&B framework, we aimed to evaluate its performance in computing the necessary relaxation and subgradient information across elements of partitions of original problem domains. This is generically similar to what a DGO solver would do while solving a nonconvex NLP, though the partition created in B&B is typically the result of a tree search. As such, we make no claims that the specific partition elements we select in our benchmark would necessarily be reached during B&B, if it were applied to these problems. Additionally, we do not claim that the partitioning strategy used within these benchmarks is the only or best way to use `SCMC`, and we relegate the task of determining how best to apply this tool within an accelerated DGO solver to our future work.

4.4.1 Compilation Time

A comparison of compilation times between `SCMC` and `McCormick.jl` is shown in Figure 4.5, where the location of points within the plot corresponds to the NLP problem size, and the color of the points indicates the relative speed of the two packages. As can be observed in the plot, the compilation time of `SCMC` was always greater than the compilation time of `McCormick.jl`, with a minimum compilation time ratio of 2.38x. Nonetheless, the compilation time was within an order of magnitude in more than half of test cases; in general, we observed a strong relationship between expression

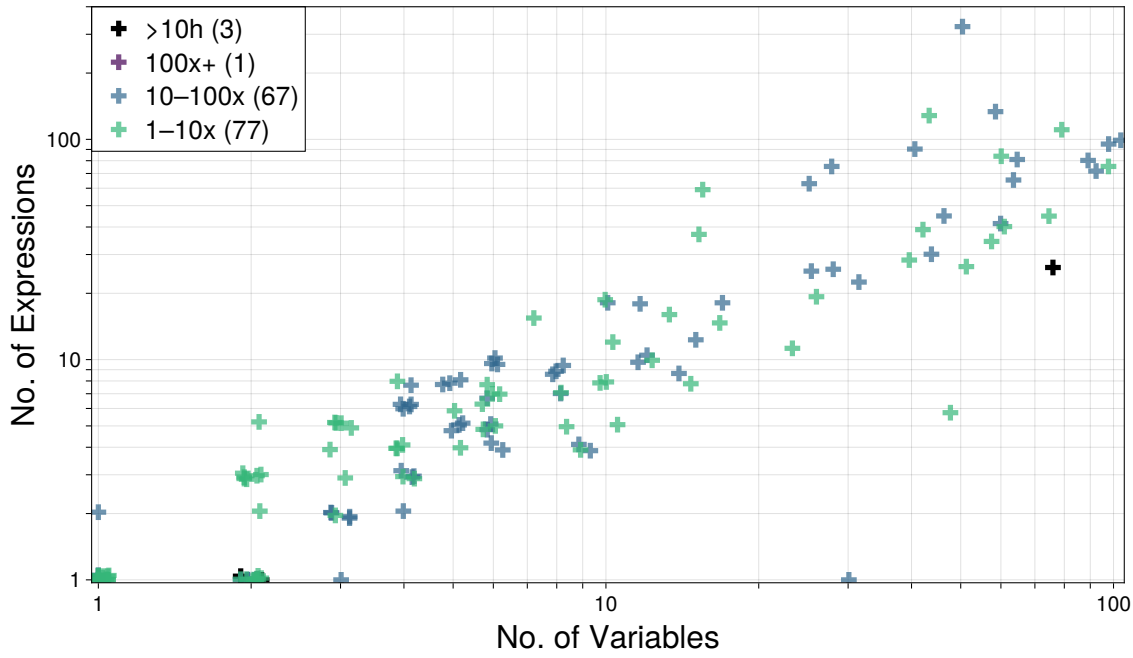

Figure 4.5: A scatterplot of problem sizes used during benchmarking with point color determined by the increase in compilation time required for `SCMC` versus `McCormick.jl`. Data is shown with a uniform random factor within ±5% added to each point to help distinguish problems of identical sizes. For each problem, the number of expressions is the number of unique constraints plus one for the objective function. Compilation times were recorded as the total time to compile all expressions for a given problem in preparation for runtime evaluations. Three problems have no comparisons available because `SCMC` functions took more than 10 hours to compile and were terminated early. Of the remaining problems, using `SCMC` increased the compilation time by a factor within $[2.38, 179]$.

complexity and compilation time, rather than a dependence on the number of expressions or variables. In particular, three problems exceeded the 10-hour compilation time limit set for `SCMC`, which included two problems with two variables and no constraints (`kriging_peaks-red200` and `kriging_peaks-red500`), and one problem with 75 variables and 25 relatively simple constraints (`elec25`). In contrast, the problem `gsg_0001` has 78 variables and 112 constraints, but only required 6.49x longer compilation time than `McCormick.jl`.

`SCMC` experiences longer compilation times than `McCormick.jl` due to the source code generation process inherent to the `SCMC` approach. Specifically, the `McCormick.jl` package relies on a library of comparatively short subroutines that correspond to mathematical operators, which are applied using multiple dispatch (the Julia equivalent of operator overloading). These subroutines are individually quick to compile. In contrast, `SCMC` creates much longer subroutines, which create more work for the Julia compiler and are more complex to optimize. As a representative example, the objective function in the `kriging_peaks-red100` example uses the operators: $\{-, +, *, \exp, \text{sqrt}, \hat{}\, 2\}$, which

represents a fairly short list of `McCormick.jl` subroutines that must be used. In `McCormick.jl`, this example compiles in 57 seconds. For `SCMC`, however, because the objective function itself is long, the file containing the generated kernels is roughly 170,000 lines of code. This code takes roughly 171 minutes to create and compile as GPU kernels, and about 100 minutes to compile if manually converted to CPU subroutines.

For cases where the compilation time of `SCMC`-generated functions is considerably longer than the compilation time of the `McCormick.jl` approach, there is a practical concern that the time saved by faster runtime execution over the course of the B&B run will not overcome the time lost in compilation. This is less of a concern if the same problem must be solved many times, such as on different domains, as the compilation time cost must only be paid once. For more general cases, however, decreasing the compilation time of the `SCMC` approach is a topic of our future work. Possible strategies to reduce the compilation time may include:

1. Identification of variables that can be treated as parameters, rather than state variables in the optimization problem. This would decrease the complexity of any expression these variables are involved in, as, for example, multiplication of two variables using Step 3b of Definition 2.6.14 is far more complex than multiplying by a variable that will appear as a floating-point value during calculation.

2. Recognizing repeated subexpressions and creating reusable parameterized kernels. I.e., rather than determine kernel break points heuristically based on code length, care could be taken to specifically create kernels that can be used multiple times. This would allow portions of code to be reused, rather than forcing the compiler to optimize effectively the same code multiple times.

3. Rather than inlining most or all of the McCormick-based rules within each kernel, some portions of code could be replaced with *device functions*, which are subroutines called directly from the GPU. Similar to the reuse of kernels in the previous point, this substitution would reduce the amount of repeated code within the kernels, which would further reduce compiler effort.

4.4.2 Evaluation Runtime

Similarly to Figure 4.5, Figure 4.6 shows a comparison of evaluation runtimes between `SCMC` and `McCormick.jl`, where the location of points within the plot corresponds to the NLP problem size,

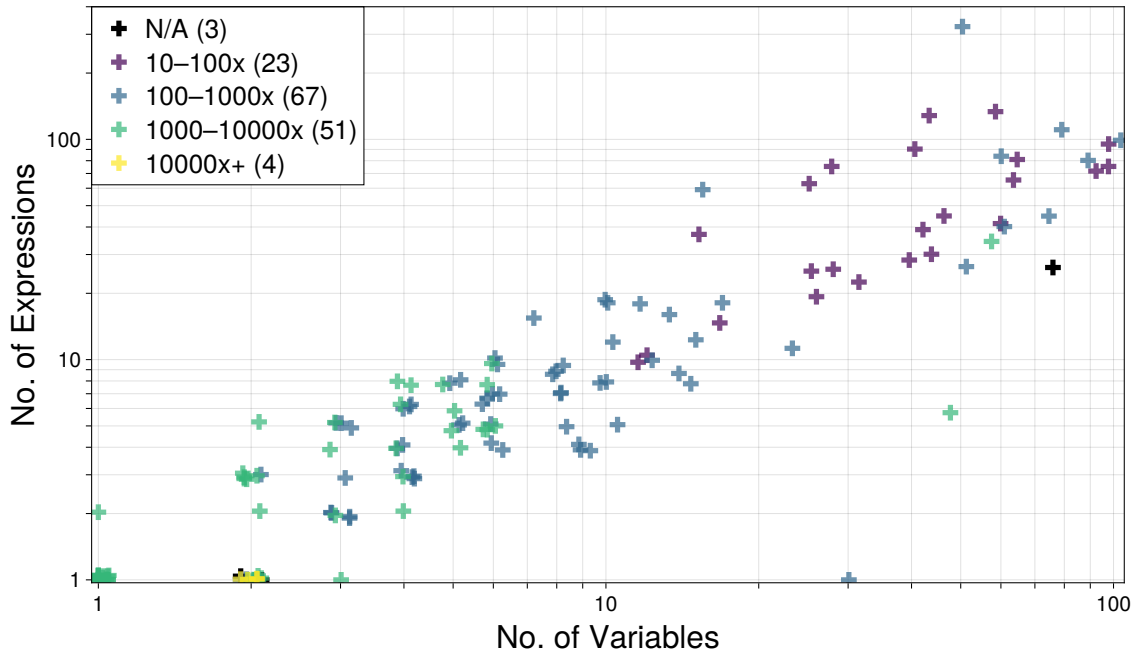

Figure 4.6: A scatterplot of problem sizes used during benchmarking with point color determined by the evaluation runtime speedup of relaxation evaluations for `SCMC` versus `McCormick.jl`. Data is shown with a uniform random factor within ±5% added to each point to help distinguish problems of identical sizes. For each problem, the number of expressions is the number of unique constraints plus one for the objective function. Evaluation times were recorded as the total time to obtain 20,480 relaxation and subgradient evaluations for each expression in a given problem. Three problems have no results because `SCMC` functions took more than 10 hours to compile and were terminated early. Of the remaining problems, the range of speedup factors was $[10.8, 15320]$.

and the color of the points indicates the relative speed of the two packages. While compilation times were always slower for `SCMC`, the evaluation runtimes were always faster by at least a factor of 10. For the vast majority of problems (80%), `kgen`-generated subroutines were between two and four orders of magnitude faster than equivalent expressions using `McCormick.jl`. It is important to mention that for both `SCMC` and `McCormick.jl`, calculations were performed using double-precision floating-point values. This may lead to degraded performance on many GPUs, as GPU hardware is typically built with significantly more single-precision floating point capacity than double-precision. For GPUs used for scientific computing, such as the GV100, the compute ratio between double-precision and single-precision floats is 1-to-2, but in many GPUs this ratio can be as low as 1-to-64. Due to the cumulative nature of the McCormick-based rules, the forced double-precision in `SCMC` is important to maintain reasonable precision, but this may lead to a drop in performance depending on the specific GPU being used. Notably, there is an apparent trend in Figure 4.6 where `SCMC` achieved a smaller speedup on problems with more expressions and more variables. A large portion of this

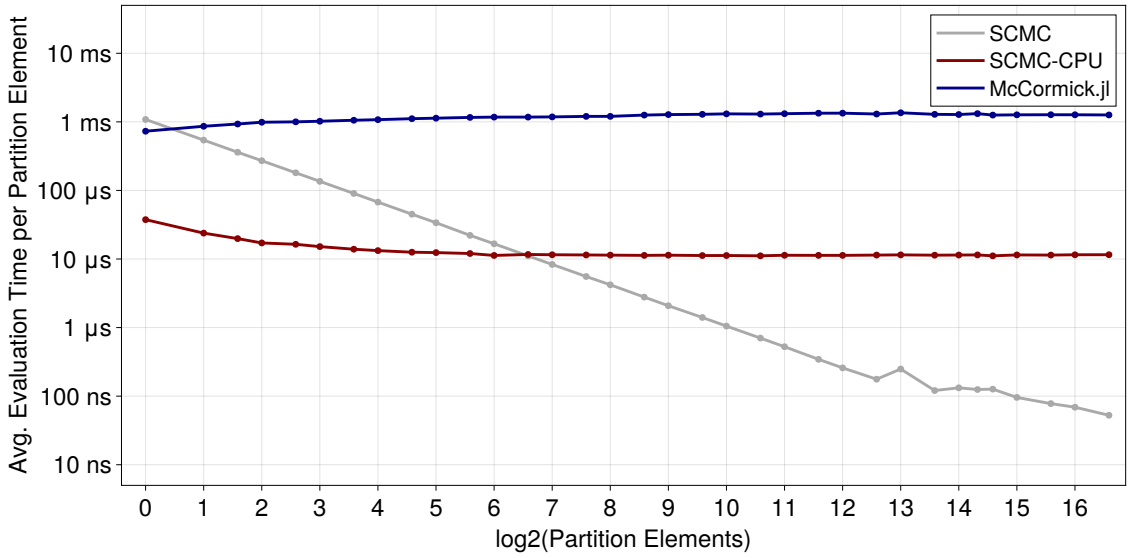

Figure 4.7: Average evaluation time as a function of the number of partition elements for the objective function of `kriging_peaks-red100` [1] for the (gray) `SCMC`, (red) `SCMC-CPU`, and (blue) `McCormick.jl` methods. The `SCMC` method benefits from GPU parallelization as the number of partition elements increases, whereas the `SCMC` kernel adapted for serial CPU operation (`SCMC-CPU`) and serial `McCormick.jl` had performance that was largely unaffected by calculation volume. Optimizations made during kernel writing make `SCMC` competitive with `McCormick.jl` even for a single partition element (i.e., without parallelization). In a direct CPU-to-CPU comparison, the CPU-adapted kernel is roughly two orders-of-magnitude faster than the `McCormick.jl` approach. With parallelization, `SCMC` eventually achieves a speedup of over four orders-of-magnitude relative to `McCormick.jl`.

apparent slowdown may be because many of the constraints in problems with more expressions are comparatively trivial. Simple expressions offer less of an opportunity for `SCMC` to gain an advantage, both because the calculations are already quite fast using CPU resources, and because the kernel launch time becomes a more dominant component of the runtime for those kernels. As problems feature more such constraints, this effect becomes more pronounced, leading to closer evaluation runtimes between the two approaches. In contrast, all four of the fastest examples for `SCMC` featured two variables and no constraints.

The speedups obtained by `SCMC` are multifactorial, depending on expression properties such as the dimensionality and sparsity pattern, the presence of reusable subexpressions, and the number and types of operators used. For example, two of the comparatively fastest problems, `kriging_peaks-red50` and `kriging_peaks-red100`, featured very long expressions that allowed more calculation throughput to occur on the GPU—though with longer corresponding compilation times (c.f. Sec. 4.4.1)—and also some subexpression repetition that `SCMC` could exploit by calculating the result once and reusing it in several places. This type of reuse is impossible with `McCormick.jl`, as it processes each

math operation independently, thereby giving SCMC a notable advantage. To help understand the performance of SCMC on kriging_peaks-red100, Figure 4.7 shows the average per-element evaluation time of the SCMC kernel, an adapted version of the kernel that allows it to operate on the CPU (SCMC-CPU), and the McCormick.jl approach. For each method, the average per-element evaluation time is shown as a function of the number of partition elements. By "average per-element evaluation time," we refer to the evaluation runtime calculated using the @belapsed macro, with the result divided by the number of partition elements used for a given evaluation. While the McCormick.jl and SCMC-CPU methods have times that are largely independent of the number of partition elements due to their serial operation, the SCMC method sees a decrease in average evaluation time as the number of partition elements increases as a direct result of the use of parallelized GPU hardware.

A notable feature of Figure 4.7 is the performance of SCMC on one partition element, where it is unable to make use of GPU parallelization. Despite GPU cores generally being individually weaker than CPU cores, the supporting optimizations performed by SCMC appear to sufficiently simplify the calculations to enable the source-code generated rule to be on par with the generalized McCormick.jl approach, even without the benefit of parallelization. This improvement is verified by the manual adjustment of the kernel to operate in serial using CPU resources (SCMC-CPU), which allows for a more direct comparison of the source code generation process against McCormick.jl. Using 20,480 partition elements as an evaluation point, McCormick.jl calculated relaxations and subgradients at an average rate of 1.32 ms per partition element. In comparison, SCMC-CPU obtained results at an average rate of 11.5 μs per partition element using the same CPU hardware, for a speedup of roughly 115x, and SCMC, using the GV100 GPU, calculated relaxations and subgradients in 125 ns per partition element, giving a speedup of 10500x relative to McCormick.jl. We note that this speedup is substantially more than what could potentially be obtained by using a multicore implementation of McCormick.jl.

Another of the four fastest examples, ex8_1_6, features a much shorter expression with no opportunity for subexpression reuse. In this case, the expression itself is fortuitously compatible with GPU parallelization. Specifically, the objective function of ex8_1_6 can be expressed with the summation:

$$f(\mathbf{x}) = \sum_{i=1}^{3} \frac{-1}{a_i + (x_1 - b_i)^2 + (x_2 - b_i)^2}, \tag{4.13}$$

with $a_i, b_i, \quad i = 1, 2, 3$, as constants. As observed in [58], one of the notable challenges with implementing McCormick-based rules on GPU hardware is the presence of branching within the rules. Conditional branching, such as choosing which terms to calculate in the min and max terms of

Step 3b in Definition 2.6.14, can lead to the phenomenon of warp divergence in which threads within a single warp attempt to follow different paths, which is known to be detrimental to GPU performance [63]. `ex8_1_6` is especially fast because the McCormick rules for most of its operators, $\{+, -, \hat{} 2\}$, feature little or no branching. Only one of its operators, `inv`, has a notable conditional branch, which checks for the sign of the operand. However, because all of the terms in the denominator are guaranteed to be non-negative, only one branch of `inv` is ever accessed and warp divergence remains low. Evaluating the generated kernel using the Nsight Compute profiling tool [109] confirms this intuition, showing an average number of active threads per warp of 31.29. This is close to the maximum possible value of 32, indicating relatively low warp divergence. To verify the importance of expression structure on this result, we generated another kernel using the same objective function as `ex8_1_6`, but with the sign of the denominator term containing x_2 set to negative instead of positive. This sign change allows the denominator to vary in sign, which causes warp divergence. Consequently, this new kernel shows an average number of active threads per warp of 24.98, indicative of moderate warp divergence.

Profiling of additional `SCMC`-generated kernels revealed more general conclusions about what limits their performance. One of the most notable weaknesses identified by Nsight Compute is the heavy use of registers, which are used in generated kernels to, for example, store temporary interval, relaxation, and subgradient values during the calculation of McCormick-based rules. Register space available in GPUs can be flexibly allocated to threads, but because the total register space in any GPU is limited, using a high number of registers per thread limits the number of warps that can be run concurrently. This is problematic from a performance perspective because GPUs hide latency—such as long memory access times—by issuing instructions to available warps while other warps are busy. In essence, because of the high register usage, the GPU cannot launch enough parallel work to effectively hide individual high-latency operations. A separate but related aspect is that individual McCormick-based rules are computationally simple but have high memory requirements, such as the need to store double-precision floating point numbers for the intervals, relaxations, and each element of the relaxation subgradients for inputs, outputs, and all intermediate terms. Roofline analysis confirms that the kernels created for the benchmarking examples are generally memory-bound, and the memory workload analysis produced by NSight Systems highlights that memory access patterns are currently suboptimal. In sum, kernels created by `SCMC` are limited by latency issues stemming from high memory requirements relative to compute throughput, which is exacerbated by inefficiencies in memory access. Addressing both of these challenges, such as by combining McCormick-based rules together to limit intermediate storage requirements and improving

thread memory access patterns, are areas of future work.

As a final note regarding performance, it is worth reiterating the discussion of data transfer to and from the GPU. Specifically, the times recorded in the benchmark exclude the time required to bring evaluation points from the CPU to the GPU, and exclude the time required to return relaxation and subgradient results to the CPU. The choice to exclude these times was made to better isolate the calculation speed of the kernels. Additionally, the inclusion of all input and output operations may not be realistic to how `SCMC` may be used in practice. For example, `kgen`-generated functions expect one `CuArray{Float64}` input of size $(m, 3)$ for each variable, though only a fraction of this array might originate on the CPU. Strategies such as constructing `CuArray{Float64}` objects on the GPU directly, using a lesser amount of information relating to subdomain bounds, may be a reasonable approach. Similarly, although a `CuArray{Float64}` of size $(m, 4 + 2n)$ is needed to hold interval, relaxation, and subgradient output information, not all of this data may need to be transferred back to the CPU. If the subgradient information is used to construct an LP that can be directly solved using GPU resources, the subgradient information itself may never leave GPU memory. Certainly, some data will need to be transferred at the beginning and end of the GPU operations overall, but it is strongly implementation-dependent. E.g., depending on how information is transferred and used, it is possible that some cases will yield no speedup over a CPU-based method. For this reason, rather than account for worst-case data transfer requirements, we excluded all data transfer, such that future applications can more easily discern where time is being spent.

4.5 Conclusions

This chapter introduces a novel method of calculating McCormick-based relaxations and their associated subgradients that relies on the algorithmic generation of custom, expression-specific CUDA kernels. By design, this approach enables deterministic global optimizers to obtain information for their lower-bounding procedures using GPU parallelization. Compared to the more traditional method of calculating relaxations and their subgradients using multiple dispatch in Julia on a CPU, these developments incur a higher up-front cost in terms of compilation time, increasing the first-call time by 1–2 orders-of-magnitude, but greatly improve the relaxation evaluation time, typically by 2–3 orders-of-magnitude using our available hardware. The fastest speeds were obtained for more complex expressions, and for expressions that characteristically caused less warp divergence. The speedup itself is the result of parallelization as well as a number of code optimizations that are not possible with run-time operator overloading methods, such as special-case rule simplifications and

the utilization of subexpression sparsity information. Additionally, while the developments within this work are exclusive to the Julia language, we note that this framework can be generalized to other languages, our approach does not rely on functionality or paradigms that are unique to Julia.

From a practical standpoint, the currently high compilation times are a notable limitation. Despite the substantial speedups possible for relaxation evaluations, if the compilation time for `SCMC` would exceed the NLP solve time using CPU-based methods, it may be more valuable to forego GPU acceleration using this method. Making this determination for a given problem may be challenging, as the B&B solve time of a given problem cannot easily be determined without solving the problem. However, given that the compilation time appears to scale nonlinearly with expression complexity, but does not scale poorly with problem dimensionality or the number of expressions in a problem, there are likely many problems where the solve time using traditional methods exceeds the `SCMC` compilation time. Additionally, for problems that may need to be solved multiple times, this method may be especially helpful as the compilation cost would only need to be paid once. Nonetheless, the practical implications make decreasing the compile time a primary focus for future development efforts.

It is also valuable to consider how `SCMC` might best be used by a GPU-accelerated DGO solver. In particular, lower-bounding techniques used by existing solvers are typically designed under the assumption of serial relaxation calculations, which results in methods that may not be able to effectively capitalize on the benefits of parallelization. Our CPU-based solver EAGO, for example, uses for its lower-bounding routine an adapted version of Kelley's cutting-plane method [74], in which new linearization points are created based on the solutions of a sequence of linear programs. This one-to-one ratio of relaxation evaluations and LP solves may be highly suboptimal if the relaxation calculations can be performed orders-of-magnitude faster using GPU resources. Alternative methods that rely on a greater number of relaxation evaluations per LP solve, such as the n-simplex algorithm proposed by Najman et al. [104], may be particularly good candidates for GPU-accelerated lower-bounding routines. More generally, the calculation speed made possible by `SCMC` may result in new lower-bounding techniques that can make use of an excess of relaxation and subgradient information, potentially leading to tighter relaxations, fewer B&B nodes explored, and faster overall solve times. The outlook for GPU parallelization of B&B for NLPs is strong, and in our future work, we aim toward the development of a DGO solver that can efficiently exploit the massive parallelism of this architecture.

Chapter 5

Re-Architected PDLP for Batch Parallelization of Linear Programs

Deterministic global optimization approaches generally rely on solving thousands or millions of linear programs (LPs) to obtain bounding information. These LPs are structurally similar to one another and are substantially smaller than those typically considered in LP literature. Following the recent successes of the primal-dual hybrid gradient (PDHG) method adapted for LPs (PDLP), including GPU-based implementations such as cuPDLP, this chapter presents `BatchPDLP`: a GPU-based batched linear solver based on PDLP. This implementation is built to function as a component of a deterministic global optimizer with the goal of solving many of the required bounding LPs simultaneously. On a test set of LPs generated from relaxations of global optimization problems in MINLPLib, `BatchPDLP` outperforms commercial and open-source solvers including Gurobi, HiGHS, and GLPK on 48% of tested problems, or 65% with problem-specific iteration limits, making it a uniquely valuable component for accelerating deterministic global optimizers. At the time of writing, this chapter is currently under review for publication and appears as reference [51].

5.1 Introduction

Problems of practical interest in science and engineering frequently feature nonconvexities as a consequence of representing physical phenomena. These nonconvexities make mathematical optimization more challenging, particularly if globally optimal solutions to these problems are required. In several contexts such as model verification [127], safety [134], and robust control [85], the need to obtain

guaranteed global solutions is unavoidable. The challenge is that nonconvex optimization problems are NP-hard [65], and consequently, solving them to guaranteed ϵ-optimality using deterministic global optimization (DGO) techniques is extremely computationally expensive as even state-of-the-art algorithms exhibit worst-case exponential scaling. As in other computationally expensive realms, such as machine learning model training [3], one approach to manage the computational cost of DGO is to transition to parallel-processing computer hardware such as graphics processing units (GPUs).

The difficulty in developing an efficient GPU implementation of a DGO algorithm lies in the appropriate distribution of work. Internally, DGO solvers mainly rely on some variation of the spatial branch-and-bound (B&B) algorithm to solve nonconvex nonlinear programs (NLPs) of the form [104]:

$$\begin{aligned} f^* = \min_{\mathbf{x}} \; & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{x} \in X \subset \mathbb{R}^n. \end{aligned} \tag{5.1}$$

Chapter 2, specifically Section 2.6.1, covers the B&B algorithm in detail, and Section 2.6.3 describes the standard lower-bounding process by which (5.1) is converted into the relaxed LP:

$$\begin{aligned} f^*_{\text{aff}} = \min_{\mathbf{x}} \; & \zeta \\ \text{s.t.} \quad & f_{\text{aff}}(\mathbf{x}) \leq \zeta \\ & \mathbf{g}_{\text{aff}}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{x} \in X' \subset X \subset \mathbb{R}^n, \end{aligned} \tag{5.2}$$

With this standard approach in mind, one way to apply GPU hardware architectures to a global solver built to solve (5.1) is to parallelize an LP solution method. The widespread importance of linear programming has strongly motivated research into the use of GPUs to accelerate LP solution methods. Several GPU-based implementations of the standard simplex algorithm [82, 96], revised simplex algorithm [17, 114, 123, 132], exterior point simplex algorithm [114], and interior point method [45, 70, 129] have appeared in the literature. Although, until recently, commercial solvers for LPs such as Gurobi [61], did not utilize GPU hardware [140]. The primary rationale for this lack of hardware support was that GPUs were not seen as well-suited for the sparse linear algebra that

dominates linear programming [140]. However, in 2024, NVIDIA released the cuDSS library [108], which enables efficient handling of sparse linear systems and which is now (at the time of writing) used by Gurobi in a GPU-accelerated barrier method [25]. In another recent development, Applegate et al. [10] introduced the PDLP algorithm and demonstrated that first-order methods—that rely on sparse matrix-vector multiplication as opposed to the sparse matrix factorization used by simplex and interior point methods—could be made competitive with standard LP solution methods. Sparse matrix-vector multiplication can already be efficiently performed by GPUs, and consequently this development quickly resulted in an open-source GPU implementation named `cuPDLP.jl` [89], as well as a GPU-accelerated version of PDLP used in Gurobi for large-scale LPs [25].

In each of the referenced GPU-accelerated LP methods, the parallelization capacity of the GPU is aligned with the dimensionality of the LP being addressed. That is, the larger the LP, the better the algorithms are able to exploit the parallelized hardware with speedups over CPU-based solvers generally realized on problem sizes over 500 variables and constraints [17, 70, 96, 132], with some examples not testing LPs smaller than 1000 variables and constraints [82, 114, 129]. For the purposes of accelerating DGO, this set of approaches is inadequate because the scale of problems commonly considered by DGO researchers typically feature far fewer than 1000 variables, and often fewer than 100 variables [1, 24], due to challenges associated with the curse of dimensionality. Using the lower-bounding scheme described previously, LPs of the form (5.2) will share the dimensionality of the global solver's internal problem representation, which is comparable to the problem size. LPs solved within DGO solvers are therefore typically too small to benefit from LP methods that have been accelerated in this manner.

To address the small LPs found in DGO subproblems using GPU resources, we can exploit the expectation that challenging DGO problems will require the solution of thousands or millions of LPs during execution of the B&B algorithm. The comparatively small size of these problems, in conjunction with the large number of problems that must be solved, naturally lends itself to a batch parallelization strategy. Only a few instances of the batch parallelization approach have been used in the current body of literature to address LP solving. The first implementation, to our knowledge, was created by Gurung and Ray [62], who achieved a roughly one order-of-magnitude speedup over the GLPK solver [94] when solving large batches of 100-dimensional LPs using the simplex method. Subsequently, Charlton et al. [32] developed a batch-parallelized implementation of Seidel's incremental LP algorithm [122] for two-dimensional LPs, which obtained a speedup of 22x over the implementation of Gurung and Ray [62]. More recently, Saha et al. [118] implemented parallel simplex and interior point methods and observed a speedup of one-to-two orders-of-magnitude over

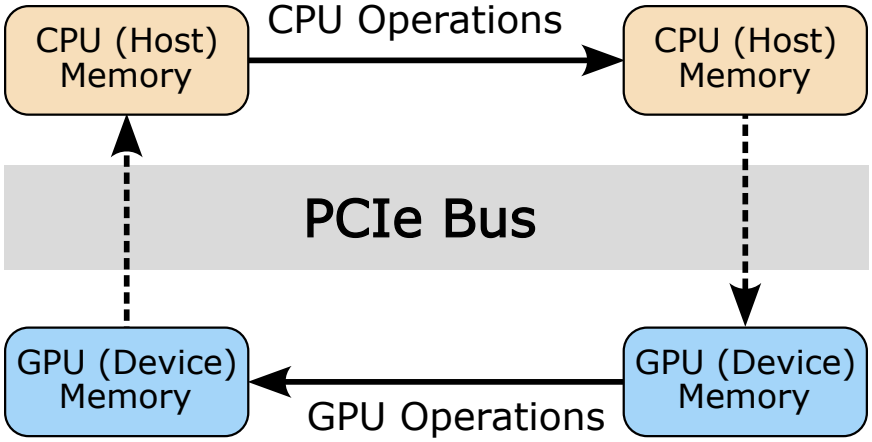

Figure 5.1: Schematic representing distinct CPU/host and GPU/device data storage. Solid arrows represent CPU or GPU operations, which are typically fast processes. Dashed arrows represent data transfer between different memory components through the PCIe Bus, which is typically slow.

the SciPy [146] `linprog` solver using its simplex and interior-point methods. Although not for LPs, a similar batching approach was used by Amos and Kolter [8] to solve quadratic programs with fewer than 1000 variables using a primal-dual interior-point method for problems embedded within neural networks, which resulted in a 26x speedup over the serial Gurobi solver [61].

Solving batched LPs in the context of DGO confers several additional advantages that may be exploited to achieve strong performance. First, given that the form of (5.2) is dependent on the form of (5.1), LPs solved over the course of solving (5.1) to ϵ-global optimality will have a fixed dimensionality, which would allow a solver to preallocate and reuse memory across multiple LP instances. Second, typical GPU-based methods—including the batched LP solvers previously mentioned—are constrained by slow data transfer rates between the host (CPU) and device (GPU), as represented in Figure 5.1. An important consideration for any GPU implementation is how and when to copy data, such as the objective, constraint, and right-hand side information of LPs, onto the device so that the GPU may begin processing the data. For example, Gurung and Ray [62] populate LPs on the host and then overlap memory copies with kernel execution to hide copy times. However, for DGO, LPs are populated based on the calculation of relaxations of (5.1). If this task can also be performed on a GPU, as we have done in our previous work [55, 58], then the LPs themselves may be generated directly in device (GPU) memory. Furthermore, solutions of the LPs do not necessarily need to be transferred to the CPU, as they may be used for other GPU-based calculations such as generating additional constraints for (5.2). It may also only be necessary to transfer the optimal objective value f^*_{aff} to the CPU, as opposed to the full optimal solution $\mathbf{x}^* \in \mathbb{R}^n$.

Considering a batched LP solver as a modular component of a larger DGO algorithm, these factors effectively eliminate the need to transfer large quantities of data between the host and device.

Based on the specific needs and advantages of a GPU-accelerated LP solver for DGO, we propose in this chapter a GPU-based batch-parallel implementation of the PDLP algorithm, named `BatchPDLP`. `BatchPDLP` is developed to exist as a component of a larger DGO algorithm that can pass LP instances in GPU memory to the solver and utilize the outputs, all without data transfer to or from CPU memory. The remainder of this chapter is organized as follows. In Section 5.2, we will provide an overview of the original PDLP algorithm. Subsequently, Section 5.3 will cover details of our novel implementation and parallelization approach, Section 5.4 will describe our benchmarking approach, and Section 5.5 will describe and contextualize the speed of `BatchPDLP` on problem instances with dimensionalities relevant to DGO, as well as discuss limitations of this method. Final remarks and plans for future work will be covered in Section 5.6.

5.2 Preliminaries

5.2.1 PDLP

As described by Applegate et al. [10], the PDLP algorithm solves primal

$$\begin{aligned} \min_{\mathbf{x}\in\mathbb{R}^n} \ & \mathbf{c}^\top \mathbf{x} \\ \text{s.t. } & \mathbf{G}\mathbf{x} \geq \mathbf{h} \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned} \tag{5.3}$$

and dual

$$\begin{aligned} \max_{\mathbf{y}\in\mathbb{R}^{m_1+m_2},\boldsymbol{\lambda}\in\mathbb{R}^n} \ & \mathbf{q}^\top \mathbf{y} + \mathbf{l}^\top \boldsymbol{\lambda}^+ - \mathbf{u}^\top \boldsymbol{\lambda}^- \\ \text{s.t. } & \mathbf{c} - \mathbf{K}^\top \mathbf{y} = \boldsymbol{\lambda} \\ & \mathbf{y}_j \geq \mathbf{0},\ j = 1, \ldots, m_1 \\ & \boldsymbol{\lambda} \in \Lambda \end{aligned} \tag{5.4}$$

LPs, where $\mathbf{G} \in \mathbb{R}^{m_1 \times n}$, $\mathbf{A} \in \mathbb{R}^{m_2 \times n}$, $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{h} \in \mathbb{R}^{m_1}$, $\mathbf{b} \in \mathbb{R}^{m_2}$, $\mathbf{l} \in (\mathbb{R} \cup \{-\infty\})^n$, $\mathbf{u} \in (\mathbb{R} \cup \{+\infty\})^n$, $\mathbf{K}^\top = (\mathbf{G}^\top, \mathbf{A}^\top)$, $\mathbf{q}^\top := (\mathbf{h}^\top, \mathbf{b}^\top)$, and

$$\Lambda = \Lambda_1 \times \cdots \times \Lambda_n, \qquad \Lambda_i := \begin{cases} \{0\} & l_i = -\infty,\ u_i = +\infty \\ \mathbb{R}^- & l_i = -\infty,\ u_i \in \mathbb{R} \\ \mathbb{R}^+ & l_i \in \mathbb{R},\ u_i = +\infty \\ \mathbb{R} & \text{otherwise} \end{cases}$$

is the domain from which $\boldsymbol{\lambda}$ can be chosen such that the dual objective is finite [10]. The main PDLP algorithm is then applied to the equivalent saddle-point problem:

$$\min_{\mathbf{x} \in X} \max_{\mathbf{y} \in Y} \mathcal{L}(\mathbf{x}, \mathbf{y}) := \mathbf{c}^\top \mathbf{x} - \mathbf{y}^\top \mathbf{K} \mathbf{x} + \mathbf{q}^\top \mathbf{y} \tag{5.5}$$

with $X := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$, and $Y := \{\mathbf{y} \in \mathbb{R}^{m_1 + m_2} : \mathbf{y}_j \geq \mathbf{0},\ j = 1, \ldots, m_1\}$ [10]. When applied to (5.5), the primal-dual hybrid gradient (PDHG) algorithm [31], on which PDLP is based, proceeds by applying the update:

$$\begin{cases} \mathbf{x}^{k+1} = \text{proj}_X(\mathbf{x}^k - \tau(\mathbf{c} - \mathbf{K}^\top \mathbf{y}^k)) \\ \mathbf{y}^{k+1} = \text{proj}_Y(\mathbf{y}^k + \sigma(\mathbf{q} - \mathbf{K}(2\mathbf{x}^{k+1} - \mathbf{x}^k))), \end{cases} \tag{5.6}$$

where τ and σ are the primal and dual step sizes, respectively [10]. Within PDLP, these are reparameterized as:

$$\tau = \eta/\omega \quad \texttt{and} \quad \sigma = \omega\eta \quad \text{with } \eta, \omega > 0,$$

where η is the *step size* and ω is the *primal weight* [10, 89]. To adapt PDHG to LPs, Applegate et al. [10] introduced several algorithmic improvements including adaptive step size and primal weight values and occasional restarts, and then applied presolving and diagonal preconditioning prior to starting the algorithm [10]. The PDLP algorithm overall is summarized by Applegate et al. [10, Algorithms 1–3].

5.3 `BatchPDLP` Implementation

The specific needs for our LP solver—namely, the ability to solve many small, similarly formatted LPs and the desire for minimal CPU–GPU memory transfer—necessitated the development of a

novel implementation of the PDLP algorithm. Much like `cuPDLP.jl` [89], `BatchPDLP` is designed to run using GPU resources, but the manner in which we employ those resources is entirely different than cuPDLP. By virtue of our intended application space of DGO, we are able to make simplifying assumptions about individual LPs, as well as about the relationship between LPs being solved in each batch, that allow for code designs and shortcuts that are inaccessible to more general solvers. As a trivial example, because the problems we consider come from affine relaxations of (5.1), we omit the handling of equality constraints within our solver.

More specifically, `BatchPDLP` is designed around the concept that all LPs that are solved at any time are identically structured in the form of (5.2), with the only differences between them being the domain X' and the constraint coefficients and constants. This identical structuring implies the following:

1. All LPs have the same number of variables. This allows constraint information for every LP to be stored in a single, fixed-width matrix.

2. All LPs have the same number of constraints. This makes it convenient to locate information for any individual LP within the constraint matrix or right-hand-side vector, as the data for any particular LP of interest start at row index $([\text{ID}] - 1) \times n_{\text{constraints}} + 1$, where $n_{\text{constraints}}$ is the (constant) number of constraints per LP.

3. The sparsity of each LP is (nearly) identical. I.e., if some constraint $g_p(\mathbf{x})$ only depends on some $x_i,\ i \in I \subsetneq \{1, 2, \cdots, n\}$, then in the rows of the constraint matrix corresponding to affine relaxations $g_{\text{aff},p}(\mathbf{x})$, we can assume that columns corresponding to $x_j,\ \forall j = \{1, 2, \cdots, n\} \backslash I$, are zero. Keeping track of this collective sparsity information allows us to skip calculations involving elements of the constraint matrix that are known ahead of time to be zero.

   Of course, it is possible that, for some LPs, the value corresponding to some $x_i,\ i \in I$ will happen to be zero despite the dependence of $g_p(\mathbf{x})$ on x_i. Although there is a small computational cost associated with performing calculations on such elements, we allow these calculations to proceed in favor of storing a single sparsity list that applies to every LP, which simplifies the code and lowers overall memory requirements.

In addition to these considerations, for our intended application, the individual LPs will be quite small relative to typical LPs addressed in the literature. For the purposes of this work, we consider LPs below 100 variables and 1000 constraints, as this is the range of problem sizes that typically arises from DGO subproblems. From a GPU architecture perspective, these values are well

Algorithm 3 Single-Kernel PDLP

for each LP (i) **do**
  Input: An initial solution $\mathbf{z}_i^{0,0}$;
  Initialize outer loop counter $\theta \leftarrow 0$, total iterations $k_i \leftarrow 0$,
   step size $\hat{\eta}_i^{0,0} \leftarrow 1/\|\mathbf{K}_i\|_\infty$, and primal weight
   $\omega_i^0 \leftarrow \texttt{InitializePrimalWeight}(\mathbf{c}_i, \mathbf{q}_i)$;
  repeat until termination criteria hold
    $t_i \leftarrow 0$;
    repeat until restart or termination criteria hold
      $\mathbf{z}_i^{\theta_i,t_i+1}, \eta_i^{\theta_i,t_i+1}, \hat{\eta}_i^{\theta_i,t_i+1} \leftarrow \texttt{AdaptivePDHGStep}(\mathbf{z}_i^{\theta_i,t_i}, \omega_i^{\theta_i}, \hat{\eta}_i^{\theta_i,t_i}, k_i)$;
      $\bar{\mathbf{z}}_i^{\theta_i,t_i+1} \leftarrow \frac{1}{\sum_{j=1}^{t_i+1} \eta_i^{\theta_i,j}} \sum_{j=1}^{t_i+1} \eta_i^{\theta_i,j} \mathbf{z}_i^{\theta_i,j}$;
      $\mathbf{z}_{\text{res},i}^{\theta_i,t_i+1} \leftarrow \texttt{GetRestartCandidate}(\mathbf{z}_i^{\theta_i,t_i+1}, \bar{\mathbf{z}}_i^{\theta_i,t_i+1}, \omega_i^{\theta_i})$;
      $t_i \leftarrow t_i + 1, k_i \leftarrow k_i + 1$;
    end
    Restart the outer loop. $\mathbf{z}_i^{\theta_i+1,0} \leftarrow \mathbf{z}_{\text{res},i}^{\theta_i,t_i}, \theta_i \leftarrow \theta_i + 1$;
    $\omega_i^{\theta_i} \leftarrow \texttt{PrimalWeightUpdate}(\mathbf{z}_i^{\theta_i,0}, \mathbf{z}_i^{\theta_i-1,0}, \omega_i^{\theta_i-1})$;
  end
  Output: $\mathbf{z}_i^{n_i,0}$.
end

aligned with the number of threads that work cooperatively in blocks: in NVIDIA GPUs produced since 2010, blocks comprise a maximum of 1024 threads [110], though due to factors such as kernel memory requirements, more complicated kernels may have an upper limit of, at a minimum, 256 threads. Due to the good alignment between problem size and the allowable number of threads per block, we designed `BatchPDLP` such that each thread block would handle one LP, with the number of threads per block set based on the (shared) size of the LPs. This distribution of work mirrors the abstractions built into CUDA, which "guide the programmer to partition the problem [solving many LPs] into coarse sub-problems [individual LPs] that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces [PDLP steps] that can be solved cooperatively in parallel by all threads within the block" [110].

As part of this overall design, `BatchPDLP` is implemented such that the main PDLP algorithm—i.e., not including problem scaling or other initial checks—is written as a single kernel, presented in Algorithm 3. This design contrasts that of `cuPDLP.jl`, which calls separate kernels for individual steps within the overall PDLP algorithm. `BatchPDLP` also maintains separate parameters for each LP, as opposed to the single set needed in cuPDLP. Keeping the main algorithm written as a single kernel provides several benefits for the batched LP use case:

1. Any time a kernel is launched, there is a brief ($\mathcal{O}(10)$ μs) launch delay. Although this delay is brief, individual PDLP iterations are also quick to calculate, especially with the smaller LPs being targeted by `BatchPDLP`. As a consequence, if each iteration required a new kernel

Algorithm 4 One PDHG step for the i-th LP using an adaptive step size [10]

function AdaptivePDHGStep($\mathbf{z}_i^{\theta_i,t_i}$,$\omega_i^{\theta_i}$,$\hat{\eta}_i^{\theta_i,t_i}$,$k_i$)**:**
  $(\mathbf{x},\mathbf{y}) \leftarrow \mathbf{z}_i^{\theta_i,t_i}$, $\eta \leftarrow \hat{\eta}_i^{\theta_i,t_i}$, $\omega \leftarrow \omega_i^{\theta_i}$;
  for $i = 1, \cdots, \infty$ **do**
    $\mathbf{x}' \leftarrow \text{proj}_{X_i}(\mathbf{x} - \frac{\eta}{\omega}(\mathbf{c}_i - \mathbf{K}_i^\top \mathbf{y}))$;
    $\mathbf{y}' \leftarrow \text{proj}_{Y_i}(\mathbf{y} + \eta\omega(\mathbf{q}_i - \mathbf{K}_i(2\mathbf{x}' - \mathbf{x})))$;
    $\bar{\eta} \leftarrow \frac{\|(\mathbf{x}'-\mathbf{x},\mathbf{y}'-\mathbf{y})\|_\omega^2}{2(\mathbf{y}'-\mathbf{y})^\top K_i(\mathbf{x}'-\mathbf{x})}$;
    $\eta' \leftarrow \min((1-(k_i+1)^{-0.3})\bar{\eta},\ (1+(k_i+1)^{-0.6})\eta)$;
    if $\eta \leq \bar{\eta}$ **then**
      return $(\mathbf{x}',\mathbf{y}'), \eta, \eta'$;
    $\eta \leftarrow \eta'$;
  end for
end

to be launched, the runtime of BatchPDLP may be dominated by launch delay. Designing the algorithm as a single kernel means the delay is only incurred once per batch of LPs, which drastically reduces the overall runtime compared to a multi-kernel design.

2. Containing the entire algorithm within one kernel allows BatchPDLP to make abundant use of shared memory, which is accessible within a kernel only to threads within an individual thread block. This reduces the total amount of information that must be stored at any time, since not all LPs will be managed simultaneously, and it also reduces the amount of data that must be stored in global (slower) GPU memory.

3. This organization allows for more independence between LPs. If the number of LPs being solved is greater than the number of blocks the GPU can run concurrently, then if one LP solves in fewer iterations than others, the next LP can be started using the newly freed hardware resources without waiting for the other initial LPs to finish. Similarly, this provides some resilience to the case where an individual LP takes an unexpectedly long time to finish, as no single LP should be able to hold up the remainder of the queue.

Algorithm 4 shows the dynamic calculation and application of the step size η for a given LP, which is practically the same as that used by Lu and Yang [89] and Applegate et al. [10]. Similarly, the initial primal weight calculation for BatchPDLP matches that of cuPDLP.jl:

$$\texttt{InitializePrimalWeight}(\mathbf{c}_i, \mathbf{q}_i) := \begin{cases} \frac{\|\mathbf{c}_i\|_2}{\|\mathbf{q}_i\|_2}, & \text{if } \|\mathbf{c}_i\|_2, \|\mathbf{q}_i\|_2 > 0 \\ 1, & \text{otherwise,} \end{cases}$$

as well as the primal weight update given by Algorithm 5 [10, 89]. In Algorithms 3–5 and the initial

Algorithm 5 Primal weight update [10]

function `PrimalWeightUpdate`($\mathbf{z}_i^{\theta_i,0}$, $\mathbf{z}_i^{\theta_i-1,0}$, $\omega_i^{\theta_i-1}$)**:**
  $(\mathbf{x}^{\theta,0}, \mathbf{y}^{\theta,0}) \leftarrow \mathbf{z}_i^{\theta_i,0}$;
  $(\mathbf{x}^{\theta-1,0}, \mathbf{y}^{\theta-1,0}) \leftarrow \mathbf{z}_i^{\theta_i-1,0}$;
  $\Delta_x^\theta \leftarrow \|\mathbf{x}^{\theta,0} - \mathbf{x}^{\theta-1,0}\|_2$;
  $\Delta_y^\theta \leftarrow \|\mathbf{y}^{\theta,0} - \mathbf{y}^{\theta-1,0}\|_2$;
  if $\Delta_x^\theta > 0$ and $\Delta_y^\theta > 0$ **then**
    return $\exp\left(0.5 \log\left(\frac{\Delta_y^\theta}{\Delta_x^\theta}\right) + 0.5\omega_i^{\theta_i-1}\right)$;
  else
    return $\omega_i^{\theta_i-1}$;
end

primal weight calculation, only the portions of LP information associated with one particular LP are ever utilized, such as $\mathbf{K}_i$ and $\mathbf{c}_i$. Due to this independence, calculating more, fewer, or different LPs in tandem will never affect the number of iterations required to solve any particular LP.

As with the original Julia implementation of PDLP [10], `BatchPDLP` utilizes an adaptive restarting strategy, resetting to either the current or an average iterate if a set of restart conditions is met. While the implementation of Applegate et al. [10] used the normalized duality gap as a restart metric, the GPU-based `cuPDLP.jl` utilized a novel scheme based on KKT error [89]. In this work, we choose to follow the approach of Lu and Yang [89] due to the simplicity of translating this method to our batched code design. Specifically, we calculate the KKT error as:

$$\texttt{KKT}(\mathbf{z}, \omega) = \left(\omega^2 \left\| \begin{pmatrix} \mathbf{Ax} - \mathbf{b} \\ [\mathbf{h} - \mathbf{Gx}]^+ \end{pmatrix} \right\|_2^2 + \frac{1}{\omega^2}\|\mathbf{c} - \mathbf{K}^\top \mathbf{y} - \boldsymbol{\lambda}\|_2^2 + \dots \right.$$
$$\left. \dots + (\mathbf{q}^\top \mathbf{y} + \mathbf{l}^\top \boldsymbol{\lambda}^+ - \mathbf{u}^\top \boldsymbol{\lambda}^- - \mathbf{c}^\top \mathbf{x})^2 \right)^{1/2}, \tag{5.7}$$

where $\mathbf{z} := (\mathbf{x}, \mathbf{y})$. The restart candidate is then chosen as the $\mathbf{z}$ that returns the smallest KKT error, i.e.:

$$\mathbf{z}_{\text{res},i}^{\theta_i,t_i+1} := \texttt{GetRestartCandidate}(\mathbf{z}_i^{\theta_i,t_i+1}, \bar{\mathbf{z}}_i^{\theta_i,t_i+1}, \omega_i^{\theta_i}) =$$
$$\begin{cases} \mathbf{z}_i^{\theta_i,t_i+1}, & \texttt{if KKT}(\mathbf{z}^{\theta_i,t_i+1}, \omega_i^{\theta_i}) < \texttt{KKT}(\bar{\mathbf{z}}^{\theta_i,t_i+1}, \omega_i^{\theta_i}) \\ \bar{\mathbf{z}}_i^{\theta_i,t_i+1} & \text{otherwise.} \end{cases}$$

Finally, we employ the same restart criteria as in Lu and Yang [89], choosing to restart the PDLP algorithm for the i-th LP if any of the following hold:

1. The KKT error has decreased to a factor of $\beta_{\text{sufficient}} = 0.2$ since the last restart:

$$\texttt{KKT}(\mathbf{z}_{\text{res},i}^{\theta_i,t_i+1}, \omega_i^{\theta_i}) \leq \beta_{\text{sufficient}} \texttt{KKT}(\mathbf{z}_i^{\theta_i,0}, \omega_i^{\theta_i}).$$

2. The KKT error has decreased to a factor of at least $\beta_{\text{necessary}} = 0.8$ since the last restart, and the most recent step yielded no progress:

$$\texttt{KKT}(\mathbf{z}_{\text{res},i}^{\theta_i,t_i+1}, \omega_i^{\theta_i}) \leq \beta_{\text{necessary}} \texttt{KKT}(\mathbf{z}_i^{\theta_i,0}, \omega_i^{\theta_i}) \quad \text{and}$$
$$\texttt{KKT}(\mathbf{z}_{\text{res},i}^{\theta_i,t_i+1}, \omega_i^{\theta_i}) > \texttt{KKT}(\mathbf{z}_{\text{res},i}^{\theta_i,t_i}, \omega_i^{\theta_i}).$$

3. The ratio of iterations since the last restart (t) to the total number of iterations (k) is at least $\beta_{\text{artificial}} = 0.36$:

$$\frac{t}{k} \geq \beta_{\text{artificial}}.$$

In sum, the `BatchPDLP` approach begins by accepting LP information for many similar LPs simultaneously, in the form of large matrices or vectors in GPU memory with one or more rows corresponding to each LP. Each of these LPs is then independently preconditioned by applying 10 iterations of Ruiz rescaling [117] and Pock and-Chambolle scaling [115] with $\alpha = 1$. Subsequently, the kernel described by Algorithm 3 is launched, and the solutions $\mathbf{x}^*$ and optimal objective values f_{aff}^* are saved to global GPU memory.

5.4 Benchmarking Methodology

In this section, we will describe the benchmarking approach used in Section 5.5 to compare the performance of `BatchPDLP` against a set of commercial and open-source solvers comprising Gurobi [61], HiGHS [67], and GLPK [94].These solvers each feature multiple LP solution methods, including primal simplex, dual simplex, and interior point methods, and Gurobi and HiGHS have newly implemented PDLP methods. For a robust comparison and to reduce the potential bias of some problems being more susceptible to one type of LP technique, we will compare `BatchPDLP` to each of these solution methods separately for the solvers mentioned.

There are many LP benchmark test sets available in the literature, but given the specialized context in which `BatchPDLP` is intended to be used, existing test sets are largely inadequate. In particular, and as described previously, `BatchPDLP` is built to solve large numbers of structurally

(near-)identical LPs simultaneously, and is explicitly not intended to handle the large, sparse LPs commonly found in test sets. Consequently, in this chapter we utilize a custom benchmark test set comprised of LPs generated from relaxations of NLPs, as this is the specific context `BatchPDLP` is built to address. NLP problems used to create this test set come from the MINLPLib database [1] with the following additional limitations:

1. The number of variables did not exceed 100 and the number of constraints did not exceed 1000. This limitation was placed to reduce the dataset down to problem sizes that might be handled by a deterministic global optimizer, and reduced the number of database NLPs from 239 to 158.

2. The operators used in the NLP did not include any of the following: {tanh, abs, log10, gamma, mod, cvpower, errorf, signpower, centropy, rpower}, where $\text{errorf}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-t^2}{2}} dt$; $\text{signpower}(x, y) = \text{sign}(x)|x|^y$ for $y > 0$; $\text{centropy}(x, y[, z])$ where $y > 0,\ z \geq 0$ returns $x \cdot \ln\left(\frac{x+z}{y+z}\right)$ if $x > 0$ and 0 if $x = 0$; and $\text{rpower}(x, y)$ returns x^y if $x > 0$, 0 if $x = 0, y > 0$, and 1 if $x = y = 0$. These operators were removed to simplify the calculation of convex and affine relaxations, and reduced the number of problems from 158 to 146.

3. The length of the objective function did not exceed 5000 characters. Similar to the previous limitation, these examples were removed to speed up the process of calculating relaxations and reduced the number of problems from 146 to 139.

4. The creation or solution of LPs did not cause errors. In some cases, the original problem featured terms such as log or sqrt that would cause numerical errors with a negative argument, in conjunction with variable bounds that would cause these errors to appear. In a global optimizer, these regions would likely be excluded with the application of constraint propagation techniques. For the present study, however, these regions could not easily be excluded, and therefore any offending examples were removed from the test set. This elimination reduced the number of problems from 139 to 124.

For each problem, we begin by partitioning the initial problem domain into $m = 10^4$ subdomains, with m selected to represent a typical number of subproblems a batched DGO solver might solve per iteration. For variables without lower and/or upper bounds, the missing bounds are arbitrarily set to $\pm 10^6$ prior to partitioning as the B&B algorithm used by DGO solvers has no well-defined branching rules for unbounded variables. Following the partitioning step, an LP of the form (5.2) is

generated for each subdomain using the following procedure, with interval extensions, relaxations, and relaxation subgradients calculated using the `SourceCodeMcCormick.jl` package [53, 54, 58]:

1. The LP subproblem is formulated as a minimization problem with the objective function being the epigraph variable ζ.

2. The lower bound of an interval extension of the original objective function $f(\mathbf{x})$ is added as a lower bound of ζ.

3. Three points in the subdomain X' are selected: $\lambda X'^{,\mathrm{LBD}}+(1-\lambda)X'^{,\mathrm{UBD}}$ for $\lambda \in \{0.25, 0.5, 0.75\}$, where $X'^{,\mathrm{LBD}}$ ($X'^{,\mathrm{UBD}}$) is the lower (upper) bound of X' in each dimension. For each point:

   (a) Evaluate a convex relaxation of $f(x)$ on the subdomain X' and its corresponding subgradient, from which we can construct a supporting hyperplane of the convex relaxation function that corresponds to $f(\mathbf{x})$. Add a constraint that ζ is greater-than-or-equal-to the supporting hyperplane;

   (b) For each "$\leq$" constraint in the NLP subproblem, calculate a convex relaxation and corresponding subgradient and add the supporting hyperplane as a constraint.

   (c) For each "$\geq$" constraint in the NLP subproblem, calculate a concave relaxation and corresponding subgradient and add the supporting hyperplane as a constraint.

   (d) For each "=" constraint in the NLP subproblem, calculate convex and concave relaxations and their corresponding subgradients and add both supporting hyperplanes as constraints.

The sizes of LP subproblems created using this approach are shown in Figure 5.2. Points are further color-coded based on the performance of `BatchPDLP`, as will be discussed in Section 5.5. A majority of LPs in this test set (95/124) have fewer than 20 variables and fewer than 100 constraints, and the highest-dimensional problem has 100 variables and 592 constraints. Although significantly smaller than LPs typically addressed in the literature, problems of this size are frequently encountered by subsolvers in DGO routines.

Once the 10^4 LPs are created from each NLP, they are passed in batch form to `BatchPDLP`, or serially to Gurobi, HiGHS, and GLPK. All problems were solved using one thread of an Intel Xeon W-2195 2.30/4.30 GHz (base/turbo) processor with 64 GB of RAM running a Windows 11 Enterprise 24H2 operating system and, for `BatchPDLP`, an NVIDIA Quadro GV100 GPU with 32 GB of HBM2 VRAM (driver v32.0.15.7657, CUDA v12.9.86). Julia v1.11.4 was used in conjunction with `CUDA.jl` 5.8.5 [16], `SourceCodeMcCormick.jl` v0.6.0, `JuMP.jl` v1.29.3 [41]. Solver versions

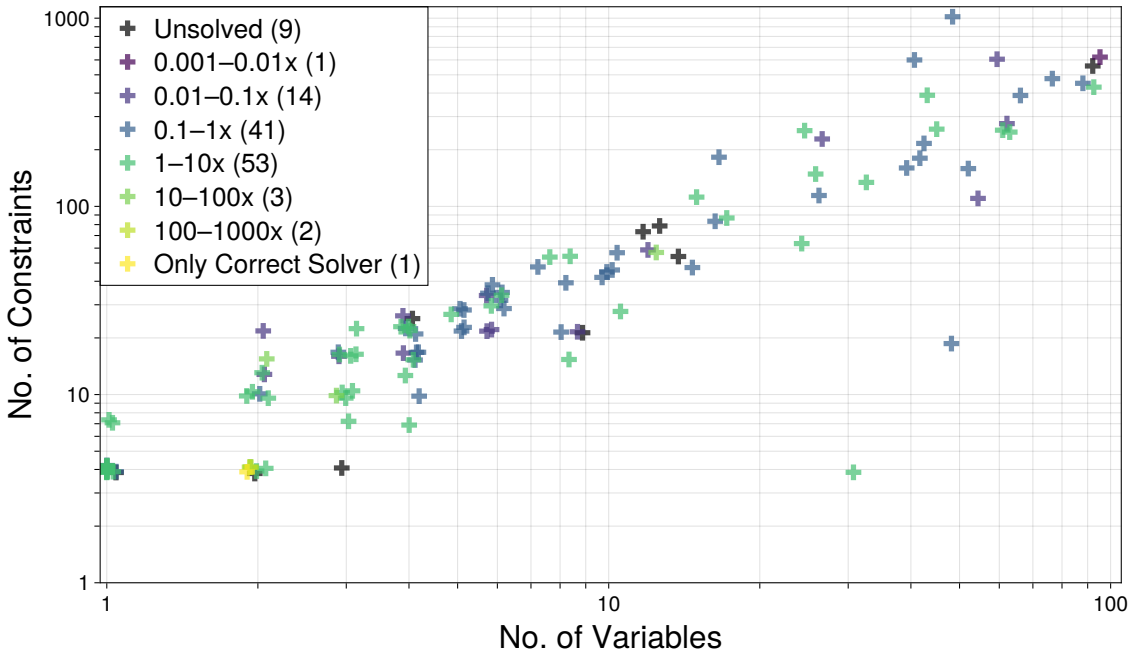

Figure 5.2: A scatterplot of LP problem sizes used during benchmarking. Data is shown with a uniform random factor within ±5% added to each point to help distinguish problems of identical sizes. Each LP is generated from an NLP in MINLPLib, where the LP shares the number of variables with the NLP and has $1+3(1+L+G+2E)$ constraints, where the parentheses includes terms that correspond to properties of the original NLP: 1 for the objective function, L total "$\leq$" constraints, G total "$\geq$" constraints, and E total "$=$" constraints. Points are colored based on the relative speed of `BatchPDLP` on each problem as compared to the fastest non-`BatchPDLP` solver.

used were `Gurobi.jl` v13.0.0 [61], `HiGHS.jl` v1.12.0 [67], and `GLPK.jl` v5.0 [94]. Note that for Gurobi experiments, the PDLP method was activated by setting `Method=6`.

5.5 Results and Discussion

5.5.1 Direct Solver Comparison

To facilitate discussion of the results, each starting NLP problem will be referred to as a *problem*, with the total solve time recorded as the sum of the solution times for each of the generated LP *subproblems*. Each subproblem is solved to a tolerance of 10^{-8}, and for each solver, a subproblem is marked as correct if it obtains the same termination status and objective value (within 10^{-7} absolute or relative tolerance) as the plurality of all solver results. Critically, many of the subproblems created in this benchmark set are numerically challenging, containing features such as near linear-dependence of constraints or wide ranges of constraint coefficients. As such, obtaining correct solutions (i.e.,

confirmed by another solver) for all 10^4 subproblems in a batch was non-trivial; four problems featured at least one subproblem where no consensus could be reached by any pair of solvers, in which case all solvers were marked as incorrect to prevent bias. Under the most rigid metric, where problems are only considered correct if all 10^4 subproblems are verified as correct, the highest (lowest) percentage of problems solved by each solver across its methods were 61% (–) for `BatchPDLP`, 69% (4%) for GLPK, 85% (72%) for Gurobi, and 81% (33%) for HiGHS. However, given the difficult nature of the subproblems and our goal of assessing performance more generally, we elect for the remainder of this section to judge each solver more flexibly, considering problems as correct if at least 90% of subproblems are solved correctly relative to the answers obtained by other solvers.

To serve as an initial comparison using the $\geq$90% metric, Figure 5.2 displays the performance of `BatchPDLP` on each problem, relative to the fastest non-`BatchPDLP` solver and solution method. On roughly half of all problems (53/124), `BatchPDLP` achieves a modest speed-up of 1–10x, and in roughly 4% of test cases (5/124), a speed-up of greater than 10x was observed; in the problem with the greatest speed-up, `BatchPDLP` obtained solutions 148x faster than any other solver. Additionally, on one problem, `BatchPDLP` was the only solver to obtain verified correct solutions on more than 90% of subproblems. On problems where `BatchPDLP` was outperformed by existing solvers (65/124), the solution speed was generally within an order-of-magnitude (41/65). In the slowest example, `BatchPDLP` took 224x longer to obtain its solutions than the fastest tested solver. Generally, `BatchPDLP` performed better on smaller problems, though good performance on larger problems was not uncommon, which suggests that even larger problems may still be amenable to this batch parallelization approach.

A performance profile [39] comparing all tested solvers on all problems in the custom benchmark set is shown in Figure 5.3. As compared to the other solvers, inclusive of all tested solving methods, `BatchPDLP` was the fastest on 48% of benchmark problems and achieved a solve time within an order of magnitude of the fastest solver and method in 81% of problems. The next fastest solver was GLPK, with 40% of the fastest times across each of its methods, followed by Gurobi (10%) and HiGHS (0%). In addition, while `BatchPDLP` was not able to correctly solve 90%+ of subproblems for every problem in the test set (93%), this is marginally superior to the strongest methods in GLPK (87%) and HiGHS (90%) and is only outcompeted by Gurobi (96%, all methods).

For the 48% of problems where `BatchPDLP` outperformed the other tested solvers, it is necessary to recognize that `BatchPDLP` was intentionally designed to handle the batched, low-dimensional LPs that are found in the benchmark test set. These problems are not expected to take a long time: across all problems and all of its methods, Gurobi finished over 97% of subproblems in under 1

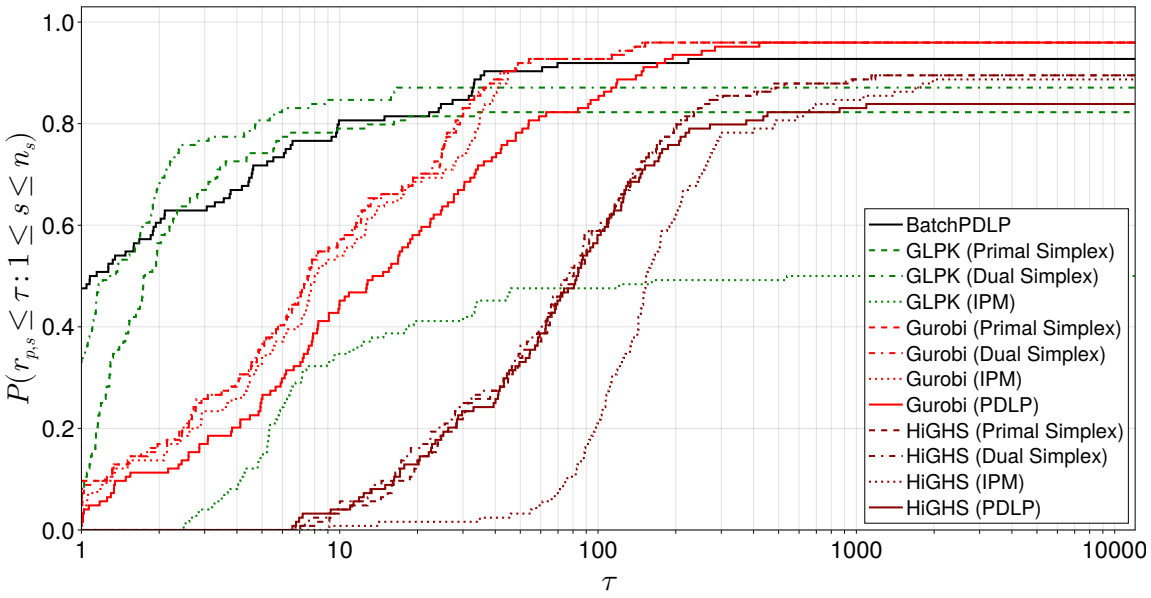

Figure 5.3: A Dolan and Moré [39] performance profile showing `BatchPDLP` and all available LP methods contained in GLPK, Gurobi, and HiGHS. `BatchPDLP` obtained the fastest result in 48% of benchmark test set problems, which is possible due to its tuned implementation to the problem format represented in the test set.

ms. To achieve better performance on these already fast problems, `BatchPDLP` makes sacrifices in both design and features that would negatively impact its performance on larger LPs. The single-kernel implementation of the main algorithm (Alg. 3), for example, makes use of relatively low-capacity memory resources that could be overwhelmed by larger problems which would lead to substantial performance degradation. Additionally, `BatchPDLP` only applies Ruiz rescaling [117] and Pock and Chambolle preconditioning [115] before solving LPs, and does not perform any presolving or other preconditioning techniques, or any postprocessing techniques such as crossover. This is in contrast with Gurobi and HiGHS, which are left on their default settings that allow the solver to determine whether to apply presolving and other methods. Although presolving techniques are helpful in general and can significantly reduce the effort required to obtain a solution, and crossover is valuable for providing interpretable solutions [25], for very small LPs the time to apply these techniques may be comparable to the overall solve time. For instance, across all problems, Gurobi's average subproblem solve time was always at least 139 μs, and HiGHS's average subproblem time was always at least 89 μs. In comparison, the fastest average subproblem solve time achieved by `BatchPDLP` was 1.0 μs.

Notably, while skipping the presolving step lowers the fastest average time reachable by `BatchPDLP`, this approach also makes `BatchPDLP` more susceptible to numerical issues during problem formula-

Table 5.1: Number of iterations required for `BatchPDLP` to converge across the 10^4 LP subproblems derived from the `mhw4d` NLP problem. Most LP instances solved in few iterations, with a median count of 10, but a small fraction of LPs required orders-of-magnitude more iterations than the median.

Iterations	Count
1–10	8896
11–100	493
101–1000	579
1001–10000	20
10001–100000	0
100001–1000000	12

tion. Identifying the presence of these numerical problems is critical for understanding the relatively poor performance of `BatchPDLP` on some problems in the test set. In particular, it was observed that on 18% (22/124) of problems, `BatchPDLP` reached its iteration limit of 10^6 on only a portion of subproblems; in more than half of these cases (13/22), fewer than 1% of subproblems hit the limit. However, nearly all of these problems saw degraded performance in `BatchPDLP` relative to the other solvers. As a representative case, the total number of iterations needed by `BatchPDLP` for the 10^4 LPs derived from the `mhw4d` NLP are shown in Table 5.1. On this problem, `BatchPDLP` took 6.5x longer than Gurobi (the fastest alternative) to complete its solves. Notably, although more than 99.6% of LPs (9968/10000) were solved in fewer than 1000 iterations, twelve problems terminated at the limit of 10^6 iterations. This disparity in iteration counts is especially detrimental to `BatchPDLP` because it is inherent in the design that only a small portion of GPU resources are devoted to any individual LP. Consequently, a small number of LPs requiring orders-of-magnitude more iterations means they will effectively be solved without the benefit of massive GPU parallelization that is central to the batched LP design.

This trend of few subproblems requiring vastly-greater-than-average numbers of PDLP iterations was observed for multiple problems in the test set, underlying a significant weak point for the `BatchPDLP` approach. While it is well known that first-order methods such as PDLP require more iterations than other LP solution techniques such as barrier methods [89], this skewed distribution of iteration counts across structurally similar LPs is remarkably consistent within the test set. Improvements to the underlying PDLP method, such as those implemented in `cuPDLPx` [88], may help to reduce the number of iterations needed to solve some of these particularly challenging LPs, although implementing these changes within `BatchPDLP` remains an area of future work.

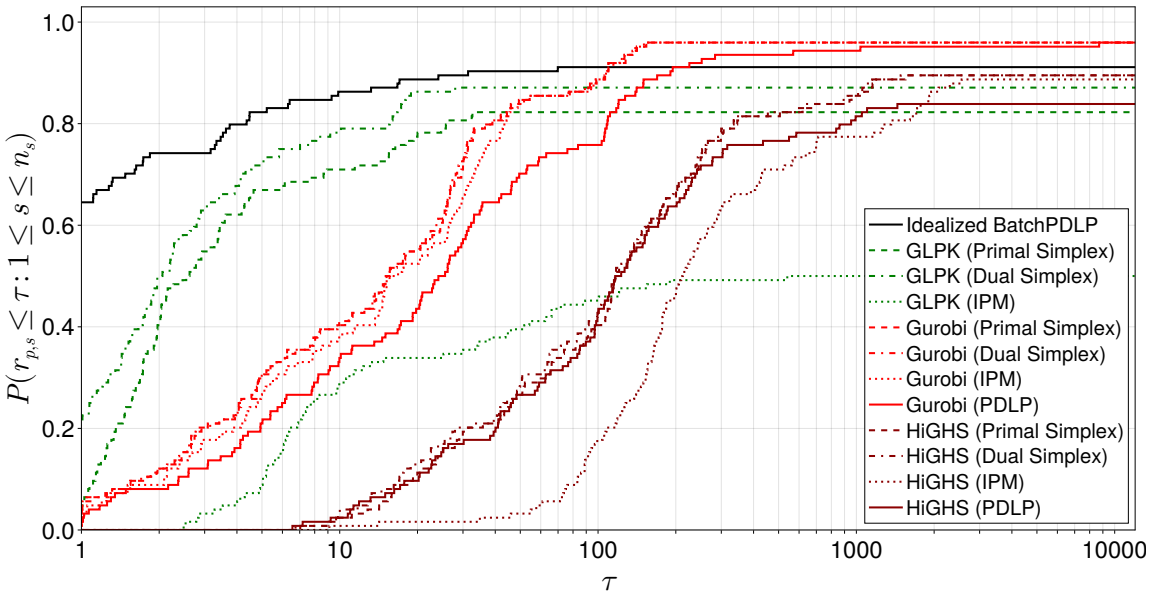

Figure 5.4: A Dolan and Moré [39] performance profile showing an idealized form of `BatchPDLP` and all available LP methods contained in GLPK, Gurobi, and HiGHS. Idealized `BatchPDLP` has iteration limits set independently for each problem such that at least 95% of LPs in each batch are solved normally, with the remainder terminating upon reaching the iteration limit. Idealized `BatchPDLP` obtained the fastest result in 65% of benchmark test set problems.

5.5.2 Iteration-Limited `BatchPDLP`

To demonstrate the performance impact of even a small number of subproblems with high iteration counts, a second performance profile is shown in Figure 5.4. Here, `BatchPDLP` is given the unfair advantage of preemptive knowledge about the iteration count profile for each problem, which is utilized to lower its internal iteration limit as far as possible while still allowing at least 95% of problems to terminate normally. This modification specifically reduces the impact of the numerically difficult problems while still obtaining unimpacted results for at least 95% of the LP subproblems, and consequently we refer to this as an *Idealized `BatchPDLP`*. With this foreknowledge about which problems will be most difficult, `BatchPDLP` sees significantly improved performance, now achieving the fastest solve time in 65% (80/124) of problems and more favorable comparisons against the alternative solvers in general. It is important to note here that the comparison solvers are still solving all 10^4 subproblems; however, while `BatchPDLP` is solving at most 5% fewer subproblems, it gains considerably more speed than the small reduction in subproblem count would suggest. For example, on the `mhw4d` problem from Section 5.5.1, `BatchPDLP` obtains solutions roughly 600x faster than before, with a new total solve time 93x faster than Gurobi (versus 6.5x more time than Gurobi without this intervention). Using the $\geq$90%-solved metric, Idealized `BatchPDLP` correctly solves

91% (113/124) of problems, which is only a slight reduction from the base `BatchPDLP` result of 93% (115/124).

Running benchmarks with `BatchPDLP` modified in this manner serves two main purposes. First, it demonstrates that instances of slow performance of this batched solver are, to a large degree, attributable to a comparatively small number of subproblems dominating overall compute time. Second, this behavior may be exploitable specifically in DGO. E.g., although the final iteration counts cannot easily be predicted a priori, it is simple to maintain a running average of final iteration counts as individual LPs are solved. This information could then be used to preemptively terminate LPs if their iteration count exceeds some factor of the running mean. In cases where the iteration distribution is highly skewed, such as in `mhw4d`, this would allow `BatchPDLP` to identify and selectively stop those subproblems that negatively impact solver performance.

While choosing not to solve certain problems is unreasonable in most contexts, it may be acceptable in DGO. During typical B&B, if the lower bound of a particular node does not allow the node to be fathomed, it will be branched on and the two resulting nodes will need to be addressed with their own lower-bounding problems. If the B&B algorithm is utilizing this Idealized `BatchPDLP` setup, ignoring some LPs is equivalent to obtaining lower bounds that do not permit fathoming. That is, the nodes will be branched on and new LPs will eventually be created. Unnecessarily creating additional nodes creates more work for the global optimizer, however there may still be a net benefit if this choice results in an order-of-magnitude faster lower bounding. Alternatively, if additional branching is undesired, `BatchPDLP` could potentially be paired with an existing serial solver which can be used to address the small number of ignored LPs. In this way, GPU hardware could be used to quickly address most problems via `BatchPDLP`, and for those LPs that are numerically difficult and obstructive for parallelization, the CPU could be used.

5.5.3 Performance on Multiple GPUs

High-performance computing clusters frequently feature multiple GPUs, making it important to evaluate how the performance of `BatchPDLP` scales with the number of GPUs. For this assessment, the benchmark test set was re-run with `BatchPDLP` using two NVIDIA Quadro GV100 GPUs, each with 32 GB of HMB2 VRAM (driver v32.0.15.7657, CUDA v12.9.86). Figure 5.5 shows performance profiles comparing single- and dual-GPU versions of both the default `BatchPDLP` and Idealized `BatchPDLP` (i.e., solving 95%+ of subproblems by modifying the iteration limit). In the direct comparison using the default version of `BatchPDLP`, roughly half (59/115) of problems obtain a

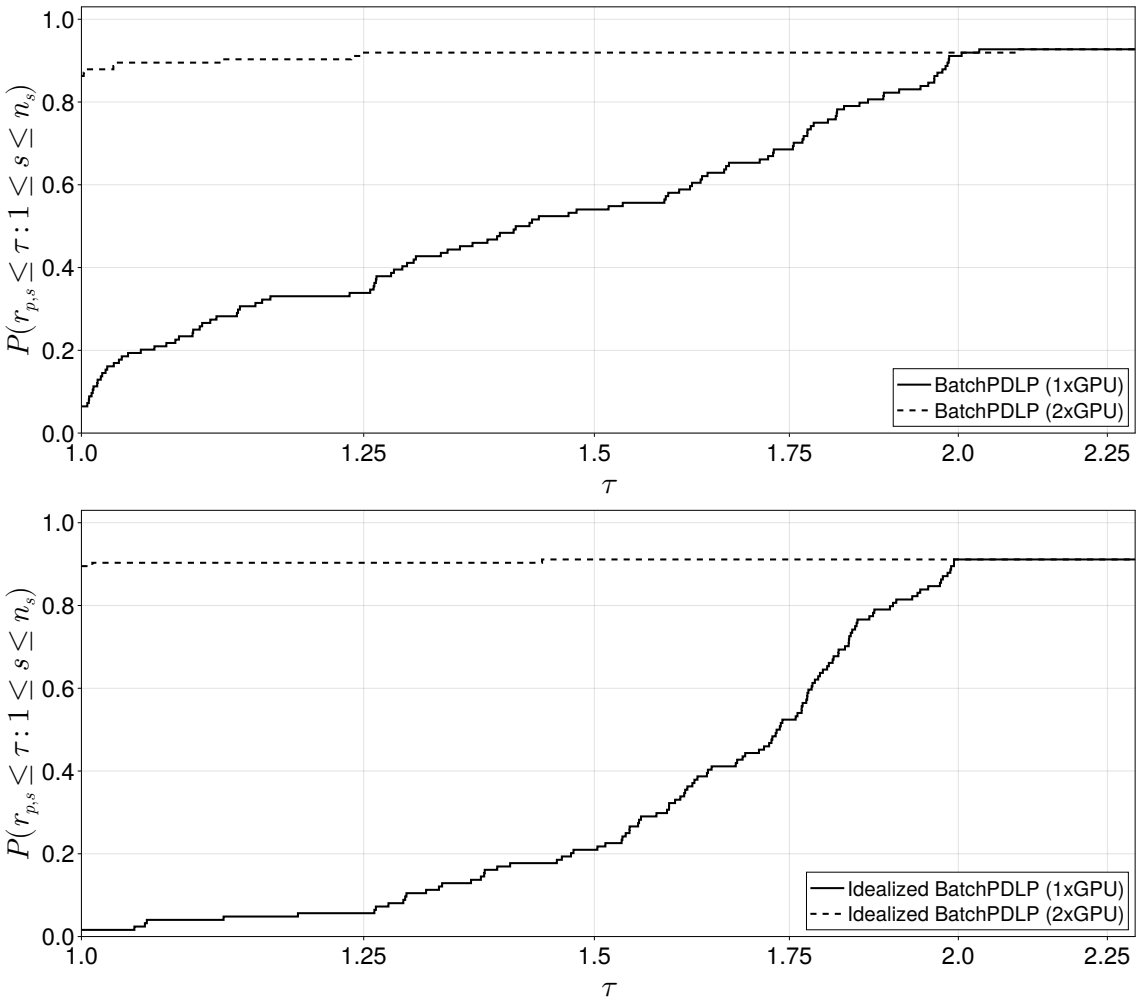

Figure 5.5: Dolan and Moré [39] performance profiles comparing the default and idealized forms of `BatchPDLP` (i.e., solving 95%+ of subproblems) run on both one and two NVIDIA GV100 GPUs. Without modifying iteration limits, running `BatchPDLP` on two GPUs yields a 35% or greater speedup on half of solved problems. When using Idealized `BatchPDLP`, two GPUs yields a 70% or greater speedup on half of solved problems, indicating that both the total runtime and the effectiveness of using multiple GPUs are strongly impacted by the presence of a minority of subproblems with abnormally high iteration counts.

speedup of at least 35% using two GPUs. However, we expect that in most cases, the runtime is dominated by those subproblems with the longest iteration counts (Sec. 5.5.2). The direct comparison using Idealized `BatchPDLP` validates this expectation: with the most time-consuming subproblems now iteration-limited, the dual-GPU setup obtains a speedup of at least 70% on roughly half (58/113) of all solved problems.

The use of two GPUs did not always provide a substantial speedup, and even when using Idealized `BatchPDLP`, a couple of problems took longer to solve than in the single-GPU case. Examination

of these slower problems revealed two scenarios that often caused poor dual-GPU performance: extremely fast problems, and problems with highly skewed iteration counts that were not effectively handled by the 95% cut-off approach. For the former case, we identified twelve problems for which single-GPU results for all 10^4 subproblems were obtained in under 20 ms. None of these twelve problems achieved a speedup above 38% when using two GPUs. The fastest single-GPU problem, `hs62`, with a total time of 10.4 ms, was one of the two problems for which two GPUs was slower, taking 14.9 ms for all subproblems. For these problems, the time scale is low enough to be greatly affected by background processes; rerunning the problem several more times yielded results as low as 9.1 ms for the dual-GPU case, which, although not twice as fast as the single-GPU case, still represents a slight speedup over the single-GPU case.

The other scenario that occasionally led to poor dual-GPU performance was when problems had more than 5% of subproblems with high iteration counts. In the `maxmin` problem, for example, 840 of the 10^4 subproblems required over 10^4 iterations to terminate, but by the idealized $\geq$95%-solved approach, the iteration limit was only reduced to roughly 1.6×10^5. While lower than 10^6, this reduction still allowed a significant portion of the overall runtime to be spent on a small number of long-running problems. Here, the single-GPU Idealized `BatchPDLP` run took 38.7 s, and the dual-GPU run took 39.0 s. For this problem—and others where few subproblems take most of the time—the benefit from using multiple GPUs depends on both effective load balancing and on LP execution order. To run 10^4 LPs on two GPUs, we split the set into two batches of 5×10^3 LPs without using any a priori information about LP iteration counts or overall runtime. As a consequence, if the LPs sent to one GPU are substantially more challenging than those sent to the other, there may be a period toward the end of the solve time where only one GPU is active. Separately, the execution order of individual LPs on GPUs is not guaranteed, which may lead to scenarios where an inefficient ordering is used.

It is also important to note that, given the challenging nature of many tested subproblems, the number of iterations required was not always guaranteed to be the same across different runs. This was particularly pertinent in the `kriging_peaks-full020` problem where we would sometimes observe either the single- or dual-GPU case taking nearly 15x longer than the other. A close examination of the benchmark data revealed one especially sensitive subproblem that only sometimes reached the iteration limit of 10^6. The cause of this stochasticity is likely an especially sharp subproblem where changes in the final floating-point digit change the trajectory of the PDLP algorithm. To not misrepresent the general performance of `BatchPDLP` in comparison to other solvers, the `kriging_peaks-full020` problem was re-run for all solvers and solver methods. In this new

benchmark instance, neither the single- nor dual-GPU runs had any subproblems reach the iteration limit.

5.6 Conclusion

The specific needs of GPU acceleration and solutions to many low-dimensional LPs necessitated the development of a novel LP solver. `BatchPDLP` effectively addresses these needs by rearchitecting the PDLP algorithm to operate in a batched fashion, utilizing blocks of GPU resources to solve sets of similar LPs. On a benchmark test set of LP subproblems generated from NLPs that might be addressed by a global optimizer, `BatchPDLP` performed competitively with leading commercial and open-source LP solvers, achieving the fastest speed in 48% of test set problems using default settings, and 65% of problems using a problem-specific iteration limit.

The targeted implementation of `BatchPDLP` limits its generality as compared to other commercial or open-source solvers. Particularly, `BatchPDLP` cannot easily handle batches of LPs with different numbers of variables or constraints, batches of LPs with different sparsity patterns, or individual large LPs. This novel implementation is also especially susceptible to small numbers of numerically difficult problems within one batch, and has no included presolving techniques that may help to address these occasional problems. Applying such presolving techniques to individual LPs without, for example, disrupting the overall sparsity pattern remains an area of future research.

Chapter 6

Integration of GPU-Parallel McCormick Relaxations and Batched PDLP for Accelerated Deterministic Global Optimization

Deterministic global optimization using branch-and-bound (B&B) is notoriously computationally expensive, but mature solvers predominantly rely exclusively on CPU hardware. Graphics processing units (GPUs) offer more compute throughput and greater energy efficiency than CPUs and are often relied upon for high-performance computing, but GPU use for B&B is still nascent. In the previous chapters, we developed methods for evaluating McCormick relaxations (Ch. 3–4) and solving batches of structurally similar LPs (Ch. 5). In this chapter, we integrate these previously developed methods to create ARION: a GPU-accelerated B&B method. ARION is fully open-source and is built as an extension of our CPU-based solver EAGO. Internally, it applies GPU hardware to address multiple lower-bounding problems simultaneously. We demonstrate the performance of ARION against other commercial and open-source solvers on a set of NLPs from the MINLPLib database and a dimension-scalable test problem. ARION consistently outperforms EAGO in the benchmark test set, especially on problems that took EAGO longer than 10 s, and solved higher-dimensional instances of the scalable test problem than any other tested solver.

6.1 Introduction

Deterministic global optimization (DGO) methods are necessary to solve to ϵ-global optimality nonconvex nonlinear programs (NLPs) of the form:

$$\begin{aligned} f^* = \min_{\mathbf{x}} \; & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \mathbf{x} \in X \subset \mathbb{R}^n. \end{aligned} \tag{6.1}$$

However, nonconvex NLPs are NP-hard [65], which manifests as worst-case exponential scaling even with state-of-the-art techniques. This presents a challenge for science and engineering applications where globally optimal solutions are necessary for tasks such as model verification [127] or robust control [85] and where problems of practical interest are typically nonconvex due to the need for phenomenologically accurate modeling. Due to the high computational cost of DGO, many such problems remain unsolvable in a practical amount of time, necessitating new developments to improve overall solution speeds.

One feature shared by most mature implementations of DGO solvers is their exclusive use of CPU hardware. Although modern CPUs are powerful and flexible hardware, graphics processing units (GPUs) outcompete them in select, repetitive, highly compute-intensive tasks such as machine learning model training [3], large-scale data analysis [128], and graphics rendering, among many others. GPUs excel at these tasks due to their highly parallelized internal architecture, which also generally gives them greater energy efficiency per calculation than CPUs [100]. Additionally, GPUs comprise a significant portion of the compute power in high-performance computing (HPC) clusters [2], which implies that effective HPC use requires GPU-enabled software. However, by default, algorithms written for CPU operation are not expected to be performant on GPU hardware. To make use of GPUs, algorithms may require a redesign to better exploit the parallel nature of the hardware. Although there have been DGO implementations that made use of different forms of CPU parallelization [68, 72, 124, 125], only a handful of GPU-based implementations exist, which to our knowledge comprises one by the present authors [58] and a more recent implementation by Zhang et al. [160]. By making use of faster and more efficient hardware, GPU-based B&B algorithms aim to outcompete conventional DGO methods in terms of both speed and cost and enable the use of HPC resources to solve especially challenging nonconvex NLPs.

Fundamentally, DGO algorithms rely on developing feasible lower and upper bounds of the objective function of (6.1) on elements of an increasingly fine partition of the original problem space. A standard lower-bounding technique used by global solvers, including BARON [119, 161], SCIP [19, 145], MAiNGO [22], and EAGO [155], is to create a linear program (LP) that is a *relaxation* of (6.1) on an element of interest in the partition. By construction, a solution of this *relaxed LP* is a lower bound of the objective function of (6.1) for that partition element, and as the partition gets progressively finer, the gap between the solution of the relaxed LP and the true feasible minimum of the objective function of (6.1) on that partition element approaches zero. By applying this process to all partition elements, a global solution (if one exists) will eventually be identified in finitely-many iterations. One clear way to adapt this methodology to GPU operation is to use or create GPU-accelerated methods of generating and solving relaxed LPs. From our previous work, we have developed the `SourceCodeMcCormick.jl` [53, 55, 58] package for evaluating McCormick relaxations [95, 99, 121] using GPUs, and the `BatchPDLP.jl` [51, 56] package for solving batches of small-scale LPs on GPUs. We aim to use these components to evaluate lower bounds for multiple partition elements simultaneously on one or more GPUs, while maintaining the remaining portions of the algorithm in their original, serial, CPU-based forms.

In this chapter, we develop a novel and efficient GPU-accelerated DGO solver that relies on the combination of our two recent advances in constructing and solving relaxed LPs. We identify this solver based on its design as an **A**ccelerated and **R**eadily scalable **I**ntegrated CPU-GPU **O**ptimization for **N**onlinear programs (ARION). Built as an extension of EAGO, ARION can address practically all NLPs that can be solved by the base version of EAGO, but with faster speeds on challenging problems and easier scaling to additional computational resources. The remainder of this chapter is structured as follows. Section 6.2 provides the necessary DGO background information. In Section 6.3, we provide implementation details of our novel approach, including our use of the `SourceCodeMcCormick.jl` and `BatchPDLP.jl` packages. Numerical benchmarks are run and discussed in Section 6.4. Section 6.5 provides a summary of our developments and plans for future work.

6.2 Background

6.2.1 Spatial Branch-and-Bound

The nonconvex NLP (6.1) can be solved to ϵ-global optimality using the spatial branch-and-bound (B&B) algorithm described in Section 2.6.1. As further described in Section 2.6.1, the implementation of the spatial B&B algorithm in EAGO is infinite but convergent. As we observe in this earlier discussion, in practice, Steps 2 through 4 of Algorithm 1 are typically iterated on for each node i, but this choice is not a requirement [58]. In principle, multiple partition elements may be addressed simultaneously, which we exploit in both this and our earlier work [58] to make use of GPU hardware.

6.2.2 Lower-Bounding Linearization Point Selection

As noted in the Section 6.1, a standard lower-bounding technique is to create a relaxed LP based on (6.1) [104]. Relaxed LPs are typically constructed using subgradients of relaxations of the objective and constraints of (6.1), with the relaxations evaluated at one or more *linearization points* [104]. One of the challenges associated with this task is determining an effective way of selecting linearization points [104]. If high-quality points are selected, the resulting relaxed LP will provide a superior lower bound than if lower-quality points are chosen, but the quality of points is generally not known a priori [104]. One valid approach is to calculate many linearization points with the hope that some would be of higher quality; however, using a larger number of points and adding more constraints to the relaxed LP may waste computational effort, both from the calculation of more relaxation subgradients and the effort required to solve LPs with unnecessarily many constraints. Solvers balance these effects heuristically, hopefully obtaining reasonably small sets of promising linearization points, such as by using bundle method-inspired approaches from convex optimization literature [104].

The EAGO solver, which we use as a platform for ARION, uses an adaptation of Kelley's algorithm [74], wherein the center of the node is selected as a linearization point, and the resulting relaxed LP is solved [155]. The solution of this LP is used to identify a second linearization point, with new constraints added to the relaxed LP. This process continues until either the objective value of the solved, relaxed LP stops improving or the preselected maximum number of iterations is reached. An alternative method described by Najman et al. [104] involves preselecting points at the corners of an n-simplex prior to running B&B, followed by scaling and shifting those points to the

node subdomain and using them as linearization points during the lower-bounding problem. This approach conveniently identifies linearization points once, rather than separately for each node, and was found to have comparable performance to Kelley's algorithm in practice [104].

6.3 Software Development

This section details the development of ARION, our novel B&B implementation that is designed to exploit GPU hardware architecture. As a high-level summary, we modify the structure of the B&B algorithm implemented in EAGO to allow multiple nodes' lower bounds to be calculated simultaneously on the GPU, but continue to utilize the serial, CPU-based stack management, pre- and post-processing techniques, and upper-bounding routines already present in EAGO. For the lower-bounding problem, we generate relaxations for all nodes simultaneously in a batch using `SourceCodeMcCormick.jl` (`SCMC`) [53, 55, 58] and then simultaneously solve the resulting relaxed LPs using, predominantly, `BatchPDLP.jl` (`BatchPDLP`) [51, 56]. Additional details on the modified B&B algorithm and parallelized lower-bounding problem are discussed in the following subsections.

6.3.1 Modified B&B Algorithm

The concept shared by the component packages `SourceCodeMcCormick.jl` and `BatchPDLP.jl` is that while the lower-bounding problems for single nodes may be too small to effectively use GPU resources, a sufficiently expensive task can be created by grouping lower-bounding problems together. In this paradigm, other typical B&B tasks such as the upper-bounding routine do not necessarily require GPU parallelization, which allows ARION to make use of CPU-based routines already extant within EAGO. This mix of serial and parallel node processing necessitates a customized B&B approach, represented visually in Figure 6.1. Structurally, ARION is similar to the `ParBB` algorithm presented in Gottlieb et al. [58], though there are several important differences, particularly with regard to start-up processes and upper-bounding.

The B&B algorithm generally begins with a single root node, which is troublesome for GPU approaches because there may not be enough work to saturate the compute capacity of the hardware. In `ParBB`, this problem was handled by immediately branching the root node into suitably many branches so that the GPU could be used more efficiently [58]. However, and as observed by Zhang et al. [160], this approach may create a large number of unnecessary nodes that can slow the overall solution time. An alternate approach utilized by Zhang et al. [160] was to consider only one node at a time and instead to use GPU parallelization to obtain tighter lower bounds by performing

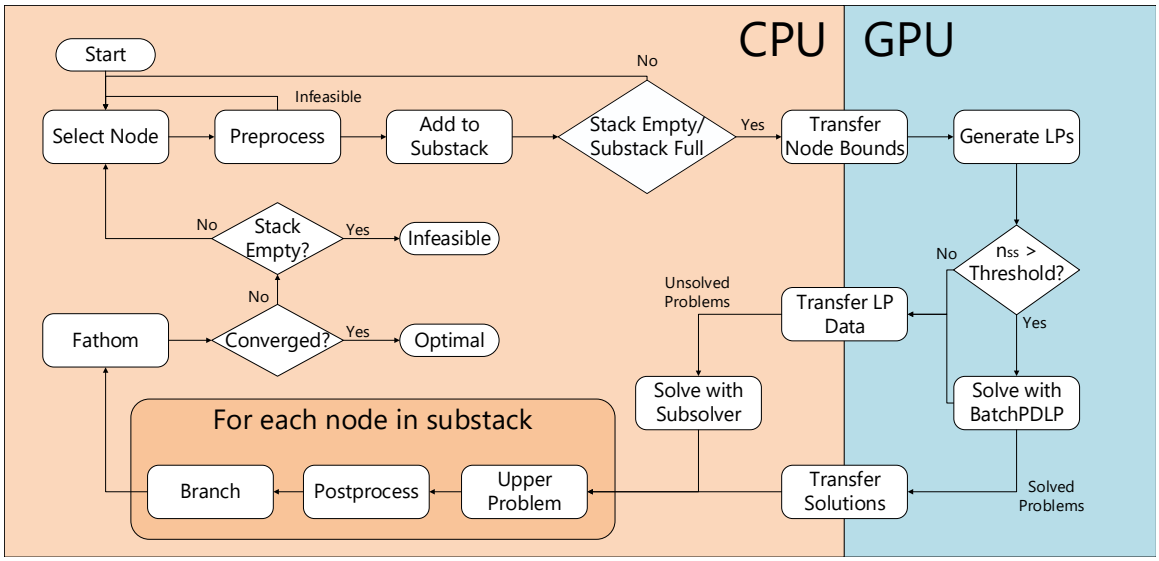

Figure 6.1: A flow diagram is illustrated for the hybrid B&B approach employed by ARION. The lower-bounding task, comprising the generation and solution of LPs, occurs predominantly on the GPU, while all other tasks occur on the CPU using routines from the EAGO solver. GPU parallelization is achieved through the use of a substack which collects information for multiple B&B nodes before passing them to the GPU for simultaneous processing.

more work on each node. Since individual nodes are still processed sequentially, this approach can immediately utilize the full capacity of the GPU on the root node and does not require modifying the B&B tree. For ARION, we employ the same batched node processing of `ParBB` but use no special start-up routine. ARION simply begins processing at the root node, accepting the lower efficiency of running our tasks on a batch size of one for the first iteration—though, as described later in Section 6.3.2, some processes are modified when the batch size is small. In general, due to the batch-processing nature of ARION, we expect that comparatively little time will be spent with a small number of nodes in the stack.

As shown in Figure 6.1, the transition between serial and batch processing is handled with the use of a substack. Nodes are selected and preprocessed serially using built-in EAGO routines, and if a node is not found to be infeasible by preprocessing, it is added to the substack. Nodes are selected from the main stack using a best-first (lowest lower bound) strategy, just as in EAGO. Once the main stack is exhausted or the substack is full, the lower-bounding routine—i.e., the boxes starting at "Transfer Node Bounds" and ending immediately prior to "Upper Problem" in Figure 6.1—is called to process all of the substack nodes simultaneously, with the individual processes described in Section 6.3.2. Following the lower-bounding routine, nodes are pulled from the substack serially, connected with their lower-bounding result, and sent to EAGO's built-in upper-bounding, post-processing, and

(if necessary) branching routines.

Due to the batch-processing style used by ARION, a different B&B tree will be created than what is made by EAGO, despite the re-use of EAGO's node selection routines. Specifically, if the substack can hold n_{ss} nodes, ARION is effectively following a best-n_{ss}-first strategy, which implies that at most $n_{ss} - 1$ nodes that did not have the lowest lower bounds will be processed and branched on in each iteration. This raises a potential disadvantage of the parallelization scheme used by ARION, in that time may be spent processing and branching on nodes that would have been fathomed if processing were performed serially. Additionally, this strategy implies that over the course of one iteration, where n_{ss} nodes are processed, each node can only be branched on once. This is in contrast to serial B&B, where over the course of n_{ss} iterations (each processing one node), a single node and its children may be branched on a large number of times if they consistently have the lowest lower bounds. While these are notable concerns, we expect that the time spent processing nodes unnecessarily will be low in practice, but greater in cases where the upper bound changes frequently as there are more opportunities for existing nodes to be fathomed. In the opposite case, where a globally optimal upper bound is identified early (but still needs to be verified with DGO), there should be no unnecessary node processing, as any non-converged node in the main B&B stack necessarily requires more processing, even in serial B&B.

Considering the convergence properties of this modified B&B algorithm, we note that selecting the n_{ss} best nodes implies that at least one node in each iteration will have the global lower bound. This ensures that the selection operation in ARION is bound improving. As will be described in greater detail in Section 6.3.2, the lower-bounding routine in ARION operates on each node independently, and we continue to use McCormick relaxations as in EAGO. This ensures that the bounding operation is consistent, and in combination with the bound improving selection operation, this guarantees that ARION is convergent. We also note that the steps of node selection, branching, and fathoming occur serially using built-in EAGO routines, which prevents ARION from encountering possible race conditions and ensures that data cannot arrive out of sequence, which could adversely affect the algorithm.

6.3.2 Lower-Bounding Algorithm

One of the main novelties of this work is the integration of the `SCMC` and `BatchPDLP` component packages into a new GPU-accelerated lower-bounding routine. The process for each B&B node requires the creation and solution of a relaxed LP, and by using these component packages, we can

execute these tasks for many nodes in parallel on GPU hardware. The following Sections 6.3.2 and 6.3.2 describe the specific uses of `SCMC` and `BatchPDLP`, respectively.

Constructing Relaxed LPs

As described in Section 6.2.2, the EAGO solver uses an adapted form of Kelley's algorithm for its lower-bounding procedure. Unfortunately, applying this method to batches of nodes is likely to be inefficient in ARION for two primary reasons. First, each iteration of this version of Kelley's algorithm requires only one evaluation of a relaxation subgradient per node, which is an inefficient way to use the mass-parallelization capabilities of `SCMC`, as the relative performance increases with the requested number of evaluations [55]. Second, ARION processes batches of nodes in parallel, but the stopping condition of Kelley's algorithm is node-dependent. If each node can independently decide to terminate Kelley's algorithm early, there will certainly be cases where only a small number of nodes in the batch require additional iterations, which would drastically reduce the effectiveness of the node parallelization.

The n-simplex method proposed by Najman et al. [104] may be a more favorable alternative to Kelley's algorithm. Adapting the n-simplex method to batched node processing on GPUs has the advantage of ensuring that each node goes through the same process without the early termination issue that would appear in a GPU-adapted Kelley's algorithm, and it ensures that more relaxation subgradients are calculated at once, which lowers the per-relaxation evaluation time of `SCMC` [55]. However, early testing revealed that `SCMC` is capable of evaluating many more points per node than what is required by the n-simplex approach without a significant rise in the relaxation evaluation time. To use `SCMC` to a greater extent, more linearization points per node are desired. For instance, more points could be obtained by using a space-filling approach such as a Sobol sequence [130], though this strategy may not scale well as problem dimensionality increases. Additionally, while evaluating relaxation subgradients may be fast, the increase in the number of relaxed LP constraints may negatively impact the LP solution time.

Consequently, in this work, we introduce a novel bundle method specifically adapted to the parallelization strength of `SCMC`, displayed visually in Figure 6.2 and described in Algorithms 6 through 8. At a high level, this method calculates many candidate linearization points in parallel and is then selective about which points are ultimately used in the relaxed LPs. The speed of `SCMC` allows the calculation of candidate points to not be a computational bottleneck, and using only a portion of the calculated points limits the number of relaxed LP constraints. Selecting from sets of calculated points also provides a unique benefit in that the linearization points used for the objective

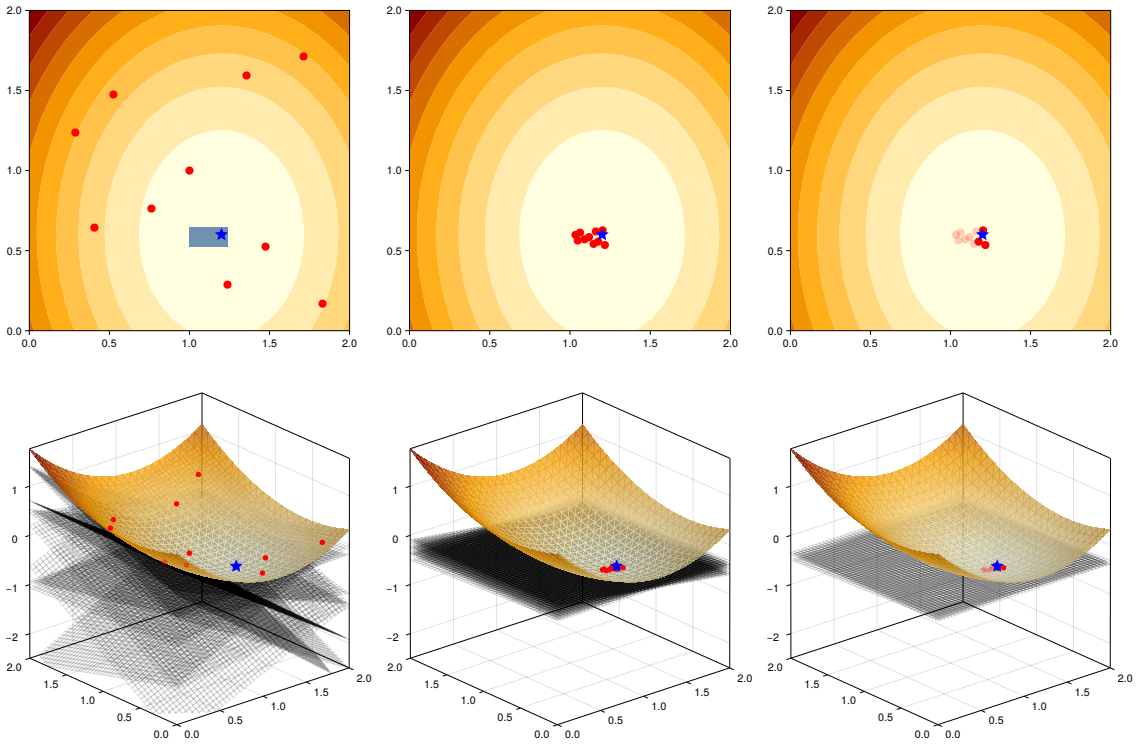

Figure 6.2: A visualization of the Algorithm 6 for the objective function $0.6\sin(x\pi)\sin(1.5y\pi) + 0.8(x-1.2)^2 + 0.5(y-0.9)^2 + 0.3y$ over $[0,2]\times[0,2]$ with $n_{\text{Sobol}} = 10$. (Left) An initial set of points is obtained from a Sobol sequence [130] (red) and the convex relaxation of the objective function (orange surface) is evaluated at each point in the set to yield subtangent hyperplanes (black). Subgradient information is then examined to predict a region of the convex relaxation that likely contains a minimum (blue rectangle). The true minimum is shown as a blue star. (Middle) The set of points from the Sobol sequence is scaled and shifted to cover the region identified as possibly containing a minimum. The convex relaxation of the objective function is evaluated at each scaled and shifted point to yield a new set of subtangent hyperplanes. (Right) From the set of subtangent hyperplanes in the previous step, three hyperplanes are selected to contribute to the relaxed LP, with other hyperplanes discarded. Relaxed LPs formed from the subgradients shown in the {bottom-left, bottom-mid, bottom-right} have solutions of $\{-0.4506, -0.3759, -0.3759\}$, compared to the true minimum of -0.3750.

function of (6.1) do not necessarily need to be the same as those used for each constraint. As a result, we use different algorithms for objective function linearization points (Algs. 6, 7) than for constraint linearization points (Alg. 8) to account for the different roles each type of expression plays in the lower-bounding problem. As with the other linearization point selection methods for GPU use, this approach is applied separately to each node being evaluated in a batch.

The goal of Algorithms 6 and 7 is to identify and add high-quality LP constraints to each relaxed LP based on convex relaxations of the objective function of (6.1). In the context of a lower-bounding problem, barring the influence of constraints in (6.1), we hypothesize that the highest quality lin-

Algorithm 6 MultiSobol Linearization Point Selection Process

Require: The objective function ξ from the original NLP
Require: The node subdomain $X \in \mathbb{IR}^n$
Require: The first n_{Sobol} points of a Sobol sequence $\sigma_j \in [0,1]^n,\ \forall j = 1, \cdots, n_{\text{Sobol}}$, collectively referred to as $\boldsymbol{\sigma}$

Obtain tighter lower and upper subdomain bounds, $[\mathbf{l}, \mathbf{u}] \in \mathbb{I}X$:
    $\mathbf{l},\ \mathbf{u} = \texttt{MultiSobolScan}(\xi, \boldsymbol{\sigma}, n_{\text{Sobol}}, X)$
Scale $\boldsymbol{\sigma}$ to $[\mathbf{l} + \delta, \mathbf{u} - \delta]$, where δ is $2.5\% \times (\mathbf{u} - \mathbf{l})$
Evaluate McCormick relaxations of ξ on X and their subgradients at all points σ_j, yielding $\{\xi_j^{\text{cv}}, \xi_j^{\text{cc}}, \xi_j^L, \xi_j^U, \mathbf{s}_j^{\text{cv}}, \mathbf{s}_j^{\text{cc}}\}$
Store relaxations and subgradients in a set $\boldsymbol{\Omega}$ such that:
    $\Omega_j = \{\xi_j^{\text{cv}}, \xi_j^{\text{cc}}, \xi_j^L, \xi_j^U, \mathbf{s}_j^{\text{cv}}, \mathbf{s}_j^{\text{cc}}\}$
Set d as the number of unique variables that appear in the expression of ξ
for $i = 1, \cdots, d+1$ **do**
    if $\boldsymbol{\Omega}$ is nonempty **then**
        Select Ω_j such that $\text{argmin}_j \sum_{i=1}^n (s_{j,i}^{\text{cv}})^2 \times (u_i - l_i)$
        Add an LP constraint that the epigraph variable lies above the subtangent hyperplane c_j described by ξ_j^{cv} and $\mathbf{s}_j^{\text{cv}}$:
            $c_j(\mathbf{z}) = \xi_j^{\text{cv}} + (\mathbf{s}_j^{\text{cv}})^\top (\mathbf{z} - \sigma_j)\ \forall \mathbf{z} \in \mathbb{R}^n$
        Remove from $\boldsymbol{\Omega}$ all Ω_k where the signs of values $\mathbf{s}_k^{\text{cv}}$ match the signs of values $\mathbf{s}_j^{\text{cv}}$
end for

earization points for the objective function are those where the subgradients of the convex relaxation of the objective function evaluated at those points have the lowest magnitude. We consider these points to be the most valuable because, all else equal, a point with a lower-magnitude subgradient in one dimension generally provides tighter information about a minimum of a convex function than another point with a higher-magnitude subgradient in that dimension. With this criterion, we also assume that higher-quality linearization points are likely to be found close to a minimum of the convex relaxation. In low dimensions, a space-filling approach, such as a Sobol sequence, may identify several points close to a minimum that may serve as high-quality linearization points. Unfortunately, as the dimensionality increases, even a large number of points may fail to stumble upon a minimum. The linearization point selection process described by Algorithms 6 and 7, therefore, first attempts to quickly reduce the search space to find a small region that likely contains a relaxation minimum on a node subdomain, and then screens linearization points within the reduced search region to identify high-quality points to become LP constraints.

The initial search is described by Algorithm 7 and is represented visually in Figure 6.2 (left). In this process, n_{Sobol} pre-calculated Sobol points are scaled to fit within the node subdomain, and the convex relaxation (and its subgradients) of the objective function of (6.1) is evaluated at those points. As shown in Figure 6.2 (bottom-left), many of these points are unlikely to be close to a relaxation minimum and therefore provide poor subgradient information. Rather than construct a

Algorithm 7 MultiSobol Scan Procedure

function `MultiSobolScan`$(\xi, \boldsymbol{\sigma}, n_{\text{Sobol}}, X)$
  Initialize $\mathbf{l}, \mathbf{u}$ such that $[\mathbf{l}, \mathbf{u}] = [\mathbf{x}^L, \mathbf{x}^U]$
  Scale $\boldsymbol{\sigma}$ to $[\mathbf{l} + \delta, \mathbf{u} - \delta]$, where δ is $2.5\% \times (\mathbf{u} - \mathbf{l})$
  Evaluate McCormick relaxations of ξ on X and their subgradients at all points σ_j, yielding $(\xi_j^{\text{cv}}, \xi_j^{\text{cc}}, \xi_j^L, \xi_j^U, \mathbf{s}_j^{\text{cv}}, \mathbf{s}_j^{\text{cc}})$
  for $i \in 1, \cdots, n$ **do**
    for $j \in 1, \cdots, n_{\text{Sobol}}$ **do**
      if $s_{j,i}^{\text{cv}} < 0$ **and** $\sigma_{j,i} > l_i$ **then**
        $l_i := \sigma_{j,i}$
      if $s_{j,i}^{\text{cv}} > 0$ **and** $\sigma_{j,i} < u_i$ **then**
        $u_i := \sigma_{j,i}$
    end for
    if $l_i > u_i$ **then**
      $l_i,\ u_i := u_i,\ l_i$
  end for
  return $\mathbf{l}$, $\mathbf{u}$
end

relaxed LP with these points, Algorithm 7 proceeds by comparing these subgradients against each other to find a region in which the signs of subgradient coefficients change. Such a region can be visualized as the blue rectangle in Figure 6.2 (top-left). For more complex cases than what is pictured, it is possible that this simplistic approach will miss a true minimum, though we expect that a minimum will not be far outside the predicted domain.

Once the minimizing region is identified, the main portion of Algorithm 6 may proceed. First, the set of Sobol points from the previous step is scaled and shifted to cover the predicted minimizing region rather than the entire node subdomain, and the convex relaxation and its subgradients are re-evaluated as these new points. This process can be seen in the center subplots of Figure 6.2. It is important to note here that McCormick relaxations in general are dependent on the domain on which they are evaluated, and that although the Sobol points have been shifted and scaled to a new, smaller subdomain, the convex relaxation is still evaluated using the original node bounds. This process is designed to identify improved linearization points, but the new subtangent hyperplanes must still be valid for the relaxation on the original subdomain. Second, following relaxation and subgradient evaluations at all candidate linearization points, up to $d + 1$ points are selected as final linearization points using the criteria in Algorithm 6 (where d is the number of unique variables that appear in the objective function of (6.1)). This process can be seen in Figure 6.2 (right) and ensures that even if n_{Sobol} is large, the resulting relaxed LPs do not have needlessly-many constraints.

Although this process is effective for the objective function of (6.1), constraints in (6.1) require a different treatment. This is because the lower-bounding procedure attempts to minimally underesti-

Algorithm 8 Sobol Linearization Point Selection Process for Constraints

Require: One "$\leq$" or "$\geq$" constraint ξ from the original NLP
Require: The first n_{Sobol} points of a Sobol sequence $\sigma_j \in [0,1]^n,\ \forall j = 1, \cdots, n_{\text{Sobol}}$
Require: The node subdomain $X \in \mathbb{IR}^n$
1: $\mathbf{l},\ \mathbf{u} = [\mathbf{x}^L, \mathbf{x}^U]$
2: Scale $\boldsymbol{\sigma}$ to $[\mathbf{l} + \delta, \mathbf{u} - \delta]$, where δ is $2.5\% * (\mathbf{u} - \mathbf{l})$
3: Evaluate McCormick relaxations of ξ on X and their subgradients at all points σ_j, yielding $\{\xi_j^{\text{cv}}, \xi_j^{\text{cc}}, \xi_j^L, \xi_j^U, \mathbf{s}_j^{\text{cv}}, \mathbf{s}_j^{\text{cc}}\}$
4: Store relaxations and subgradients in a set $\boldsymbol{\Omega}$ such that:
$\Omega_j = \{\xi_j^{\text{cv}}, \xi_j^{\text{cc}}, \xi_j^L, \xi_j^U, \mathbf{s}_j^{\text{cv}}, \mathbf{s}_j^{\text{cc}}\}$
5: Set d as the number of unique variables that appear in the expression of ξ
6: **for** $i = 1, \cdots, d+1$ **do**
7: **if** $\boldsymbol{\Omega}$ is nonempty **then**
If ξ is a "$\leq$" constraint
8: Select the Ω_j that minimizes abs(ξ_j^{cv})
9: Add an LP constraint using the subtangent hyperplane described by ξ_j^{cv} and $\mathbf{s}_j^{\text{cv}}$
10: Remove from $\boldsymbol{\Omega}$ all Ω_k where the signs of values $\mathbf{s}_k^{\text{cv}}$ match the signs of values $\mathbf{s}_j^{\text{cv}}$
If ξ is a "$\geq$" constraint
11: Select the Ω_j that minimizes abs(ξ_j^{cc})
12: Add an LP constraint using the subtangent hyperplane described by ξ_j^{cc} and $\mathbf{s}_j^{\text{cc}}$
13: Remove from $\boldsymbol{\Omega}$ all Ω_k where the signs of values $\mathbf{s}_k^{\text{cc}}$ match the signs of values $\mathbf{s}_j^{\text{cc}}$
14: **end for**

mate the minimum of a convex relaxation of the objective function, but the search area is constrained to a region delineated by the zero level-sets of relaxations of the constraints. Unlike zero-sublevel sets of convex functions, zero-level sets of convex functions are not typically convex. An analogous version of Algorithm 7 for constraints of (6.1) may attempt to identify a subdomain containing the zero-level set of a constraint relaxation, but the benefit may not be worth the added computational cost. Rather than use the same process, ARION uses a simpler approach for constraints of (6.1) that is described by Algorithm 8. Effectively, Algorithm 8 skips the scanning and rescaling of Algorithm 6—akin to skipping the subgradient recalculation seen in Figure 6.2 (center)—and uses a different selection metric to pick between candidate linearization points. This algorithm is written for inequality constraints in (6.1); in the case of equalityconstraints, ARION considers the constraint as pairwise inequalities (i.e., $\{\mathbf{h}(\mathbf{x}) \leq \mathbf{0}, \mathbf{h}(\mathbf{x}) \geq \mathbf{0}\}$) and therefore applies Algorithm 8 twice, possibly allowing for different linearization points for the convex and concave relaxations of the constraint.

Much like the n-simplex method of Najman et al. [104], Algorithms 6 through 8 allow the n_{Sobol} points to be calculated once, prior to running B&B, as these points can be rescaled to any subdomain of interest as needed. Similarly, Algorithms 6 through 8 have a benefit for GPU use over Kelley's method in that every node can be processed identically. By default, we select n_{Sobol}

to be 400, as this value performed well in preliminary testing. Consequently, Algorithm 6 requires a total of 800 relaxation evaluations for every node, which we recognize is considerably greater than what is needed for Kelley's method or the n-simplex method. However, the relative performance of `SCMC` improves as additional evaluations are requested [55], which allows even this large number of relaxation evaluations to not dominate the overall B&B runtime. Rather, we intentionally use a large number of points to develop relaxed LPs whose solutions underestimate the node's lower bound by a hopefully small amount relative to other methods, which contributes to faster node fathoming and thereby faster overall convergence. For the purposes of benchmarking, we implemented a GPU-adapted version of Kelley's algorithm, the method described by Algorithms 6 through 8, and a version of the previous method that skips the scan procedure of Algorithm 7.

LP Considerations

By addressing a large number of B&B nodes simultaneously, and by using `SCMC` as described in Section 6.3.2, ARION necessarily creates a large number of relaxed LPs that require solutions. This case of needing solutions for batches of LPs, ideally using GPU resources, motivated the development of our `BatchPDLP.jl` package [51], which we incorporate here as a component of ARION. As was noted in Gottlieb et al. [51], `BatchPDLP` is a competitive option as compared to existing commercial and open-source solvers, though it occasionally encounters numerical difficulties on individual LPs within batches, which strongly affect overall performance. In this section, we discuss how this tool can be used effectively while mitigating the impact of this issue.

One of the immediate benefits of using a custom LP subsolver is the ability to control solver termination. In this case, we are cognizant of the fact that the solutions to LPs may be used to fathom B&B nodes if the solution to the LP is greater than the current global upper bound identified by ARION. Consequently, because `BatchPDLP` utilizes dual information, we can apply duality theory to assert that if a dual-feasible value is obtained for an individual LP whose value is greater than the current global upper bound, the solution to that LP is guaranteed to also lie above the global upper bound. This information allows us to terminate the LP early, before it has fully converged, as we know the node will be fathomed regardless. In practice, this form of early termination does have some small impact, though the effect could be greater if the subsolver were tuned to quickly identify dual-feasible solutions or if, for example, a custom version of a dual simplex algorithm were used instead.

Separately, it is well known that first-order methods such as PDLP are particularly fast at identifying lower-quality solutions. For this reason, it is especially beneficial when using `BatchPDLP`

to be mindful of the convergence tolerance required for LP solutions. Since many global optimization benchmark problems described in Section 6.4 are typically solved to a tolerance of 10^{-3}, it should be sufficient in many cases to set an LP subsolver convergence tolerance of 10^{-6}, as opposed to a typical default of 10^{-8}. This adjustment is minimally impactful for methods that rely on simplex or interior-point methods, but we have found a modest speedup with this change due to the first-order nature of PDLP.

Another notable but important disadvantage of `BatchPDLP` is its comparatively slower speed when addressing only small numbers of LPs [51]. For this reason, although we aim to solve most LPs using this GPU-parallelized component, when the number of nodes per batch is lower than 500 (by default), LPs are solved using an alternate, serial LP solver such as GLPK (by default). This break point is also enforced when using the GPU-adapted Kelley's algorithm: if fewer than 500 nodes require additional iterations of the algorithm, the CPU-based solver is used for the remainder of the lower-bounding problem. In practice, we have found that on challenging problems where GPU use is warranted, the number of active nodes in the B&B stack quickly surpasses this limit and `BatchPDLP` becomes enabled.

One of the more critical aspects identified in Gottlieb et al. [51] was the susceptibility of BatchPDLP to individual numerically challenging LPs within batches. More specifically, `BatchPDLP` can solve many problems in every batch quickly, though the total time ultimately depends largely on the worst-case problems. To address this issue, ARION applies a unique approach wherein `BatchPDLP` is applied to all problems but with a dynamic iteration limit. As problems are solved by `BatchPDLP` in each batch, the average number of iterations required for each solution is tracked. After the first 100 problems per batch (by default) have been solved, all subsequent problems are subjected to a relatively strict termination limit of twice the average number of iterations. This average continues to be adjusted throughout each batch, so that if many problems are reaching the iteration limit, it will increase to allow future problems to have more time. However, by design and based on the iteration count patterns observed in Gottlieb et al. [51], this aggressive approach will likely cause at least a small number of LPs per batch to terminate before convergence. When problems are terminated early using this approach, ARION will send the remaining problems to the alternate, CPU-based solver. In effect, `BatchPDLP` is allowed to attempt to solve every LP, but its built-in dynamic limit prevents it from spending the majority of its runtime on a handful of problems. Instead, many problems are solved very quickly, which limits the work required for the alternate solver in terms of both setup time and solve time.

6.4 Numerical Benchmarking

This section presents the numerical benchmarks used to compare ARION against a set of other commercial and open-source DGO solvers including BARON, SCIP, and our CPU-based solver EAGO. All problems in this section were solved using one thread of an Intel Xeon W-2195 2.30/4.30 GHz (base/turbo) processor with 64 GB of RAM running a Windows 11 Enterprise 24H2 operating system and, for ARION, an NVIDIA Quadro GV100 GPU with 32 GB of HBM2 VRAM (driver v32.0.15.7657, CUDA v12.9.86). Julia v1.12.5 was used in conjunction with `CUDA.jl` v5.11.2 [16], `JuMP.jl` v1.30.1 [41], and GAMS v52.4.0 through `GAMS.jl` v0.5.4 [46]. Solver versions used were BARON v25.12.10 [119, 161], SCIP v10.0.0 through SCIP.jl v0.12.8 [19, 145], EAGO v0.9.2 [155], and ARION v0.1.0. The ARION solver was run in conjunction with `SourceCodeMcCormick.jl` v0.5.2 [55] and `BatchPDLP.jl` v0.1.0 [51]. Unless otherwise noted, each solver was run using its default subsolvers; options for each solver are presented in Table 6.1. For the ARION solver, problems were run using the algorithm described in Section 6.3.2 (marked as "MultiSobol"), a version of the algorithm in Section 6.3.2 without the scanning procedure of Algorithm 7 (marked as "MultiSobol w/o Scan"), and a GPU-adapted version of Kelley's algorithm (marked as "Kelley").

These solvers were first compared using problems from the MINLPLib benchmark library [1] satisfying the following criteria:

1. The problem type was listed as "NLP";
2. The number of variables did not exceed 15;
3. The operators used in the NLP did not include any of the following: {tanh, log10, gamma, mod, cvpower, errorf, signpower, centropy, rpower}, where:

$$\text{errorf}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-t^2}{2}} dt; \tag{6.2}$$

$$\text{signpower}(x, y) = \text{sign}(x)|x|^y, \text{ for } y > 0; \tag{6.3}$$

$$\text{centropy}(x, y[, z]) = \begin{cases} x \cdot \ln\left(\frac{x+z}{y+z}\right) & \text{if } x > 0,\ y > 0,\ z \geq 0 \\ 0 & \text{if } x = 0,\ y > 0,\ z \geq 0 \end{cases} \tag{6.4}$$

$$\text{rpower}(x, y) = \begin{cases} x^y & \text{if } x > 0 \\ 0 & \text{if } x = 0,\ y > 0 \\ 1 & \text{if } x = 0,\ y = 0. \end{cases} \tag{6.5}$$

Table 6.1: The solver options that were used to obtain the benchmarking results for each problem in this section (unless otherwise noted).

Solver	Parameters

BARON	`optca = 1E-3`
	`optcr = 1E-3`
	`reslim = 3600`
	`LogOption = 2`
	`JuMP.set_silent()`

SCIP	`display/verblevel = 0`
	`limits/time = 3600.0`
	`limits/gap = 1E-3`
	`limits/absgap = 1E-3`
	`JuMP.set_silent()`

EAGO	`time_limit = 3600.0`
	`log_on = true`
	`log_interval = 1000`
	`JuMP.set_silent()`
	Lower problem subsolver: Gurobi

ARION (MultiSobol)	(EAGO) `time_limit = 3600.0` (EAGO) `log_on = true` (ARION) `max_parallel_nodes = 10240` (ARION) `perform_scan = true` (ARION) `num_eval_points = 400` (ARION) `abs_tol = 1E-6` (ARION) `rel_tol = 1E-6` (ARION) `skip_hard_problems = true` (ARION) `cpu_solver = Gurobi.Optimizer()` `JuMP.set_silent()`
ARION (Kelley)	(EAGO) `time_limit = 3600.0` (EAGO) `log_on = true` (ARION) `max_parallel_nodes = 10240` (ARION) `abs_tol = 1E-6` (ARION) `rel_tol = 1E-6` (ARION) `skip_hard_problems = true` (ARION) `cpu_solver = Gurobi.Optimizer()` `JuMP.set_silent()`

With these restrictions, there were a total of 105 problems in the benchmark test set. However, of these problems, 14 featured triginometric terms that cannot be addressed by BARON. To ensure fairness, the results are therefore presented in two forms. First, we present results for the full suite of 105 problems, but with BARON excluded as a solver. Second, results are presented with BARON included, but with the 14 trignometric problems removed. Splitting the analysis in this way allows solvers that can address these problems to be compared against one another, without penalizing BARON based on the number of such problems in the test set. As another important note, on 18 problems, BARON terminated with a status of `JuMP.LOCALLY_SOLVED`, indicating that a globally optimal solution was not verified. These problems were rerun in BARON using the settings `optca =`

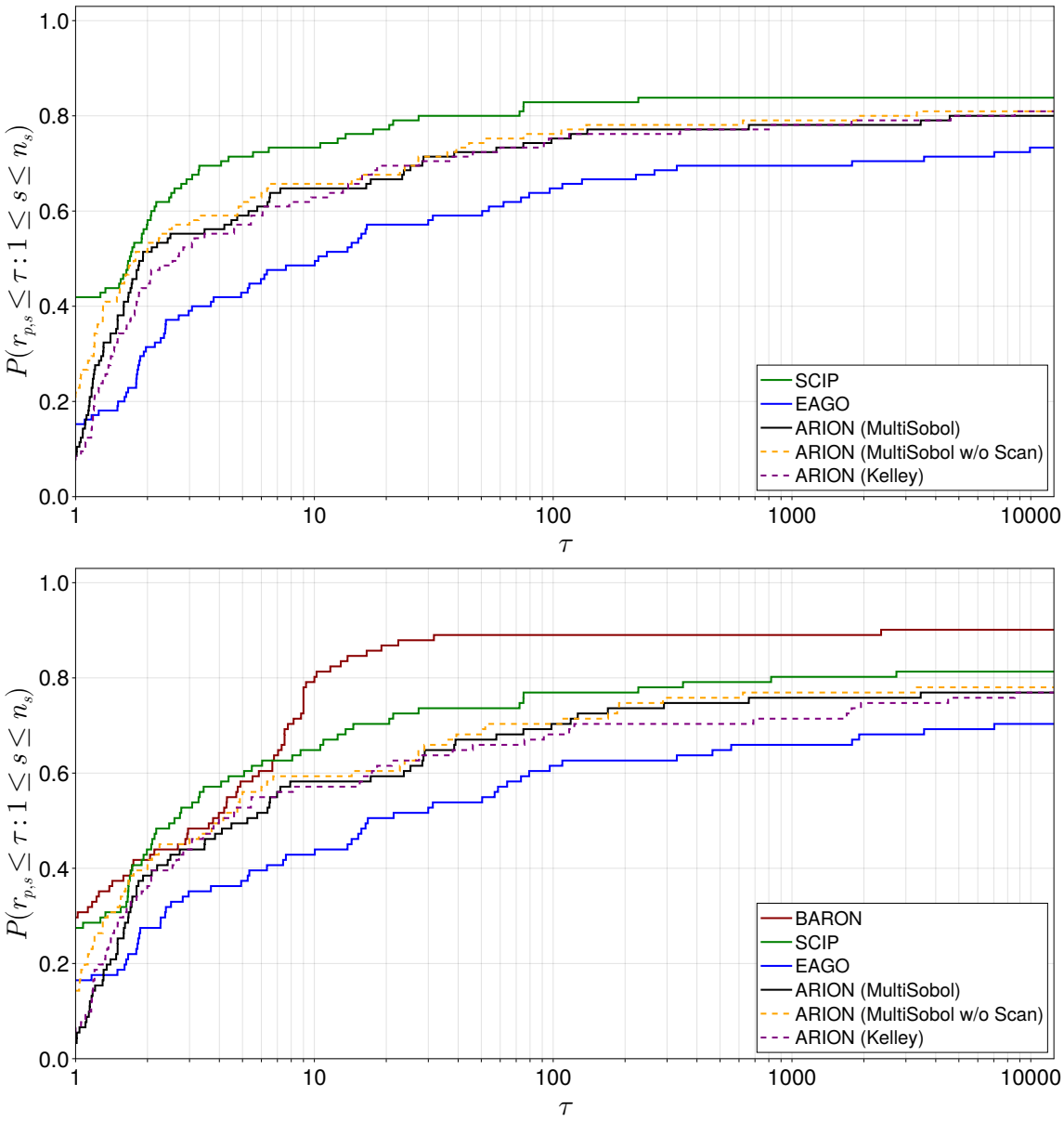

Figure 6.3: Dolan and Moré performance profiles [39] comparing BARON, SCIP, EAGO, and three forms of ARION—one using the method presented in Algorithm 6 (MultiSobol), one using Algorithm 6 without the scanning procedure of Algorithm 7 (MultiSobol w/o Scan), and one using a GPU-adapted version of Kelley's method (Kelley). Profiles are shown for (top) all benchmark problems, not including BARON, and (bottom) all problems not featuring triginometric functions, with the BARON solver included. In both profiles, the three versions of ARION are clear favorites over EAGO, solving more benchmark problems and reporting consistently faster times. Note that self-competition between different versions of ARION cause the profiles to appear lower than EAGO at $\tau = 1$, but that with any two ARION versions removed, the remaining ARION profile is entirely above the EAGO profile. On problems where EAGO is the fastest, the time reported by EAGO is always under 0.1 s, with times reported by other solvers generally under 1.0 s, indicating that it solves relatively easy problems very quickly. In comparison, ARION (MultiSobol) reported four fastest times above 1.0 s, including a 12.9 s solve on problem `ex6_2_9` that outpaced BARON (35.6 s) and EAGO (1025.2 s), and a 196.6 s time on `ex6_2_13` that no CPU-based solver could solve within the 1-hour time limit.

`0` and `optcr = 0` to force BARON to terminate with a message of global optimality. In three of these problems, BARON terminated after one hour without managing to achieve the original tolerances of 1×10^{-3}, and consequently these problems were marked as unsolved. For six problems, BARON obtained the correct solution but did not certify the solution as globally optimal due to unbounded variables, so these problems were marked as correct. Two remaining problems were incorrectly classified as dual infeasible instead of optimal, and the remaining problems obtained certificates of optimality in times comparable to the original time of obtaining a local solution, so the global time was used as the solve time.

Dolan and Moré performance profiles [39] are shown in Figure 6.3 for all problems (top) and the non-triginometric problems (bottom). Of primary importance in this figure is the relationship between ARION and EAGO. Since ARION is built as an extension of EAGO and utilizes the same preprocessing and upper-bounding routines, the differences between these approaches is more clearly attributable to the GPU acceleration and batched node processing approach described in this work. For both plots in the set, the three versions of ARION are clear improvements over the serial EAGO solver. Around $\tau = 1$, EAGO appears to outperform some versions of ARION, but this is mainly the result of self-competition between the different ARION versions: if any two versions of ARION are removed, the remaining profile sits entirely above the profile for EAGO, for both the top and bottom plots. Considering only the MultiSobol version of ARION, a greater proportion of problems were solved by ARION than EAGO (84 versus 77), and of the 77 problems solved by both solvers, ARION reported faster times in 50 cases. Additionally, of the problems where EAGO was faster, solution times reported by ARION were generally below 1.0 s, indicating that relatively easier problems may favor serial calculation. Notably, ARION was slower than EAGO on three problems with solution times over 2.0 s (`hs62`, `st_e11`, `st_e16`). A closer examination of these problems reveals that a majority of the solve time is spent on the upper-bounding problem, which currently is called stochastically within EAGO. These slower times may therefore be the result of the upper-bounding solver (by default, IPOPT [148]) being called a greater number of times in these particular runs. With the exceptions of `hs62` and `st_e11`, for all problems solved by EAGO in more than 10.0 s (16 of 105), ARION achieved a faster time. ARION also solved an additional seven problems that could not be solved by EAGO.

Comparing ARION against the alternative solvers of SCIP and BARON in Figure 6.3, ARION is competitive but generally weaker than the more mature solvers. A large part of this difference may be due to more sophisticated preprocessing techniques that exist within SCIP and BARON. We hypothesize that without these techniques, EAGO and ARION are unable to solve, or take

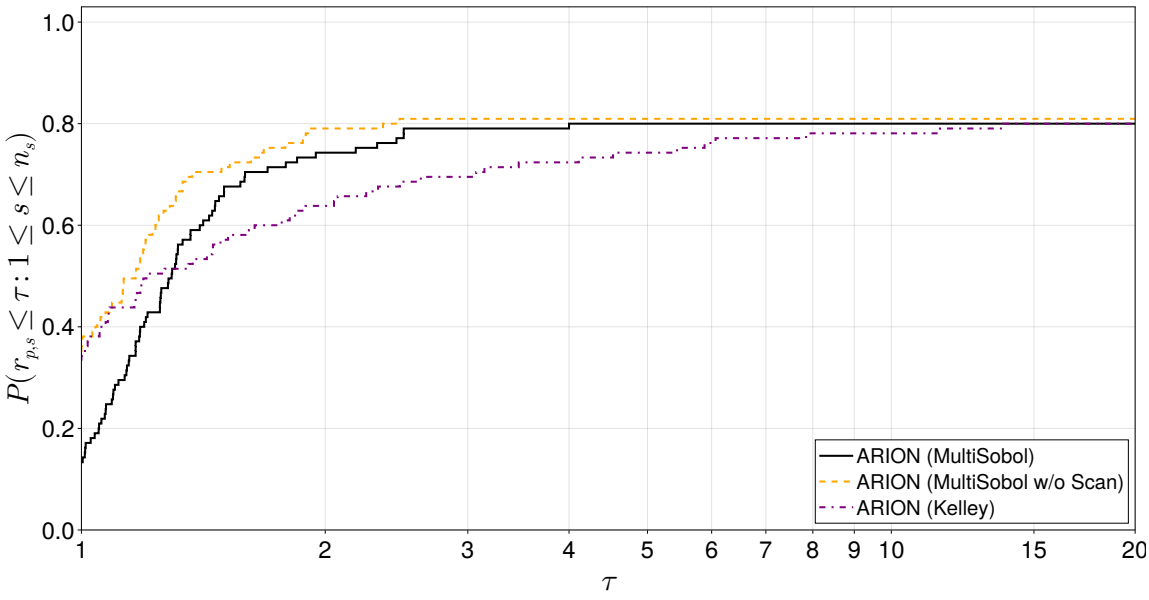

Figure 6.4: Dolan and Moré performance profiles [39] comparing the three forms of ARION—one using the method presented in Algorithm 6 (MultiSobol), one using Algorithm 6 without the scanning procedure of Algorithm 7 (MultiSobol w/o Scan), and one using a GPU-adapted version of Kelley's method (Kelley). Profiles are shown for all benchmark problems. Between the three options for ARION, the version without the scanning procedure of Algorithm 7 had the strongest performance, obtaining all solutions within 2.5× of the fastest ARION time. With the scan, solutions were obtained within 4× of the fastest time. The version of ARION using Kelley's method obtained more fastest times than ARION using the MultiSobol algorithm, but was more inconsistent in its solution times, obtaining some results at most 14× slower than the faster of the other two methods.

significantly longer on, some problems that are quickly solved by the other solvers. The increased processing speed made possible through GPU acceleration clearly represents an improvement over EAGO even without these techniques, but this speed alone is not enough to generally outclass existing mature solvers on this set of problems.

It is also valuable to consider differences between the performances of the three presented versions of ARION. At a high level, the performance across the benchmark set was remarkably consistent across the methods. For a closer inspection, Figure 6.4 shows performance profiles for only the thee versions of ARION. From this plot, it is more clear that the version of ARION using Algorithm 6 without the scan in Algorithm 7 is marginally superior to the other two methods, but including the scan is not, in general, a large detriment to the solve times. E.g., the version without the scan never took more than 2.5× the other ARION methods, and the version with the scan never took more than 4× the other methods. With Kelley's method, although this version was competitive at achieving the fastest times compared to other ARION methods, it struggled more with consistency, finding solutions at worst 14× slower than the other versions of ARION. Part of the differences

between these methods may be due to different B&B trees that are created due to differences in the lower bounds obtained for each node.

A more major factor explaining differences between the ARION methods is the time spent calculating relaxations and solving LPs. For ARION using Kelley's method, each node requires at most eight calls to the GPU-accelerated relaxation evaluator produced by `SourceCodeMcCormick.jl`, and at most eight LP solutions from either `BatchPDLP.jl` or the CPU-based Gurobi solver. The relaxations are quick to calculate in general, but eight separate calls to the relaxation kernel may add a small amount of overhead time to each lower-bounding procedure. The time required for obtaining the LP solutions is likely a larger contributor to the total lower-bounding time with Kelley's method. For the version of ARION using Algorithm 6, significantly more relaxations are evaluated (800 per node, versus at most eight from Kelley's method), but because the relaxations are fast to evaluate, this method is more consistently fast due to only solving one LP per node. Across the problems in the benchmark set, the scan procedure appears to be a slight detriment as compared to the version without the scan. This can be attributed to the doubled number of relaxations evaluated (800 per node, versus 400 per node without the scan), as well as the small amount of logic needed to perform the scan and rescale the linearization points. The marginally tighter lower bounds that could be obtained did not make up for the increased calculation effort for a majority of problems in the test set. This may be due, in part, to the complexity and problem-specific characteristics of the objective functions present in the test problems.

To better explore the relationship between problem complexity and the relative performance of ARION, we further compared SCIP, EAGO, and ARION on the `Alpine02` optimization problem:

$$f^* = \max_{\mathbf{x}} \prod_{i=1}^{n} \sin(x_i)\sqrt{x_i} \tag{6.6}$$
$$\text{s.t.} \quad \mathbf{0} \leq x_i \leq \mathbf{10} \quad \forall i = 1, \cdots, n.$$

The difficulty of (6.6) can be adjusted by changing the problem dimensionality n, which allows for a scalable difficulty without significant alterations to the problem structure. The dimensionality was increased for each solver until the problem could no longer be solved within the testing time of two hours, with the combined results displayed in Figure 6.5. Note that for EAGO and all versions of ARION, the setting `branch_cvx_factor = 1.0` was used, as it gave stronger results during testing. For problem dimensionalities below 7, all three solvers obtained times at or below 10.0 s, with ARION performing marginally better than EAGO. Starting at a dimensionality of 7, however, all

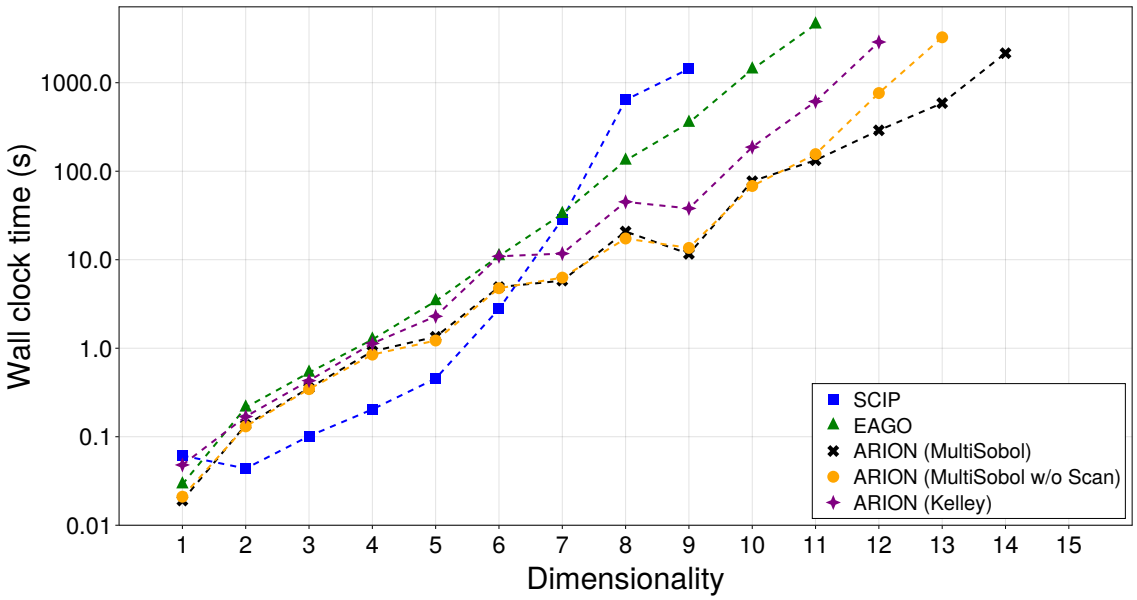

Figure 6.5: The wall clock time required by SCIP, EAGO, and three versions of ARION to solve the Alpine02 problem with different dimensionalities n. ARION is tested using the method described by Algorithm 6 (MultiSobol), the same method without the scan procedure of Algorithm 7 (MultiSobol w/o Scan), and a GPU-adapted version of Kelley's method (Kelley). The CPU-based SCIP and EAGO solvers could solve at most the 9- and 11-dimensional versions of this problem within two hours, whereas all versions of ARION were able to solve the 12-dimensional problem instance. ARION with Algorithm 6 was either equal or the fastest out of all methods for instances above 6 dimensions, and was the only version capable of solving the 14-dimensional instance.

three versions of ARION are able to overtake the CPU-based solvers and consistently report faster results.

Similar to the analysis between ARION methods presented for Figure 6.4, the version of ARION with Kelley's method appears to be the weakest among the ARION methods, solving at most the 12-dimensional version of the problem within two hours. This version of ARION is the closest GPU-based analog to EAGO, since effectively the same lower-bounding approach is used. Below solve times of roughly 10.0 s, this version of ARION was roughly equivalent with EAGO. Beyond 10.0 s, ARION quickly became faster, eventually maintaining a roughly one order-of-magnitude speedup relative to EAGO as the problem complexity increased. This ultimately resulted in an improvement of one dimension in the size of the `Alpine02` problem that could be addressed within the time limit.

Across all dimensionalities, the versions of ARION using Algorithm 6 obtained faster times than the version with Kelley's method. Much like in the more general test set, the versions with and without the scan procedure were fairly consistent with one another up to a dimensionality of 11. Above this size, however, the version with the scan becomes more dominant. The improved performance

of the scan in this problem may be due to the high dimensionality and relative complexity of the objective function. Particularly, as the dimensionality increases, the points selected by the Sobol sequence become increasingly sparse within each node, which decreases the likelihood of identifying valuable linearization points. For cases such as this with higher dimensions, scanning and rescaling with Algorithm 7 provides greater opportunity to find good linearization points within each node, with the result being tighter lower bounds and a smaller B&B tree. ARION with the scanning procedure is the only version or solver that was able to solve the 14-dimensional `Alpine02` problem within two hours, which was a three order-of-magnitude improvement over the CPU-based EAGO solver.

6.5 Conclusion

In this work, we demonstrate how individual GPU-accelerated components may be combined together to enable GPU-accelerated B&B for solving NLPs to global optimality. Using the EAGO solver as a platform, we reconstruct the main B&B algorithm and lower-bounding routine to allow simultaneous lower-bounding of multiple nodes in the B&B tree. Within the batched lower-bounding process, relaxations of the original NLP for each node are calculated in concert using problem-specific kernels generated by `SourceCodeMcCormick.jl`. Information about the relaxations and relaxation subgradients are then composed into relaxed LPs, which are subsequently solved using a combination of a GPU-accelerated batched LP solver (`BatchPDLP.jl`) and a CPU-based subsolver (Gurobi, in this work). Lower bounds and the solutions to the relaxed problems are then distributed to the appropriate B&B nodes which can then be branched on to continue the spatial B&B algorithm. The combination of these elements enables a majority of the computational effort in solving NLPs to global optimality to occur on specialized, parallel hardware.

The implementation of this approach, which we name ARION, results in a stark improvement over the comparable CPU-based EAGO solver. With a set of different lower-bounding methods, ARION cleanly outperforms EAGO across a wide selection of NLPs from MINLPLib and on a scalable benchmark test problem. The speedup obtained from using a GPU enables more benchmark problems to be solved, results in faster times on a majority of benchmark problems, and allows higher-dimensional problems to be addressed, as compared to the non-augmented EAGO solver. As compared to the more mature solvers of BARON and SCIP, ARION lags behind on the benchmark test set problems most likely due to advanced preprocessing and bounds tightening techniques that are not originally available in EAGO. Nonetheless, the node processing speedup of ARION relative

to EAGO substantially reduces the gap to the other solvers. On the scalable benchmark problem, ARION significantly outperformed SCIP and EAGO in terms of solution speed on more challenging instances and on the scale of instances that were solvable within the time limit.

The developments described herein are a strong motivation for the use of GPUs within DGO solvers. While improved preprocessing and bounds tightening techniques are, and will continue to be, important for addressing a variety of problems, GPU acceleration is a viable path toward decreasing the solution time of problems more generally. We also demonstrate within this work that different methodologies that may be impractical for CPU use may become practical on parallelized hardware. Given the clear advantage that GPU acceleration offers, we expect mature and commercial solvers to begin leveraging this widely available and useful hardware. Our future efforts will be similarly focused on developing new ideas to improve upon and effectively use GPU-based tools for global optimization.

Chapter 7

Reflection and Future Opportunities

7.1 Summary and Reflection

The work described in this thesis centered around the goal of applying GPU parallelization to the spatial B&B method to solve nonconvex NLPs. Primarily, these efforts were inspired by an interest in applying optimization tools to traditional chemical engineering applications, where systems are frequently nonlinear in nature. For many such applications, including large-scale process design, model validation, chemical and phase equilibria, and safety and robust control, global optimization and global minima are uniquely valuable. However, it is often challenging to apply deterministic global optimization tools to chemical engineering problems of practical interest due to their relatively high dimensionality and the high computational costs associated with the spatial B&B method. This work sought to expand the range of problems that can be practically solved by leveraging powerful, modern computing architectures—graphics processing units (GPUs)—that are, at the time of writing, currently underutilized for deterministic global optimization. As part of this work, and to enable the solution of optimization problems in engineering more generally, this thesis does not address specific instances of problems, but rather utilizes mathematics and computer science concepts to create a general solver that is capable of addressing many potential problems.

The approach used to create a GPU-accelerated global solver was to first create individual, GPU-accelerated solver components. As laid out in Chapter 2, this work focused on parts of the expensive lower-bounding routine—namely, relaxation calculations and relaxed LP solving—which each have

inherent challenges for parallelization that perhaps in part explain why GPU parallelization has so far not been incorporated into leading commercial global solvers. As a reiteration of the hypothesis originally expressed in Chapter 1, this work is rooted in the idea that despite these challenges, these components can still be made effective on parallelized hardware. The hypothesis continues that if these components can be made faster, their integration together should result in a fast, parallelized global solver.

The first portion of this thesis addressed the development of the aforementioned components. Specifically, in Chapters 3 and 4, a source code generation approach was applied to allow McCormick relaxations to be evaluated using GPU resources. The resulting open-source product, available as `SourceCodeMcCormick.jl`, was shown to be considerably faster than a serial CPU implementation of the McCormick relaxation rules. Although some of the initial concerns about parallelizing the relaxations, such as the potential for warp divergence, were shown to be well founded, the impact on performance was not critical enough to render the approach ineffective. This development is valuable to the optimization and process systems engineering disciplines due to the new accessibility it offers to use hardware accelerators for global optimization tasks. The second major component, covered by Chapter 5, was the parallelization of batches of small-scale LPs. This approach too was shown to be highly effective, such that the open-source product, `BatchPDLP.jl`, was competitive with existing commercial and open-source LP solvers. The batched approach did, however, experience high variance in solution times across LPs within each batch, but this particular issue was found to be manageable. This development is of direct importance to global optimization, as it addresses the challenge of quickly solving large numbers of LPs in a way that is competitive with existing serial LP solvers. This work may also be valuable in cases where multiple parameter sets are being explored for the same problem structure, which may occur in, for example, process design applications. These three chapters, and the research products they produced, confirmed the first part of the hypothesis that the individual components could be separately effective.

The second part of this thesis discussed the integration of these components. Applying the components together to solve NLPs was briefly touched on in Chapter 3, but it was the main focus of Chapter 6. This integration was necessary to determine whether this approach to GPU parallelization would be effective in practice. Across the benchmark test set of typical NLPs and a scalable-difficulty problem, the hybrid CPU-GPU approach (ARION) was shown to be effective and represented a clear improvement over the serial CPU-based solver it used as a foundation (EAGO). Within the benchmarking, there remained a gap in performance relative to commercial solvers that we expect is the result of B&B techniques not yet implemented in EAGO. Nonetheless, the strong

performance of ARION relative to EAGO suggests that these commercial solvers would also see some form of benefit from GPU use. This is also suggested by the particularly strong performance on the scalable test problem relative to the non-GPU solvers. With this demonstrated level of performance on a recognized benchmark test set, Chapter 6 and ARION effectively confirm the second part of the hypothesis that the integrated GPU components would lead to an improvement in solution speed on problems typically addressed by other, CPU-based global solvers. An especially strong use case for this approach may be in optimal control applications, where the impact of the initial compilation time is lower due to the need for repeated solving of similarly structured problems by the controller.

7.2 Future Opportunities

7.2.1 Relaxations

The source code transformation approach was shown to be effective and provided some unique benefits over a potential operator overloading-like approach, but both methods may still be made more effective. An operator overloading or multiple dispatch approach, for example, experiences higher overhead than the source code generation approach due to the larger number of kernels being called. However, this overhead cost can be greatly reduced in successive iterations if CUDA Graphs are utilized, which was not explored in this work. This approach would likely be simpler to implement and maintain than the methods utilized by `SourceCodeMcCormick.jl`, though some optimizations, such as expression substitutions, would need to be handled externally as, without adding significant complexity, the functions called through operator overloading would lack the context of the larger expression.

With regard to the source code generation approach itself, one of the key weaknesses is the long compilation time. Strategies exist to avoid this compilation time in practice, with sufficient notice about the problem being solved, but for a general-purpose solver this behavior is not desirable. Improvements to the heuristics of the length of generated kernels may help, but the potential impact of this modification is limited. A likely path to more substantial improvements is replacing portions of generated kernels with precompiled device functions. This change would allow for shorter, less complicated kernels to compile at runtime. A significant challenge with this approach is that device functions do not allow dynamic dispatch, and thus the conditional logic within many McCormick relaxations will need to be handled carefully.

Another notable challenge lies in the calculation of natural interval extensions within the gen-

erated kernels. With CPU-based relaxation calculations, such as through `McCormick.jl`, packages such as `IntervalArithmetic.jl` may be used to ensure compliance with IEEE-1788 (the official standard for interval arithmetic). Within `SourceCodeMcCormick.jl`, the interval calculations are not currently compliant with this standard because correct floating-point rounding is not enforced within the kernel. To the best of my knowledge, this is not currently possible with the `CUDA.jl` tool that enables CUDA kernels to be written in Julia. An alternative may be to switch to C/C++ for the generated kernels, as this would allow for more direct control over the use of CUDA, such as specific GPU rounding modes for individual calculations. Making this change would allow for greater numerical stability of the relaxations as calculated on the GPU and would ensure the validity of the calculated intervals.

7.2.2 LP Solving

The choice to use the PDLP algorithm for GPU parallelization was made in part because GPU-accelerated versions of PDLP existed for large-scale LPs, and the adaptation to small-scale LPs was relatively straightforward. Nonetheless, simplex and interior-point methods may also be valuable options to explore given their long and robust history of development. In particular, the dual simplex algorithm would be valuable to implement in batched form for either NLP or MINLP solving. With the dual simplex algorithm, once an LP is solved, the addition of a constraint does not require finding a new feasible point to restart the algorithm. This effective hot-start property makes dual simplex a good choice for implementing, e.g., an improved version of Kelley's algorithm on the GPU, as solutions to LPs beyond the first should be comparatively fast. This development would also be helpful for solving MIPs if, for example, whole batches of LPs could be maintained during branching to enable batched hot-starting.

For solving MINLPs, although the relaxation calculation approach would not be substantially affected, any batched LP solver would need to implement crossover. Crossover shifts continuous solutions to nearby integer values without affecting the objective function, which is critical for verifying whether integer-valued variables are integer-valued in the LP solution. This work would likely be fruitful, as crossover methods are well studied and MIPs have a wide variety of industrial applications. There are several examples of GPU-accelerated MIP solvers, but to the best of my knowledge, no GPU-accelerated MINLP solvers exist. The improvement obtained within the current work for NLP solving suggests that the use of GPUs for MINLPs would result in similarly improved computational performance.

7.3 Final Remarks

The work in this thesis was intended to address a challenge in chemical engineering, though in order to address this challenge, it was necessary to pull from, and make contributions that are relevant to, a variety of other disciplines as well. Most notably, the development of this GPU-accelerated global solver took elements from applied mathematics, computer science, and the process systems engineering specialization, and was presented at conferences to researchers from those fields as well as to chemical engineers. This work may be directly relevant to researchers in these fields as parallelized optimization algorithms continue to be explored. Additionally, research products such as `BatchPDLP.jl` may find unexpected value in applications that require the solution of many, similar LPs. In sum, by demonstrating an efficient method of GPU parallelization for the global solution of NLPs, this research represents a significant step toward the use of more powerful computing hardware for solving complex engineering challenges more generally.

Bibliography

[1] *MINLPLib: A library of mixed-integer and continuous nonlinear programming instances*, https://minlplib.org/ (2025). Accessed 2025-12-16.

[2] *Top 500 list* (2025), URL `https://www.top500.org/lists/top500/2025/11/`.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous distributed systems*, https://arxiv.org/abs/1603.04467 (2016).

[4] K.M. Abughalieh and S.G. Alawneh, *A survey of parallel implementations for model predictive control*, IEEE Access 7 (2019), pp. 34348–34360.

[5] C. Adjiman, S. Dallwig, C. Floudas, and A. Neumaier, *A global optimization method, αBB, for general twice-differentiable constrained NLPs — I. theoretical advances*, Computers & Chemical Engineering 22 (1998), pp. 1137–1158.

[6] C.S. Adjiman and C.A. Floudas, *Rigorous convex underestimators for general twice-differentiable problems*, Journal of Global Optimization 9 (1996), pp. 23–40.

[7] M.E. Álvarez, M. Bourouis, and X. Esteve, *Vapor-liquid equilibrium of aqueous alkaline nitrate and nitrite solutions for absorption refrigeration cycles with high-temperature driving heat*, Journal of Chemical & Engineering Data 56 (2011), pp. 491–496.

[8] B. Amos and J.Z. Kolter, *OptNet: Differentiable optimization as a layer in neural networks*, in *International conference on machine learning*. PMLR, 2017, pp. 136–145.

[9] I.P. Androulakis, C.D. Maranas, and C.A. Floudas, *αBB: A global optimization method for general constrained nonconvex problems*, Journal of Global Optimization 7 (1995), pp. 337–363.

[10] D. Applegate, M. Díaz, O. Hinder, H. Lu, M. Lubin, B. O'Donoghue, and W. Schudy, *Practical large-scale linear programming using primal-dual hybrid gradient*, arXiv (2021). https://arxiv.org/abs/2106.04756.

[11] D. Applegate, O. Hinder, H. Lu, and M. Lubin, *Faster first-order primal-dual methods for linear programming using restarts and sharpness*, Mathematical Programming 201 (2022), pp. 133–184.

[12] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, and J.M. Siskind, *Automatic differentiation in machine learning: a survey*, The Journal of Machine Learning Research 18 (2018), pp. 1–43.

[13] M. Beckers, V. Mosenkis, and U. Naumann, *Adjoint Mode Computation of Subgradients for McCormick Relaxations*, Springer Berlin Heidelberg (2012), pp. 103–113.

[14] D.E. Bernal, C.D. Laird, S.M. Harwood, D. Trenev, and D. Venturelli, *Impact of emerging computing architectures and opportunities for process systems engineering applications*, in *FOCAPO-CPC 2023*, Jan., San Antonio, TX. FOCAPO-CPC, 2023.

[15] D.E. Bernal Neira, C.D. Laird, L.R. Lueg, S.M. Harwood, D. Trenev, and D. Venturelli, *Utilizing modern computer architectures to solve mathematical optimization problems: A survey*, Computers & Chemical Engineering 184 (2024), p. 108627.

[16] T. Besard, C. Foket, and B. De Sutter, *Effective extensible programming: Unleashing Julia on GPUs*, IEEE Transactions on Parallel and Distributed Systems (2018), pp. 827–841.

[17] J. Bieling, P. Peschlow, and P. Martini, *An efficient GPU implementation of the revised simplex method*, in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*, Apr., Atlanta, GA, USA. IEEE, 2010, pp. 1–8.

[18] C. Bischof and H. Bucker, *Computing derivatives of computer programs*, NIC Series 3 (2000), pp. 287–299.

[19] S. Bolusani, M. Besançon, K. Bestuzheva, A. Chmiela, J. Dionísio, T. Donkiewicz, J. van Doornmalen, L. Eifler, M. Ghannam, A. Gleixner, C. Graczyk, K. Halbig, I. Hedtke, A. Hoen, C. Hojny, R. van der Hulst, D. Kamp, T. Koch, K. Kofler, J. Lentz, J. Manns, G. Mexi, E. Mühmer, M.E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano,

M. Turner, S. Vigerske, D. Weninger, and L. Xu, *The SCIP Optimization Suite 9.0*, Technical report, Optimization Online (2024). https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/.

[20] A. Bompadre and A. Mitsos, *Convergence rate of McCormick relaxations*, Journal of Global Optimization 52 (2011), pp. 1–28.

[21] D. Bongartz and A. Mitsos, *Deterministic global optimization of process flowsheets in a reduced space using McCormick relaxations*, Journal of Global Optimization 69 (2017), pp. 761–796.

[22] D. Bongartz, J. Najman, S. Sass, and A. Mitsos, *MAiNGO - McCormick-based Algorithm for mixed-integer Nonlinear Global Optimization*, Tech. rep., RWTH-Aachen University (2018). http://permalink.avt.rwth-aachen.de/?id=729717.

[23] A. Boukedjar, M.E. Lalami, and D. El-Baz, *Parallel branch and bound on a cpu-gpu system*, in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Feb. IEEE, 2012, pp. 392–398.

[24] M.R. Bussieck, A.S. Drud, and A. Meeraus, *MINLPLib—a collection of test models for mixed-integer nonlinear programming*, INFORMS Journal on Computing 15 (2003), pp. 114–119.

[25] D.T.S. Cara Touretzky, Robert Luce, *Using GPUs to solve LPs: What's in it for me?*, Online (2025), URL https://www.gurobi.com/resources/blog/using-gpus-to-solve-lps-what-s-in-it-for-me/. Accessed: Apr. 28, 2025.

[26] L. Casado, J. Martínez, I. García, and E. Hendrix, *Branch-and-bound interval global optimization on shared memory multiprocessors*, Optimization Methods and Software 23 (2008), pp. 689–701.

[27] M. Ceberio and F. Modave, *Interval-Based Multicriteria Decision Making*, Elsevier (2006), pp. 281–294.

[28] B. Chachuat, B. Houska, R. Paulen, N. Peri'c, J. Rajyaguru, and M.E. Villanueva, *Set-theoretic approaches in analysis, estimation and control of nonlinear systems*, IFAC-PapersOnLine 48 (2015), pp. 981–995.

[29] I. Chakroun and N. Melab, *Operator-level GPU-accelerated branch and bound algorithms*, Procedia Computer Science 18 (2013), pp. 280–289.

[30] I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi, *Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm*, Concurrency and Computation: Practice and Experience 25 (2012), pp. 1121–1136.

[31] A. Chambolle and T. Pock, *A first-order primal-dual algorithm for convex problems with applications to imaging*, Journal of Mathematical Imaging and Vision 40 (2011), pp. 120–145.

[32] J. Charlton, S. Maddock, and P. Richmond, *Two-dimensional batch linear programming on the gpu*, Journal of Parallel and Distributed Computing 126 (2019), pp. 152–160.

[33] J. Chen and J. Revels, *Robust benchmarking in noisy environments*, https://arxiv.org/abs/1608.04295 (2016).

[34] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cuDNN: Efficient primitives for deep learning*, arXiv (2014). https://arxiv.org/abs/1410.0759.

[35] C/MSC - Microprocessor Standards Committee, *1788-2015 - IEEE standard for interval arithmetic*, IEEE Std 1788-2015 (2015), pp. 1–79.

[36] C/MSC - Microprocessor Standards Committee, *754-2019 - IEEE standard for floating-point arithmetic*, IEEE Std 754-2019 (Revision of IEEE 754-2009) (2019), pp. 1–84.

[37] C.J. Corbett, M. Maier, M. Beckers, U. Naumann, A. Ghobeity, and A. Mitsos, *Compiler-generated subgradient code for McCormick relaxations*, Tech. rep., RWTH Aachen, Department of Computer Science (2011).

[38] A.C. Crespo, J.M. Dominguez, A. Barreiro, M. Gómez-Gesteira, and B.D. Rogers, *Gpus, a new tool of acceleration in cfd: Efficiency and reliability on smoothed particle hydrodynamics methods*, PLoS ONE 6 (2011), p. e20685.

[39] E.D. Dolan and J.J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical Programming 91 (2002), pp. 201–213.

[40] K. Du and R.B. Kearfott, *The cluster problem in multivariate global optimization*, Journal of Global Optimization 5 (1994), pp. 253–265.

[41] I. Dunning, J. Huchette, and M. Lubin, *JuMP: A modeling language for mathematical optimization*, SIAM Review 59 (2017), pp. 295–320.

[42] M.B. Eriksen and S. Rasmussen, *GPU accelerated parameter estimation by global optimization using interval analysis*, Master's thesis, Aalborg University (2013).

[43] Fair Isaac Corporation (FICO), *FICO Xpress Optimizer Reference Manual* (2024), URL `https://www.fico.com/fico-xpress-optimization/docs/latest/overview.html`.

[44] M.J. Flynn, *Some computer organizations and their effectiveness*, IEEE Transactions on Computers C-21 (1972), pp. 948–960.

[45] N.F. Gade-Nielsen, J.B. Jørgensen, and B. Dammann, *MPC toolbox with GPU accelerated optimization algorithms*, in *10th European workshop on advanced control and diagnosis*. Technical University of Denmark, 2012.

[46] GAMS Development Corporation, *General Algebraic Modeling System (GAMS) release 52.4.0* (2026).

[47] M. Garland, S.L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, *Parallel computing experiences with CUDA*, IEEE Micro 28 (2008), pp. 13–27.

[48] D. Geer, *Chip makers turn to multicore processors*, Computer 38 (2005), pp. 11–13.

[49] B. Gendron and T.G. Crainic, *Parallel branch-and-branch algorithms: Survey and synthesis*, Operations Research 42 (1994), pp. 1042–1066.

[50] J. Glaser, T.D. Nguyen, J.A. Anderson, P. Lui, F. Spiga, J.A. Millan, D.C. Morse, and S.C. Glotzer, *Strong scaling of general-purpose molecular dynamics simulations on gpus*, Computer Physics Communications 192 (2015), pp. 97–107.

[51] R.X. Gottlieb, D. Alston, and M.D. Stuber, *Re-architected PDLP for batch parallelization of linear programs*, Under Revision (2026).

[52] R.X. Gottlieb and M.D. Stuber, *Global dynamic optimization using hardware-accelerated programming*, in *AIChE Annual Meeting 2022*, Nov., Phoenix, AZ. AIChE, 2022.

[53] R.X. Gottlieb and M.D. Stuber, *PSORLab/SourceCodeMcCormick.jl v0.5.2*, `https://github.com/PSORLab/SourceCodeMcCormick.jl` (2023). Accessed 2026-04-03.

[54] R.X. Gottlieb and M.D. Stuber, *GPU-accelerated deterministic global optimization*, in *ISMP 2024*, Jul., Palais des congrès de Montréal, Montréal, Canada. ISMP, 2024.

[55] R.X. Gottlieb and M.D. Stuber, *Automatic generation of GPU kernels for evaluators of McCormick-based relaxations and subgradients*, Under Revision (2025).

[56] R.X. Gottlieb and M.D. Stuber, *PSORLab/BatchPDLP.jl*, https://github.com/PSORLab/BatchPDLP.jl (2025).

[57] R.X. Gottlieb, P. Xu, and M.D. Stuber, *Automatic source code generation of complicated models for deterministic global optimization with parallel architectures*, in *FOCAPO-CPC 2023*, Jan., San Antonio, TX. FOCAPO-CPC, 2023.

[58] R.X. Gottlieb, P. Xu, and M.D. Stuber, *Automatic source code generation for deterministic global optimization with parallel architectures*, Optimization Methods and Software 41 (2026), pp. 308–346.

[59] S. Gowda, Y. Ma, A. Cheli, M. Gwóźdź, V.B. Shah, A. Edelman, and C. Rackauckas, *High-performance symbolic-numerics via multiple dispatch*, ACM Communications in Computer Algebra 55 (2021), pp. 92–96.

[60] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, no. 19 in Frontiers in Appl. Math., SIAM, Philadelphia, PA, 2000.

[61] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual* (2024), URL `https://www.gurobi.com`.

[62] A. Gurung and R. Ray, *Solving batched linear programs on gpu and multicore cpu*, arXiv (2016). https://arxiv.org/abs/1609.08114.

[63] T.D. Han and T.S. Abdelrahman, *Reducing branch divergence in GPU programs*, in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, Mar. ACM, no. 3 in GPGPU-4, 2011, pp. 1–8.

[64] M. Hladík, *On the efficient Gerschgorin inclusion usage in the global optimization αBB method*, Journal of Global Optimization 61 (2014), pp. 235–253.

[65] R. Horst and H. Tuy, *Global Optimization: Deterministic Approaches*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013 Nov., URL `https://books.google.com/books?id=Pe_1CAAAQBAJ`.

[66] S. Huang, S. Xiao, and W. Feng, *On the energy efficiency of graphics processing units for scientific computing*, in *2009 IEEE International Symposium on Parallel & Distributed Processing*, May. IEEE, 2009, pp. 1–8.

[67] Q. Huangfu and J.A.J. Hall, *Parallelizing the dual revised simplex method*, Mathematical Programming Computation 10 (2017), pp. 119–142.

[68] M. Imai, Y. Yoshida, and T. Fukumura, *A parallel searching scheme for multiprocessor systems and its application to combinatorial problems*, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., IJCAI'79, 1979, p. 416–418.

[69] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis*, Springer-Verlag, London, 2001.

[70] J.H. Jung and D.P. O'Leary, *Implementing an interior point method for linear programs on a cpu-gpu system*, Electronic Transactions on Numerical Analysis 28 (2008), p. 37.

[71] N. Karmarkar, *A new polynomial-time algorithm for linear programming*, in *Proceedings of the sixteenth annual ACM symposium on Theory of computing - STOC '84*. ACM Press, STOC '84, 1984, pp. 302–311.

[72] N. Kazazakis, *Parallel computing, interval derivative methods, heuristic algorithms, and their implementation in a numerical solver, for deterministic global optimization*, Ph.D. diss., Imperial College London (2017).

[73] N. Kazazakis and C.S. Adjiman, *Arbitrarily tight αBB underestimators of general non-linear functions over sub-optimal domains*, Journal of Global Optimization 71 (2018), pp. 815–844.

[74] J.E. Kelley Jr., *The cutting-plane method for solving convex programs*, Journal of the Society for Industrial and Applied Mathematics 8 (1960), pp. 703–712.

[75] G.K. Kenway, C.A. Mader, P. He, and J.R. Martins, *Effective adjoint approaches for computational fluid dynamics*, Progress in Aerospace Sciences 110 (2019), p. 100542.

[76] A. Khajavirad and N.V. Sahinidis, *A hybrid LP/NLP paradigm for global optimization relaxations*, Mathematical Programming Computation 10 (2018), pp. 383–421.

[77] K.A. Khan, H.A.J. Watson, and P.I. Barton, *Differentiable McCormick relaxations*, Journal of Global Optimization 67 (2016), pp. 687–729.

[78] K.A. Khan, M. Wilhelm, M.D. Stuber, H. Cao, H.A.J. Watson, and P.I. Barton, *Corrections to: Differentiable McCormick relaxations*, Journal of Global Optimization 70 (2018), pp. 705–706.

[79] S. Kiel, E. Auer, and A. Rauh, *Uses of GPU powered interval optimization for parameter identification in the context of SO fuel cells*, IFAC Proceedings Volumes 46 (2013), pp. 558–563.

[80] T. Koch, T. Berthold, J. Pedersen, and C. Vanaret, *Progress in mathematical programming solvers from 2001 to 2020*, EURO Journal on Computational Optimization 10 (2022), p. 100031.

[81] A. Krizhevsky, *Cuda-convnet* (2014), URL `code.google.com/p/cuda-convnet/`.

[82] M.E. Lalami, D. El-Baz, and V. Boyer, *Multi GPU implementation of the simplex algorithm*, in *2011 IEEE International Conference on High Performance Computing and Communications*, sep. IEEE, 2011.

[83] B. Legat, O. Dowson, J.D. Garcia, and M. Lubin, *MathOptInterface: A data structure for mathematical optimization problems*, INFORMS Journal on Computing 34 (2022), pp. 672–689.

[84] C.E. Leiserson, N.C. Thompson, J.S. Emer, B.C. Kuszmaul, B.W. Lampson, D. Sanchez, and T.B. Schardl, *There's plenty of room at the top: What will drive computer performance after Moore's law?*, Science 368 (2020).

[85] D. Limon, T. Alamo, E. Camacho, and J. Bravo, *Robust MPC of constrained nonlinear systems based on interval arithmetic*, IEE Proceedings - Control Theory and Applications 152 (2005), pp. 325–332.

[86] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, *NVIDIA Tesla: A unified graphics and computing architecture*, IEEE Micro 28 (2008), pp. 39–55.

[87] H. Lu, *First-order methods for linear programming* (2024). https://arxiv.org/abs/2403.14535.

[88] H. Lu, Z. Peng, and J. Yang, *cuPDLPx: A further enhanced GPU-based first-order solver for linear programming*, arXiv (2025). https://arxiv.org/abs/2507.14051.

[89] H. Lu and J. Yang, *cuPDLP.jl: A GPU implementation of restarted primal-dual hybrid gradient for linear programming in Julia*, arXiv (2024). https://arxiv.org/abs/2311.12180.

[90] H. Lu and J. Yang, *On the geometry and refined rate of primal–dual hybrid gradient for linear programming*, Mathematical Programming 212 (2024), pp. 349–387.

[91] H. Lu, J. Yang, H. Hu, Q. Huangfu, J. Liu, T. Liu, Y. Ye, C. Zhang, and D. Ge, *cuPDLP-C: A strengthened implementation of cuPDLP for linear programming by C language*, arXiv (2023). https://arxiv.org/abs/2312.14832.

[92] D. Luebke, *CUDA: Scalable parallel programming for high-performance scientific computing*, in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, May. IEEE, 2008.

[93] D. Maclaurin, D. Duvenaud, and R. Adams, *Gradient-based hyperparameter optimization through reversible learning*, in F. Bach and D. Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning Proceedings of Machine Learning Research* vol. 37, *Proceedings of Machine Learning Research* vol. 37, 07–09 Jul, Lille, France. PMLR, 2015, pp. 2113–2122, URL `https://proceedings.mlr.press/v37/maclaurin15.html`.

[94] A. Makhorin, *GLPK (GNU linear programming kit)*, http://www.gnu.org/s/glpk/glpk.html (2008).

[95] G.P. McCormick, *Computability of global solutions to factorable nonconvex programs: Part I — convex underestimating problems*, Mathematical Programming 10 (1976), pp. 147–175.

[96] X. Meyer, P. Albuquerque, and B. Chopard, *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 (BALCOR 2011), 22-25 September 2011, Thessaloniki, Greece*. 2011.

[97] M. Mezmaz, N. Melab, and E.G. Talbi, *A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems*, in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–9.

[98] R. Misener and C.A. Floudas, *ANTIGONE: Algorithms for coNTinuous / Integer Global Optimization of Nonlinear Equations*, Journal of Global Optimization 59 (2014), pp. 503–526.

[99] A. Mitsos, B. Chachuat, and P.I. Barton, *McCormick-based relaxations of algorithms*, SIAM Journal on Optimization 20 (2009), pp. 573–601.

[100] S. Mittal and J.S. Vetter, *A survey of methods for analyzing and improving GPU energy efficiency*, ACM Computing Surveys 47 (2014), pp. 1–23.

[101] Modal, *GPU glossary* (2026), URL `https://modal.com/gpu-glossary`.

[102] R.E. Moore, *Methods and Applications of Interval Analysis*, Studies in Applied and Numerical Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1979 Jan.

[103] W.S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S.H.K. Narayanan, M. Schanen, and J. Doerfert, *Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. ACM, SC '21, 2021.

[104] J. Najman, D. Bongartz, and A. Mitsos, *Linearization of McCormick relaxations and hybridization with the auxiliary variable method*, Journal of Global Optimization 80 (2021), pp. 731–756.

[105] Y. Nesterov, *Subgradient methods for huge-scale optimization problems*, Mathematical Programming 146 (2013), pp. 275–297.

[106] A. Neumaier, *Complete search in continuous global optimization and constraint satisfaction*, Acta Numerica 13 (2004), pp. 271–369.

[107] NVIDIA, *CUDA Toolkit*, `https://developer.nvidia.com/cuda-toolkit` (2024). Accessed 2025-07-15.

[108] NVIDIA, *NVIDIA cuDSS (preview): A high-performance CUDA library for direct sparse solvers*, Online (2024), URL `https://docs.nvidia.com/cuda/cudss/`. Accessed: Apr. 28, 2025.

[109] NVIDIA, *NVIDIA Nsight Compute*, `https://developer.nvidia.com/nsight-compute` (2025). Version 2025.2.0, Accessed 2025-07-15.

[110] NVIDIA, *CUDA C++ Programming Guide v13.2* (2026), URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[111] NVIDIA, *NVIDIA Nsight Systems*, `https://developer.nvidia.com/nsight-systems` (2026). Version 2026.2.1, Accessed 2026-05-21.

[112] P.M. Pardalos and S.A. Vavasis, *Quadratic programming with one negative eigenvalue is np-hard*, Journal of Global Optimization 1 (1991), pp. 15–22.

[113] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, *Automatic differentiation in pytorch*, in *NIPS 2017 Workshop on Autodiff*. 2017, URL `https://openreview.net/forum?id=BJJsrmfCZ`.

[114] N. Ploskas and N. Samaras, *Efficient gpu-based implementations of simplex type algorithms*, Applied Mathematics and Computation 250 (2015), pp. 552–570.

[115] T. Pock and A. Chambolle, *Diagonal preconditioning for first order primal-dual algorithms in convex optimization*, in *2011 International Conference on Computer Vision*, Nov. IEEE, 2011, pp. 1762–1769.

[116] D. Reed, D. Gannon, and J. Dongarra, *HPC forecast: Cloudy and uncertain*, Communications of the ACM 66 (2023), pp. 82–90.

[117] D. Ruiz, *A scaling algorithm to equilibrate both rows and columns norms in matrices*, Tech. Rep. RAL-TR-2001-034, Rutherford Appleton Laboratory (2001). CM-P00040415.

[118] R.K. Saha, A. Pradhan, T. Ghosh, M. Jenamani, S.K. Singh, and A. Routray, *Gpu accelerated parallel implementation of linear programming algorithms*, in E. Pardede, P. Delir Haghighi, I. Khalil, and G. Kotsis (eds.), *Information Integration and Web Intelligence*, Cham. Springer Nature Switzerland, 2022, pp. 378–384.

[119] N.V. Sahinidis, *BARON: A general purpose global optimization software package*, Journal of Global Optimization 8 (1996), pp. 201–205.

[120] D.P. Sanders and L. Benet, *Intervalarithmetic.jl* (2014), URL `https://github.com/JuliaIntervals/IntervalArithmetic.jl`.

[121] J.K. Scott, M.D. Stuber, and P.I. Barton, *Generalized McCormick relaxations*, Journal of Global Optimization 51 (2011), pp. 569–606.

[122] R. Seidel, *Small-dimensional linear programming and convex hulls made easy*, Discrete & Computational Geometry 6 (1991), pp. 423–434.

[123] U.A. Shah, S. Yousaf, I. Ahmad, S.U. Rehman, and M.O. Ahmad, *Accelerating revised simplex method using gpu-based basis update*, IEEE Access 8 (2020), pp. 52121–52138.

[124] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch, *ParaSCIP: A parallel extension of SCIP*, in *Competence in High Performance Computing 2010*, Springer Berlin Heidelberg (2011), pp. 135–148.

[125] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler, *FiberSCIP—a shared memory parallelization of SCIP*, INFORMS Journal on Computing 30 (2018), pp. 11–30.

[126] A.B. Singer, *Global dynamic optimization*, Ph.D. diss., Massachusetts Institute of Technology (2004).

[127] A.B. Singer, J.W. Taylor, P.I. Barton, and W.H. Green, *Global dynamic optimization for parameter estimation in chemical kinetics*, The Journal of Physical Chemistry A 110 (2006), pp. 971–976.

[128] D. Singh and P.L. Dhepe, *Understanding the influence of alumina supported ruthenium catalysts synthesis and reaction parameters on the hydrodeoxygenation of lignin derived monomers*, Molecular Catalysis 480 (2020), p. 110525.

[129] E. Smith, J. Gondzio, and J. Hall, *GPU Acceleration of the Matrix-Free Interior Point Method*, Springer Berlin Heidelberg, Berlin, Heidelberg (2012), pp. 681–689.

[130] I. Sobol, *On the distribution of points in a cube and the approximate evaluation of integrals*, USSR Computational Mathematics and Mathematical Physics 7 (1967), pp. 86–112.

[131] Y. Song, H. Cao, C. Mehta, and K.A. Khan, *Bounding convex relaxations of process models from below by tractable black-box sampling*, Computers & Chemical Engineering 153 (2021), p. 107413.

[132] D.G. Spampinato and A.C. Elstery, *Linear optimization on modern GPUs*, in *2009 IEEE International Symposium on Parallel & Distributed Processing*, May, Rome, Italy. IEEE, 2009, pp. 1–8.

[133] M.D. Stuber, J.K. Scott, and P.I. Barton, *Convex and concave relaxations of implicit functions*, Optimization Methods and Software 30 (2015), pp. 424–460.

[134] M.D. Stuber, A. Wechsung, A. Sundaramoorthy, and P.I. Barton, *Worst-case design of subsea production facilities using semi-infinite programming*, AIChE Journal 60 (2014), pp. 2513–2524.

[135] Y. Sun, N.B. Agostini, S. Dong, and D. Kaeli, *Summarizing CPU and GPU design trends with product data*, https://arxiv.org/abs/1911.11313 (2019).

[136] J. Tang, R. Sorokin, E. Ignasheva, G. Jensen, L. Chen, J. Lee, A. Kulik, and M. Grundman, *Scaling on-device GPU inference for large generative models*, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 2025, pp. 6354–6363.

[137] M. Tawarmalani and N.V. Sahinidis, *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming: Theory, Algorithms, Software, and Applications*, Nonconvex Optimization and Its Applications, Springer, New York, NY, 2002.

[138] J.W. Taylor, *Direct measurement and analysis of cyclohexadienyl oxidation*, Ph.D. diss., Massachusetts Institute of Technology (2005).

[139] Theano Development Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J.B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.A. Côté, M. Côté, A. Courville, Y.N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S.E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J.A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.A. Manzagol, O. Mastropietro, R.T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I.V. Serban, D. Serdyuk, S. Shabanian, Étienne Simon, S. Spieckermann, S.R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D.J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, *Theano: A python framework for fast computation of mathematical expressions*, arXiv (2016). https://arxiv.org/abs/1605.02688.

[140] C. Touretzky, *Does Gurobi support GPUs?*, Online (2025), URL `https://support.gurobi.com/hc/en-us/articles/360012237852-Does-Gurobi-support-GPUs`. Accessed: Apr. 28, 2025.

[141] A. Tsoukalas and A. Mitsos, *Multivariate McCormick relaxations*, Journal of Global Optimization 59 (2014), pp. 633–662.

[142] B. van Merrienboer, D. Moldovan, and A. Wiltschko, *Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming*, in S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems* vol. 31, vol. 31. Curran As-

sociates, Inc., 2018, URL https://proceedings.neurips.cc/paper_files/paper/2018/file/748d6b6ed8e13f857ceaa6cfbdca14b8-Paper.pdf.

[143] B. van Merriënboer, A.B. Wiltschko, and D. Moldovan, *Tangent: Automatic differentiation using source code transformation in python*, arXiv preprint arXiv:1711.02712 (2017).

[144] R.J. Vanderbei, *Linear Programming: Foundations and Extensions*, Springer US, 2014.

[145] S. Vigerske and A. Gleixner, *SCIP: global optimization of mixed-integer nonlinear programs in a branch-and-cut framework*, Optimization Methods and Software 33 (2018), pp. 563–593.

[146] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. van der Walt, M. Brett, J. Wilson, K.J. Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, E. Larson, C.J. Carey, I. Polat, Y. Feng, E.W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E.A. Quintero, C.R. Harris, A.M. Archibald, A.H. Ribeiro, F. Pedregosa, P. van Mulbregt, A. Vijaykumar, A.P. Bardelli, A. Rothberg, A. Hilboll, A. Kloeckner, A. Scopatz, A. Lee, A. Rokem, C.N. Woods, C. Fulton, C. Masson, C. Häggström, C. Fitzgerald, D.A. Nicholson, D.R. Hagen, D.V. Pasechnik, E. Olivetti, E. Martin, E. Wieser, F. Silva, F. Lenders, F. Wilhelm, G. Young, G.A. Price, G.L. Ingold, G.E. Allen, G.R. Lee, H. Audren, I. Probst, J.P. Dietrich, J. Silterra, J.T. Webber, J. Slavič, J. Nothman, J. Buchner, J. Kulick, J.L. Schönberger, J.V. de Miranda Cardoso, J. Reimer, J. Harrington, J.L.C. Rodríguez, J. Nunez-Iglesias, J. Kuczynski, K. Tritz, M. Thoma, M. Newville, M. Kümmerer, M. Bolingbroke, M. Tartre, M. Pak, N.J. Smith, N. Nowaczyk, N. Shebanov, O. Pavlyk, P.A. Brodtkorb, P. Lee, R.T. McGibbon, R. Feldbauer, S. Lewis, S. Tygier, S. Sievert, S. Vigna, S. Peterson, S. More, T. Pudlik, T. Oshima, T.J. Pingel, T.P. Robitaille, T. Spura, T.R. Jones, T. Cera, T. Leslie, T. Zito, T. Krauss, U. Upadhyay, Y.O. Halchenko, and Y. Vázquez-Baeza, *SciPy 1.0: fundamental algorithms for scientific computing in Python*, Nature Methods 17 (2020), pp. 261–272.

[147] T.T. Vu and B. Derbel, *Parallel branch-and-bound in multi-core multi-CPU multi-GPU heterogeneous environments*, Future Generation Computer Systems 56 (2016), pp. 95–109.

[148] A. Wächter and L.T. Biegler, *On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming*, Mathematical Programming 106 (2006), pp. 25–57.

[149] A. Wechsung and P.I. Barton, *Global optimization of bounded factorable functions with discontinuities*, Journal of Global Optimization 58 (2014), pp. 1–30.

[150] A. Wechsung, S.D. Schaber, and P.I. Barton, *The cluster problem revisited*, Journal of Global Optimization 58 (2014), pp. 429–438.

[151] A. Wechsung, J.K. Scott, H.A.J. Watson, and P.I. Barton, *Reverse propagation of McCormick relaxations*, Journal of Global Optimization 63 (2015), pp. 1–36.

[152] M.J. White, *Nvidia says falling GPU prices are 'a story of the past'*, Digital Trends (2022), URL `https://www.digitaltrends.com/computing/nvidia-says-falling-gpu-prices-are-over/`.

[153] M.E. Wilhelm, R.X. Gottlieb, and M.D. Stuber, *PSORLab/McCormick.jl v0.14.0*, `https://github.com/PSORLab/McCormick.jl` (2020). Accessed 2026-04-03.

[154] M.E. Wilhelm, A.V. Le, and M.D. Stuber, *Global optimization of stiff dynamical systems*, AIChE Journal 65 (2019), pp. 1–20.

[155] M.E. Wilhelm and M.D. Stuber, *EAGO.jl: Easy Advanced Global Optimization in Julia*, Optimization Methods and Software 37 (2022), pp. 425–450.

[156] M.E. Wilhelm, C. Wang, and M.D. Stuber, *Convex and concave envelopes of artificial neural network activation functions for deterministic global optimization*, Journal of Global Optimization 85 (2022), pp. 569–594.

[157] Y. Yajima, *Convex envelopes in optimization problems*, in *Encyclopedia of Optimization*, Springer US (2001), pp. 343–344.

[158] J. Ye and J.K. Scott, *Extended McCormick relaxation rules for handling empty arguments representing infeasibility*, Journal of Global Optimization 87 (2023), pp. 57–95.

[159] Y. Ye and E. Tse, *An extension of karmarkar's projective algorithm for convex quadratic programming*, Mathematical Programming 44 (1989), pp. 157–179.

[160] H. Zhang, T. Kerkenhoff, N. Kichler, M. Dahmen, A. Mitsos, U. Naumann, and D. Bongartz, *Accelerating deterministic global optimization via GPU-parallel interval arithmetic*, arXiv (2025). https://arxiv.org/abs/2507.20769.

[161] Y. Zhang and N.V. Sahinidis, *Solving continuous and discrete nonlinear programs with BARON*, Computational Optimization and Applications 92 (2024), pp. 1123–1161.