GPU-Parallel Branch-and-Bound with Custom Kernels and Specialized PDLP

Robert X. Gottlieb[1] and Matthew D. Stuber[1]

[1]*Chemical & Biomolecular Engineering, University of Connecticut, Storrs, CT, USA*

Abstract Two novel developments are detailed that are critical to hardware acceleration of deterministic global optimization: automatic relaxation subgradient calculation on a GPU for LP relaxations, and a native GPU-parallel LP solver. To calculate relaxation subgradients, we rely on a novel method to automatically generate problem- and expression-specific GPU-compatible functions, with preliminary results indicating a 10–100x speedup over a CPU implementation of McCormick-based rules. To address LPs, we developed a novel GPU-accelerated variant of the PDLP method. Preliminary results show increased performance of up to 100x relative to commercial solvers, specifically in the case of solving large numbers of small-sized LP instances. Subsequently, we combine these two developments to form a GPU-accelerated global optimizer. The performance of these developments are demonstrated on a range of problems, across different hardware capabilities, and against existing commercial solvers.

Keywords: SIMD Parallelization, Global Optimization, Branch-and-Bound

1. Introduction

Modern tools for deterministic global optimization (DGO) use a variety of advanced techniques to solve problems that were once considered intractable. Such techniques include reformulations to more easily solvable problem forms [1], specialized domain reduction techniques [2], and the development of improved relaxations or envelopes of critical functional forms [3, 4], among others. However, these techniques alone are not suffi-

cient to render all DGO problems tractable, and for many problems of practical interest, obtaining a solution still requires processing thousands or millions of branch-and-bound (B&B) nodes. For these problems, processing power and calculation throughput are critical factors that determine how quickly a global solution can be found. In other fields that faced a similar computational bottleneck, such as in machine learning model training [5], the dominant strategy has become the use of parallel processing hardware such as graphics processing units (GPUs). While new tools and techniques continue to make many DGO problems solvable, a move toward parallelization using GPU hardware would serve to increase the solution speed for a wide variety of the most challenging problems via faster calculation throughput.

To the best of our knowledge, all existing DGO solvers—besides our solver [6]—are designed to run exclusively on CPU hardware. For a GPU-based global solver, one of the challenges is aligning the B&B algorithm with the strengths of the hardware. Particularly, GPU-based local solvers typically become more effective on large-scale instances, such as the NLP solver `MadNLP.jl` [8], which realizes a speedup over a CPU implementation on problem instances with more than 20,000 variables. DGO problems are typically several orders-of-magnitude smaller than this, which implies that individual bounding subproblems are unlikely to benefit from GPU acceleration. Instead, in this work, we focus on simultaneously processing large numbers of B&B nodes, aligning the size of the B&B node stack with the parallelization capacity of GPUs.

The state-of-the-art approach for obtaining B&B node lower bounds is to construct linear programs (LPs) from subtangent hyperplanes of relaxations of the objective and constraints. Consequently, and as discussed in [7], a GPU-based DGO solver would need two key features to be competitive with existing solvers: a method of calculating relaxation subgradients, and a method of solving LPs. In this paper, we detail our development of GPU-parallel implementations of these two features, each with the goal of addressing many B&B nodes simultaneously.

2. Implementations and Results

GPU-Parallel Relaxation Subgradients. In this work, we utilize a source-code generation approach to compute McCormick-based relaxations of functions on a GPU—conceptually similar to our previous work [6]—in an entirely novel way. This approach contrasts the operator overloading-like scheme used by `McCormick.jl` [9]. Given an input expression, such as a symbolically defined objective function or constraint, we begin by per-

forming a factorization of the expression to generate a primal trace. We then apply transformations as desired, such as automatically identifying mathematical forms for which specialized relaxation rules exist and substituting those expressions. From this modified primal trace, we are prepared to construct numerical functions for evaluation.

Finally, the primal trace is parsed and a GPU-compatible function (a "kernel") is automatically generated that sequentially applies generalized McCormick rules to its inputs (variables and their domains) to calculate relaxations (and their subgradients) of the original expression on the given variable domains. The generation of such kernels, valid on any subdomain, is possible because the objective and constraints of a problem are identical across B&B nodes. This also allows us to exploit problem sparsity, both here and in our LP solver, to speed up calculations. These custom, problem-specific kernels apply GPU parallelization to return relaxations for many B&B nodes simultaneously, each of which necessarily has a unique domain. From preliminary testing, this approach outperforms `McCormick.jl` [9] over a range of expression complexities by roughly 10–100x.

GPU-Parallel LP Solver. Standard approaches for solving LPs are mostly variants of the simplex and interior point methods, although in recent years, first-order methods have become more popular due to their potential for easier GPU acceleration. A notable recent development is the PDLP algorithm [10], and its open-source GPU implementation `cuPDLP.jl` [11]. In this work, we develop a `cuPDLP.jl`-based solver that scales with the number of LPs solved as opposed to instance size, with the primary focus being the smaller LPs encountered in B&B lower-bounding procedures. Additionally, since the goal is to incorporate this solver into a B&B scheme, and we generate relaxation subgradients on GPUs, we eliminate the expensive step of transferring LP instances between CPU and GPU memory. Effectively, the global optimizer would handle node organization, domain reduction, and the upper-bounding problems using the CPU, and would offload the entirety of the lower-bounding work, which is generally the most time-intensive step, to the GPU.

Similar to the novel relaxation approach presented in this work, the performance of this version of PDLP scales with the number of LPs being solved. Preliminary benchmark results shown in Table 1 indicate speed improvements of up to 100x relative to commercial LP solvers on small problems. We attribute this to the lack of CPU-GPU memory transfer, the specialized use case of handling multiple LPs, and the focus on comparatively small LP instances.

Table 1: Average solve times over 20480 LP instances generated from affine relaxations of partitioned examples from the MINLPLib database. Parallelized PDLP solves LPs in parallel in a GV100 GPU, and GLPK solves LPs in serial on an Intel Xeon W-2195 CPU. This version of GPU-accelerated PDLP is most dominant on extremely small LPs and at weaker tolerance.

Example	Variables	Constraints	Avg. Solution Time (μs)			
			Parallelized PDLP		GLPK	
			1E-8 Tol.	1E-4 Tol.	1E-8 Tol.	1E-4 Tol.
`ex4_1_1`	1	0	1.186	0.665	75.238	73.944
`rbrock`	2	0	0.661	0.323	78.462	75.092
`ex3_1_1`	8	6	5.676	5.538	104.361	103.909
`chem`	11	4	23.655	13.132	137.179	143.991
`ramsey`	33	22	18.066	18.102	204.618	213.732
`lakes`	90	78	749.429	749.268	1144.600	1155.039

3. Conclusion

In this work, we discuss our latest GPU-accelerated relaxation subgradient technique and describe the implementation of a GPU-parallelized LP solver based on PDLP. We aim to demonstrate the strengths of these approaches across numerical experiments and a range of hardware capabilities. We also aim to demonstrate how these approaches can be paired together into a GPU-accelerated DGO solver, and compare its performance on benchmark test sets against commercial solvers.

References

[1] Amarger, R.J. et al. (1992). An automated modelling and reformulation system for design optimization. *Comput. Chem. Eng*. 16(7): 623–636.

[2] Zhang, Y. et al. (2020). Optimality-based domain reduction for inequality-constrained NLP and MINLP problems. *J. Global Optim*. 77(3): 425–454.

[3] Wilhelm, M.E. and Stuber, M.D. (2023). Improved convex and concave relaxations of composite bilinear forms. *J. Optim. Theory and Appl*. 197: 174–204.

[4] Wilhelm, M.E. et al. (2022). Convex and concave envelopes of artificial neural network activation functions for deterministic global optimization. *J. Global Optim*. 85: 569–594.

[5] Abadi, M. et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. doi: 10.48550/ARXIV.1603.04467

[6] Gottlieb, R.X. et al. (2024). Automatic source code generation for deterministic global optimization with parallel architectures. *Optim. Methods Software*. 1–39.

[7] Gottlieb, R.X. and Stuber, M.D. (2024). GPU-accelerated deterministic global optimization. In: ISMP 2024, Montréal, Canada.

[8] Shin, S. et al. (2024). Accelerating optimal power flow with GPUs: SIMD abstraction of nonlinear programs and condensed-space interior-point methods. *Electr. Power Syst. Res*. 236: 110651.

[9] Wilhelm, M.E. et al. (2020). PSORLab/McCormick.jl. URL: `https://github.com/PSORLab/McCormick.jl`

[10] Applegate, D., et al. (2021). Practical large-scale linear programming using primal-dual hybrid gradient. doi: 10.48550/ARXIV.2106.04756.

[11] Lu, H. and Yang, J. (2024). cuPDLP.jl: A GPU implementation of restarted primal-dual hybrid gradient for linear programming in Julia. doi: 10.48550/ARXIV.2311.12180.