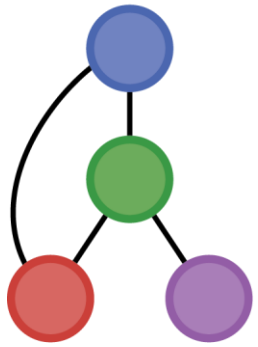
UCONN
UNIVERSITY OF CONNECTICUT

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

Robert Gottlieb, Matthew Stuber

Sept. 4, 2025

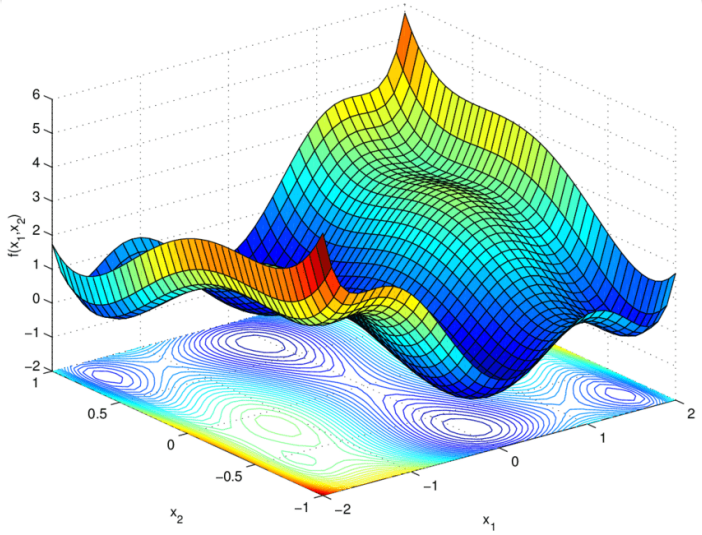


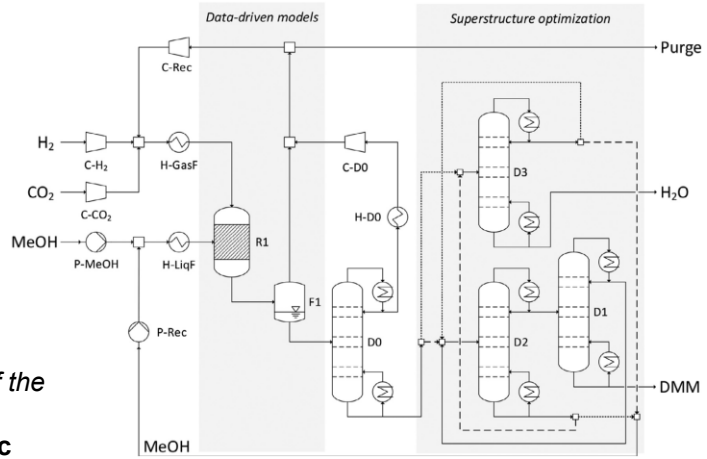
Process Systems and Operations Research Laboratory

Deterministic Global Optimization

- Focus is **nonconvex (MI)NLPs**
- Standard approach is **spatial B&B**

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



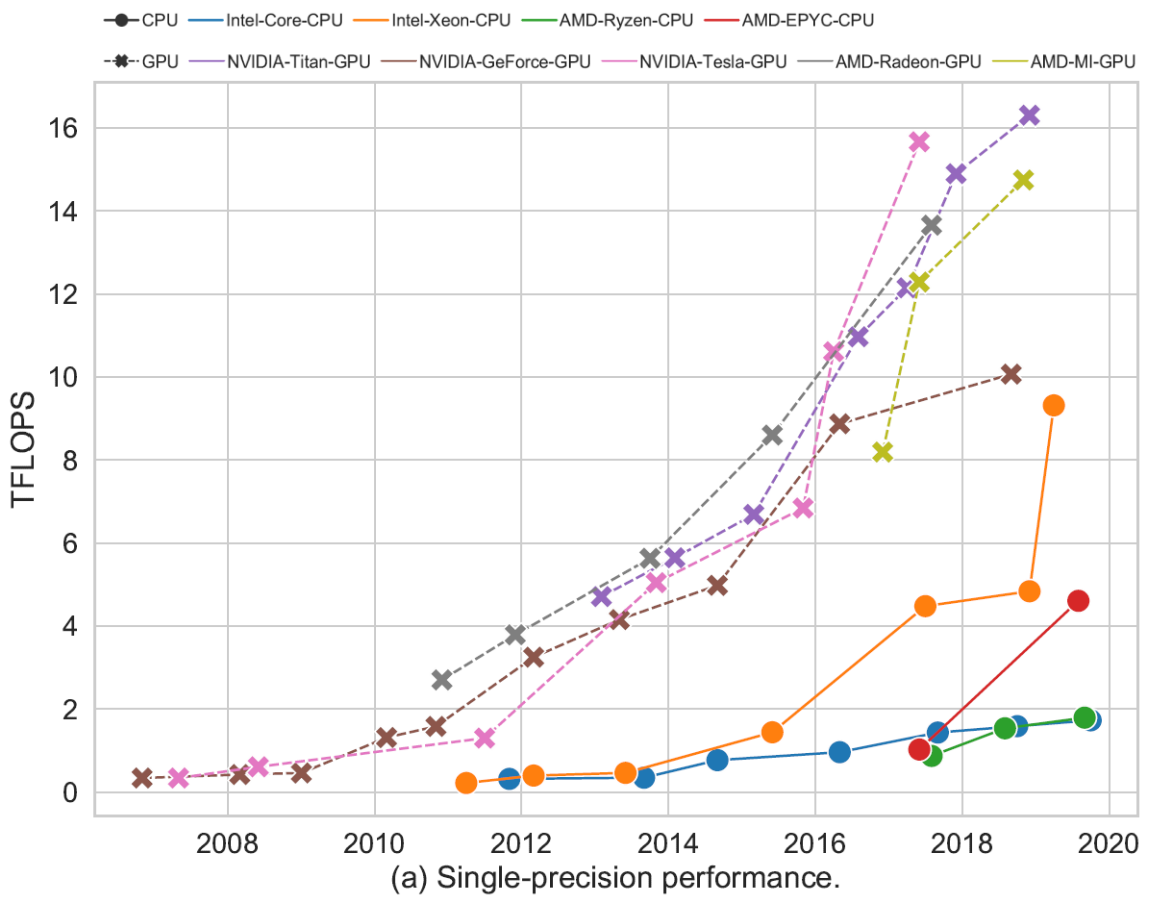


1. D. Henrion and J.-B. Lasserre. **GloptiPoly: global optimization over polynomials with MATLAB and SeDuMi.** *In Proceedings of the 41st IEEE Conference on Decision and Control* (2002).
2. Burre, J., et al. **Global flowsheet optimization for reductive dimethoxymethane production using data-driven thermodynamic models.** *Computers & Chemical Engineering*, (2022): 107806.

High-Performance Computing

Graphics Processing Units (GPUs)

- Graphics rendering
- Machine learning model training[4]
  - Generative AI
- Data analysis[5]
- Large-scale simulations[6]
  - Molecular dynamics
  - CFD modeling
- Supercomputing
- […]

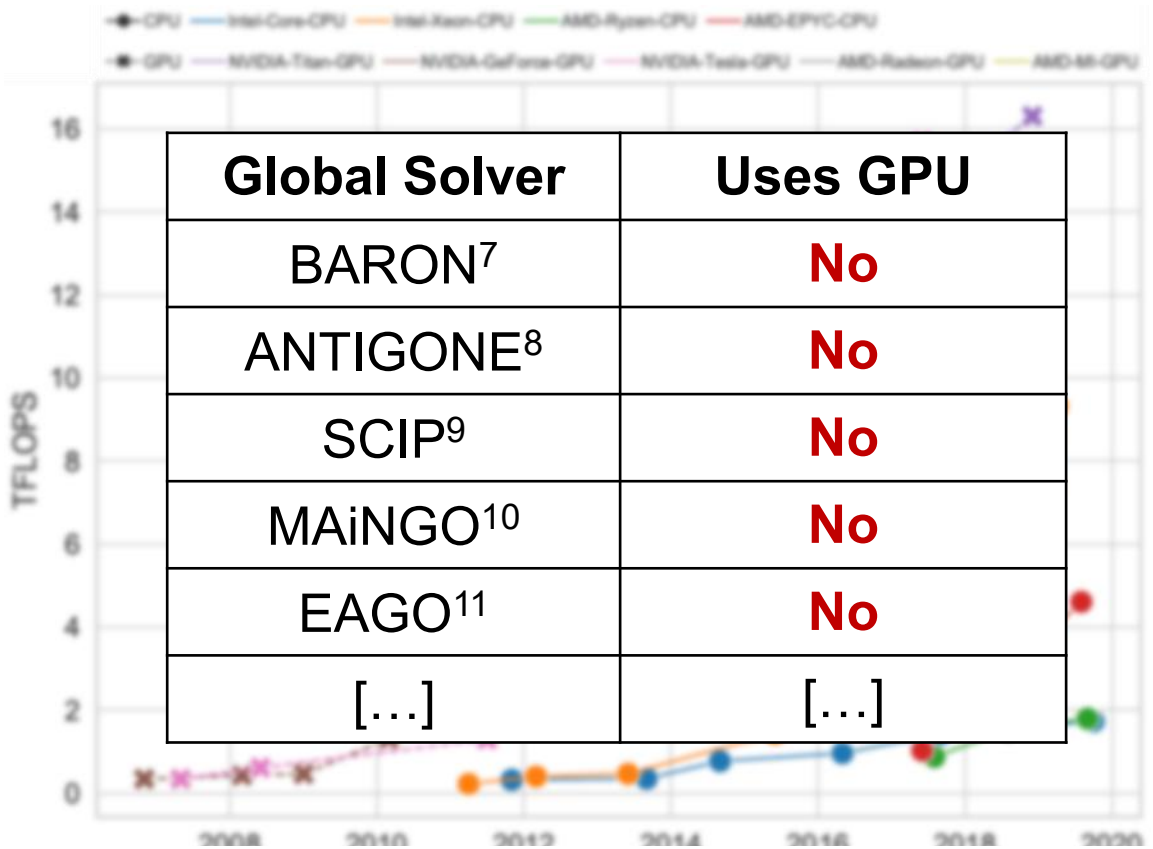


3. Sun, Y., et al. **Summarizing CPU and GPU design trends with product data.** *arXiv*, (2019). arXiv: 1911.11313
4. Steinkraus, D., et al. **Using GPUs for machine learning algorithms**. *Eighth International Conference on Document Analysis and Recognition*, Seoul, South Korea, 2: 1115-1120 (2005).
5. Singh, H., et al. **GPU and CUDA in Hard Computing Approaches: Analytical Review**. *Proceedings of ICRIC 2019*, 177-196 (2020).
6. Stone, J.E., et al. **GPU-accelerated molecular modeling coming of age.** *Journal of Molecular Graphics and Modelling*, 29(2):116-125 (2010).

High-Performance Computing

Graphics Processing Units (GPUs)

- Graphics rendering
- Machine learning model training[4]
  - Generative AI
- Data analysis[5]
- Large-scale simulations[6]
  - Molecular dynamics
  - CFD modeling
- Supercomputing

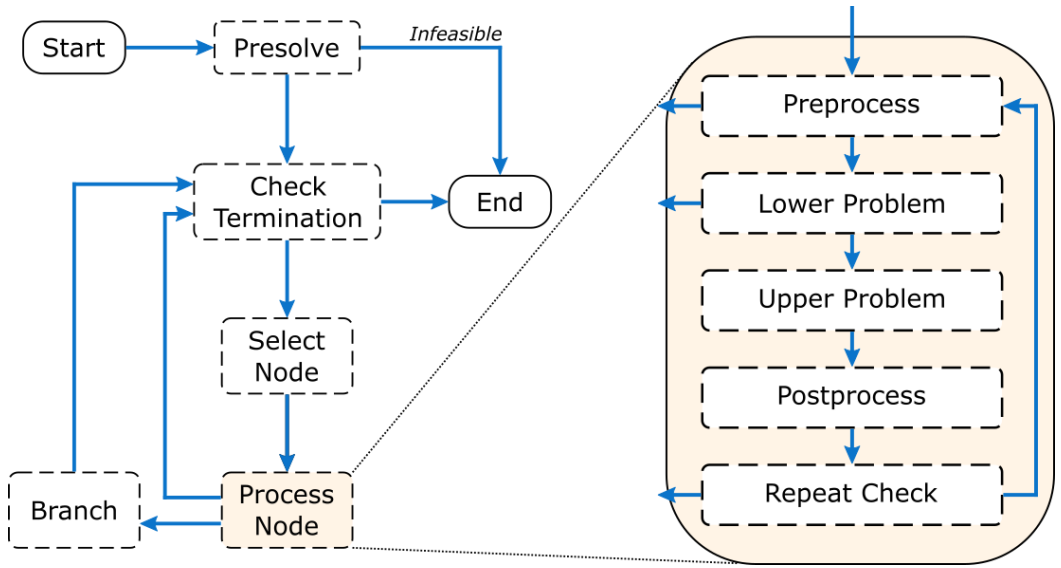


Global Solver	Uses GPU
BARON[7]	No
ANTIGONE[8]	No
SCIP[9]	No
MAiNGO[10]	No
EAGO[11]	No
[…]	[…]

7. Sahinidis, N.V. **BARON: A general purpose global optimization software package.** *Journal of Global Optimization*, 8(2):201–205 (1996).
8. Misener, R. and Floudas, C.A. **ANTIGONE: Algorithms for coNTinuous / Integer Global Optimization of Nonlinear Equations.** *Journal of Global Optimization*, 59(2-3):503–526 (2014).
9. Vigerske, S. and Gleixner, A.. **SCIP: global optimization of mixed-integer non-linear programs in a branch-and-cut framework.** *Optimization Methods and Software*, 33(3):563–593 (2017).
10. Bongartz, D., et al. **MAiNGO - McCormick-based Algorithm for mixed-integer Nonlinear Global Optimization.** Technical report, RWTH-Aachen (2018). URL https://www.avt.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaabclahw.
11. Wilhelm, M.E. and Stuber, M.D. **EAGO.jl: easy advanced global optimization in Julia.** *Optimization Methods and Software*, 37(2):425–450 (2022).

Aligning B&B with GPUs

B&B Algorithm



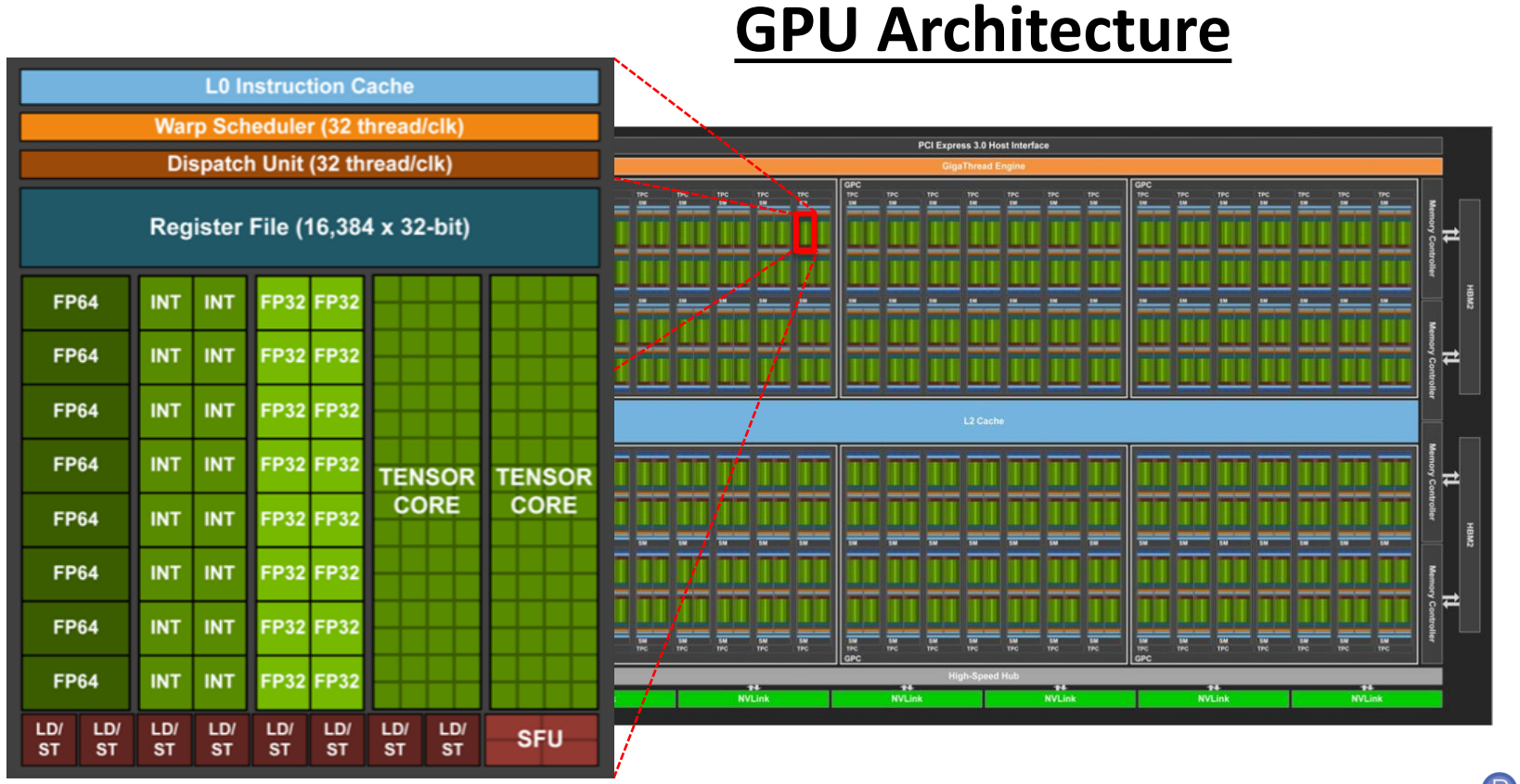
?

GPU Architecture

11. Wilhelm, M.E. and Stuber, M.D. **EAGO.jl: easy advanced global optimization in Julia.** *Optimization Methods and Software*, 37(2):425–450 (2022).
12. NVIDIA. **NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU** [White paper]. NVIDIA (2017).

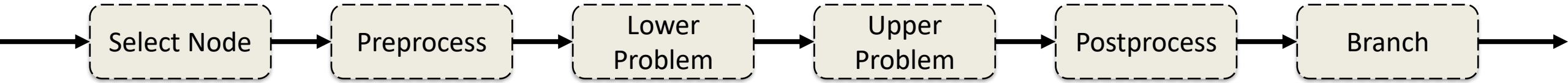
GPU Architecture

- GPUs are composed of **thousands of cores**
- Single instructions are executed by **many cores simultaneously** on different portions of data

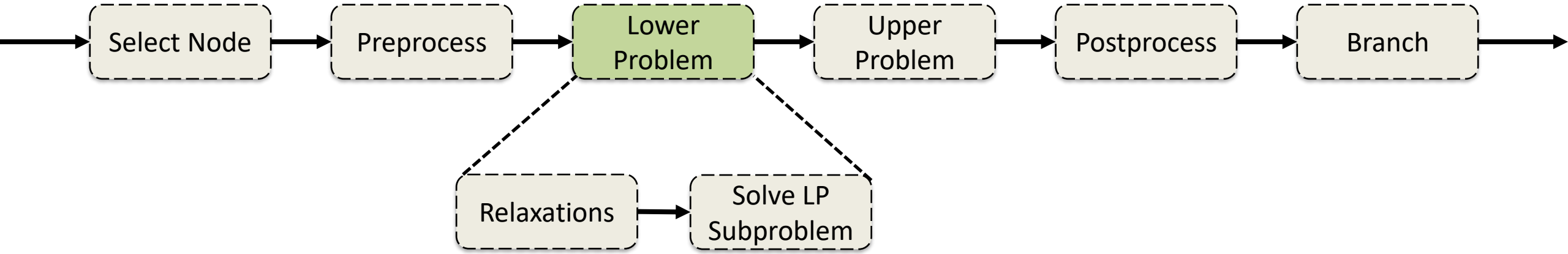


12. NVIDIA. **NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU** [White paper]. NVIDIA (2017).

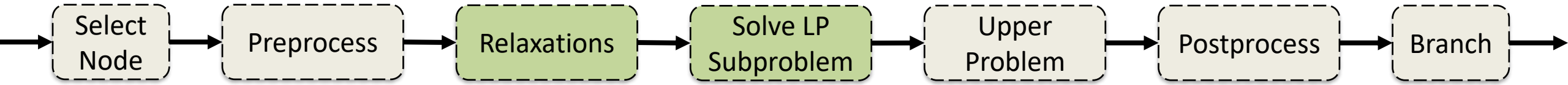
Serial CPU B&B



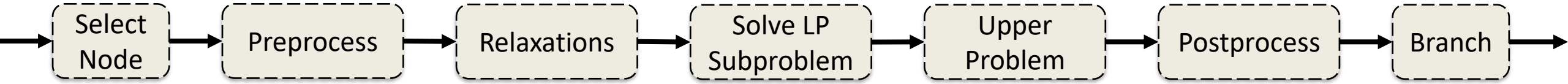
Serial CPU B&B

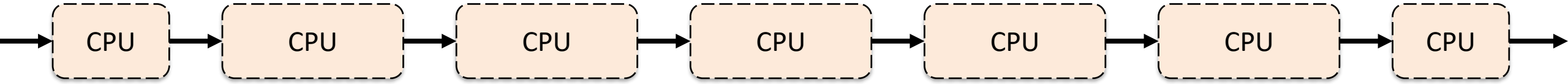


Serial CPU B&B

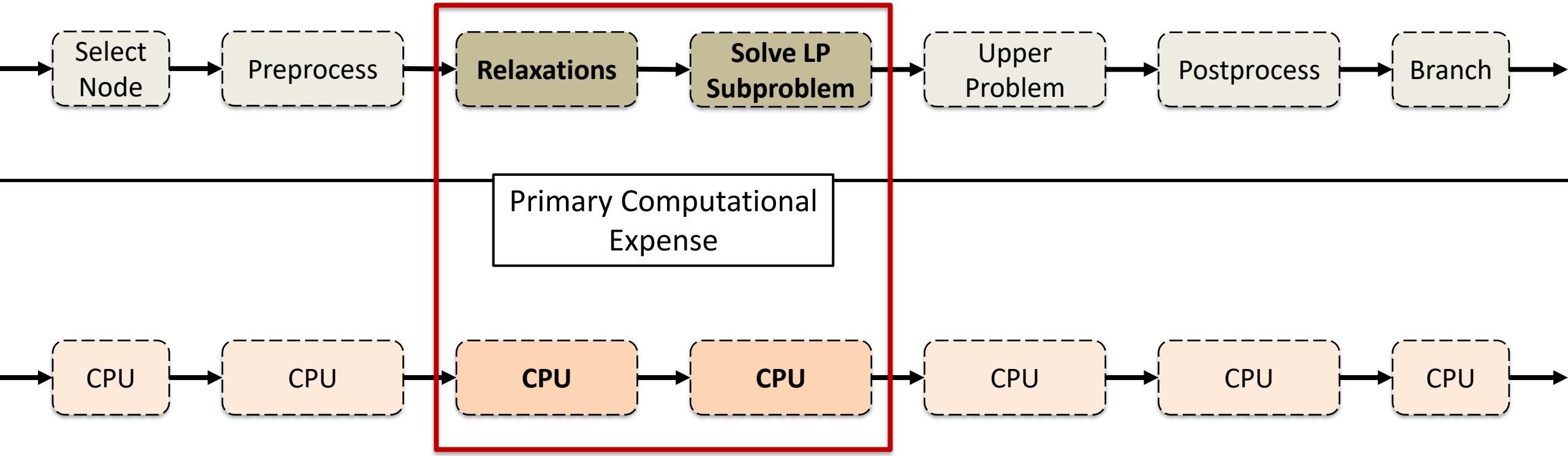


Serial CPU B&B

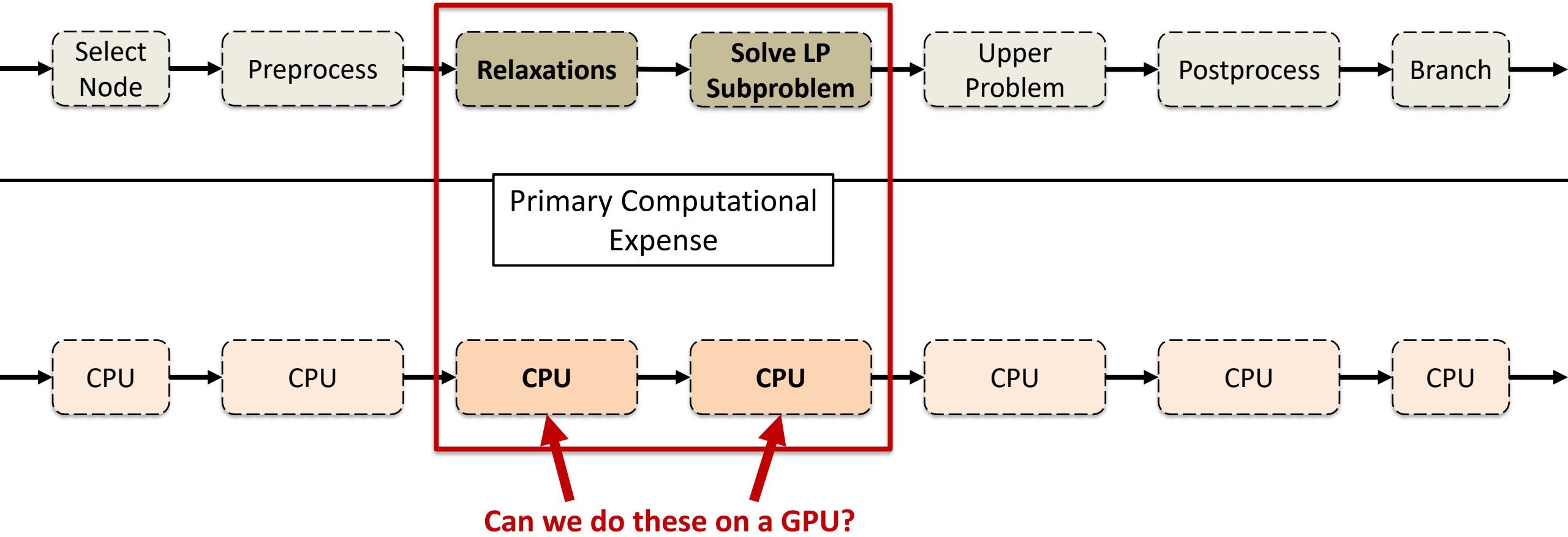




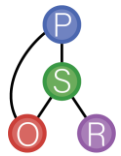
Serial CPU B&B



Serial CPU B&B



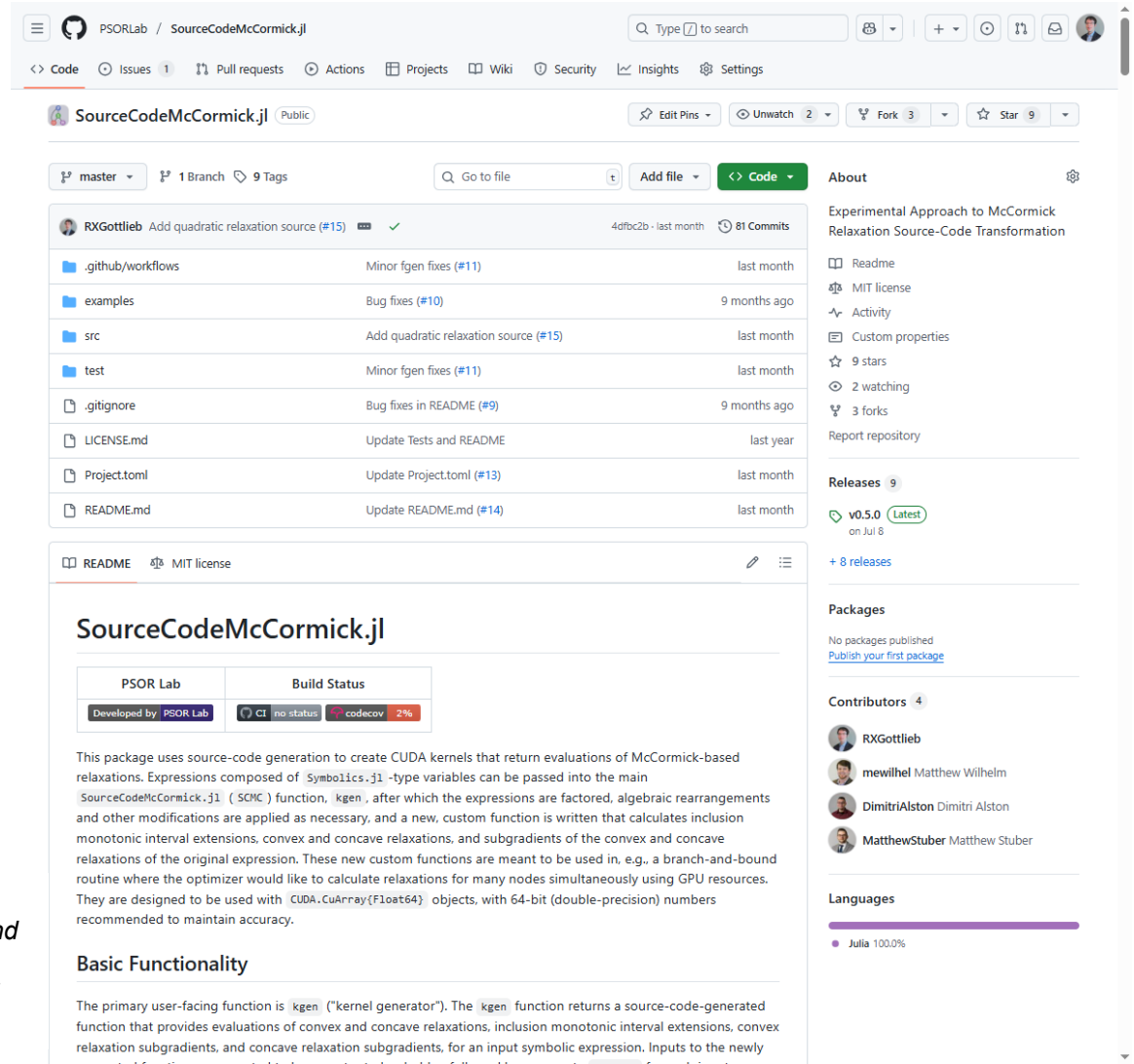
1) Relaxations



SourceCodeMcCormick.jl

SourceCodeMcCormick.jl (SCMC)[13,14]

- Based on source code generation
- Enables GPU-accelerated:
  - Inclusion monotonic interval extensions
  - McCormick relaxations
  - Subgradients of McCormick relaxations



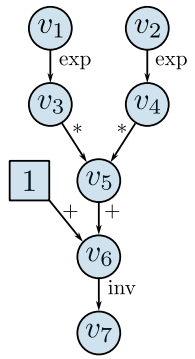
13. Gottlieb, R.X., Xu, P., and Stuber, M.D. **Automatic Source Code Generation for Deterministic Global Optimization with Parallel Architectures.** *Optimization Methods and Software*, 1–39 (2024).
14. Gottlieb, R.X. and Stuber, M.D. **Automatic Generation of GPU Kernels for Evaluators of McCormick-Based Relaxations and Subgradients.** Under Review, (2025).

SourceCodeMcCormick.jl

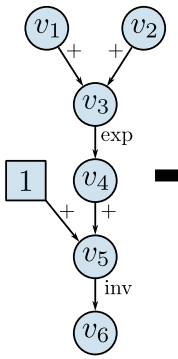
- Generates customized CUDA kernels to calculate **many relaxations at once**

```
using SourceCodeMcCormick
@variables x, y
func = kgen(1 / (1 + exp(x)*exp(y)))
```

$$
\begin{aligned}
v_1 &= x \\
v_2 &= y \\
v_3 &= \exp(v_1) \\
v_4 &= \exp(v_2) \\
v_5 &= v_3 v_4 \\
v_6 &= 1 + v_5 \\
v_7 &= v_6^{-1}
\end{aligned}
$$



$$
\begin{aligned}
v_1 &= x \\
v_2 &= y \\
v_3 &= v_1 + v_2 \\
v_4 &= \exp(v_3) \\
v_5 &= 1 + v_4 \\
v_6 &= v_5^{-1}
\end{aligned}
$$




```
# Generated at 2025-08-22T17:22:03.345

# Kernel(s) generated for the expression: 1 / (1 + exp(y)*exp(x))

function f_B3WZmzXUeWL_1(OUT, x, y)
    idx = threadIdx().x + (blockIdx().x - Int32(1)) * blockDim().x
    stride = blockDim().x * gridDim().x
    col = Int32(1)
    colmax = Int32(2)
    temp1_lo = 0.0
    temp1_hi = 0.0
    temp1_cv = 0.0
    temp1_cc = 0.0
    temp1_cvgrad = @MVector zeros(Float64, 2)
    temp1_ccgrad = @MVector zeros(Float64, 2)
    temp2_lo = 0.0
    temp2_hi = 0.0
    temp2_cv = 0.0
    temp2_cc = 0.0
    temp2_cvgrad = @MVector zeros(Float64, 2)
    temp2_ccgrad = @MVector zeros(Float64, 2)
    while idx <= Int32(size(OUT,1))
        ################################
        ## Addition of Two Variables ##
        ################################

        # Reset the column counter
        col = Int32(1)

        # Begin rule
```

Automatically Generated

13. Gottlieb, R.X., Xu, P., and Stuber, M.D. **Automatic Source Code Generation for Deterministic Global Optimization with Parallel Architectures.** *Optimization Methods and Software*, 1–39 (2024).
14. Gottlieb, R.X. and Stuber, M.D. **Automatic Generation of GPU Kernels for Evaluators of McCormick-Based Relaxations and Subgradients.** Under Review, (2025).

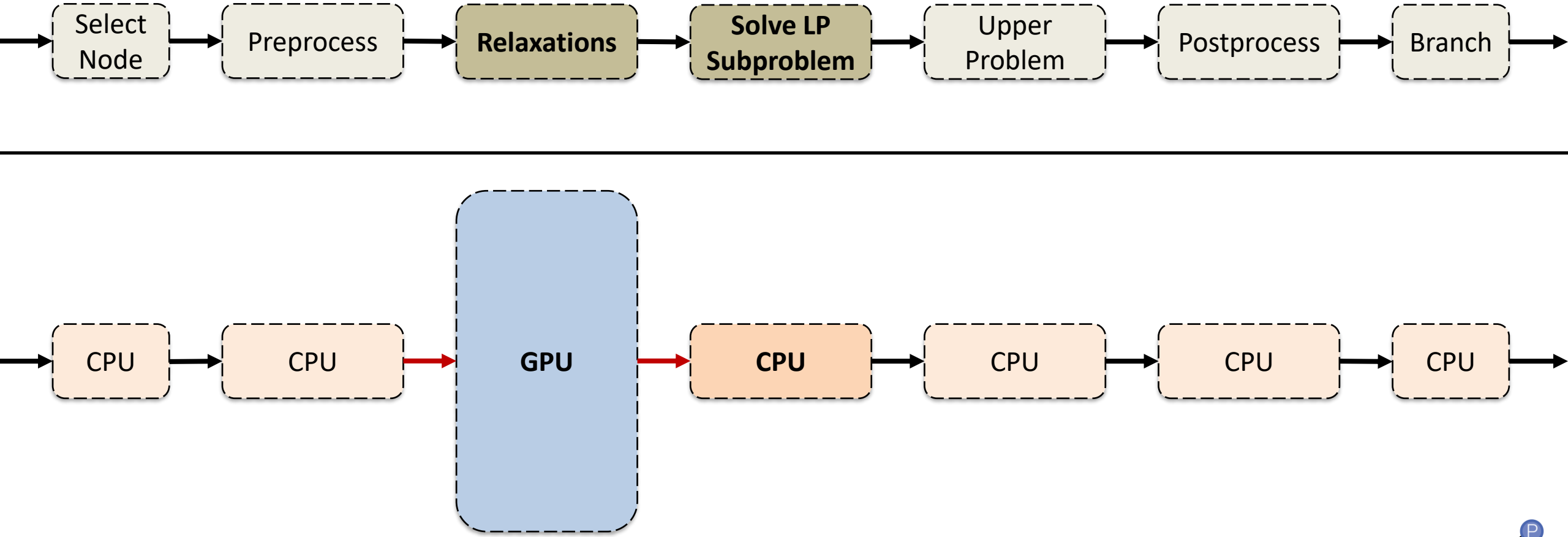
SourceCodeMcCormick.jl

Expression Form	Initial Domain	Dimensionality	Nonlinear Operators	SCMC Avg. Evaluation Time (ns)	McCormick.jl Avg. Evaluation Time (ns)	Evaluation Time Speed-Up
$\sum_{i=1}^{10} x_i$	$[-1,1]^{10}$	10	(None)	1.9×10^{0}	1.90×10^{2}	99.9x
$\exp(\sum_{i=1}^{10} x_i)$	$[-1,1]^{10}$	10	exp	3.0×10^{0}	2.72×10^{2}	90.6x
$(\sum_{i=1}^{10} x_i)^2$	$[-1,1]^{10}$	10	^2	3.1×10^{0}	2.36×10^{2}	76.2x
$\prod_{i=1}^{10} x_i$	$[-1,1]^{10}$	10	*	1.15×10^{1}	5.79×10^{2}	50.3x
$\exp(\prod_{i=1}^{10} x_i)$	$[-1,1]^{10}$	10	*, exp	1.19×10^{1}	6.38×10^{2}	53.6x
$(\prod_{i=1}^{10} (x_i)^2)$	$[-1,1]^{10}$	10	*, ^2	1.02×10^{1}	9.40×10^{2}	92.1x
$x_1/(\prod_{i=2}^{10} x_i)$	$[-1,1] \times [0.01,1]^9$	10	/, *	6.2×10^{0}	5.11×10^{2}	82.4x
alkyl objective	$[0,2.0] \times [0,1.6] \times [0,1.2] \times [0,5.0] \times [0,2.0] \times [0.9,0.95]$	6	*	$7. \times 10^{-1}$	1.61×10^{2}	229.9x
ex6_2_10 objective	$[10^{-7},0.2]^2 \times [10^{-7},0.4]^4$	6	/, *, log	9.04×10^{1}	1.37×10^{4}	151.1x
arki0002 objective	$[-10,10]^{152}$	152	^2	2.58×10^{3}	1.62×10^{7}	6269.5x
Trained ANN	$[-5,5]^8$	8	*, sigmoid	1.90×10^{1}	5.51×10^{3}	290.0x
Training ANN	$[-5,5]^{31}$	31	*, sigmoid	9.59×10^{1}	1.28×10^{4}	133.4x

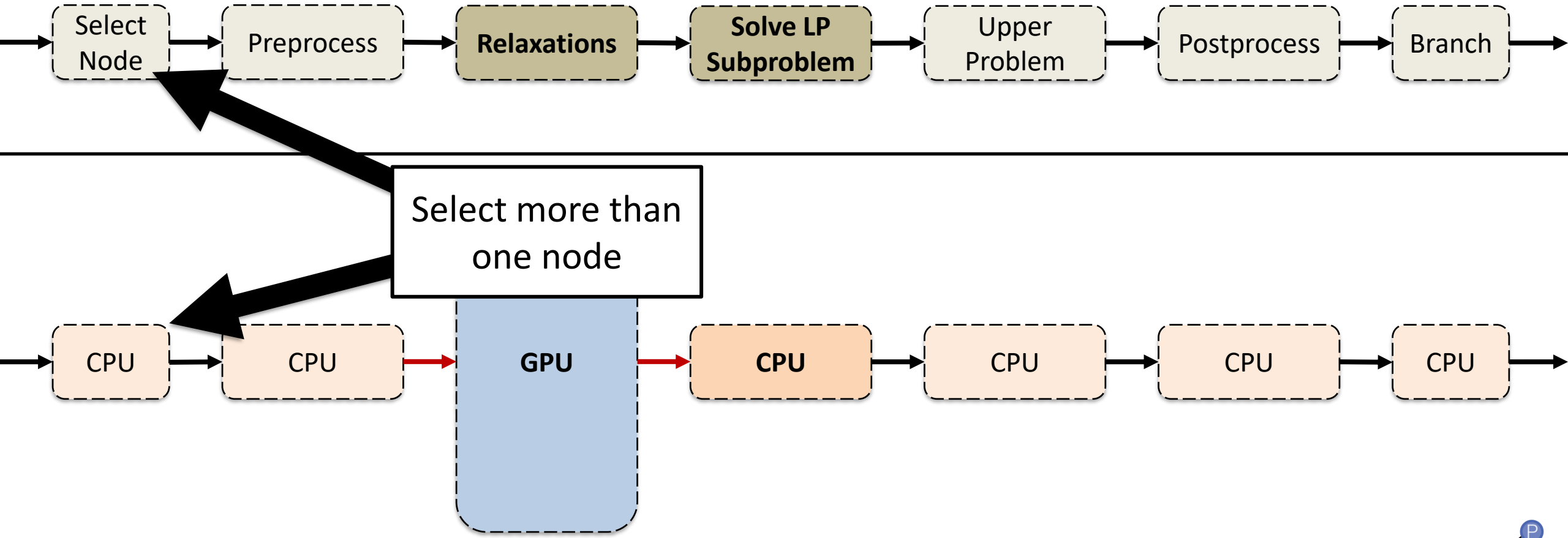
Speedup for 20480 relaxation evaluations

14. Gottlieb, R.X. and Stuber, M.D. **Automatic Generation of GPU Kernels for Evaluators of McCormick-Based Relaxations and Subgradients.** Under Review, (2025).

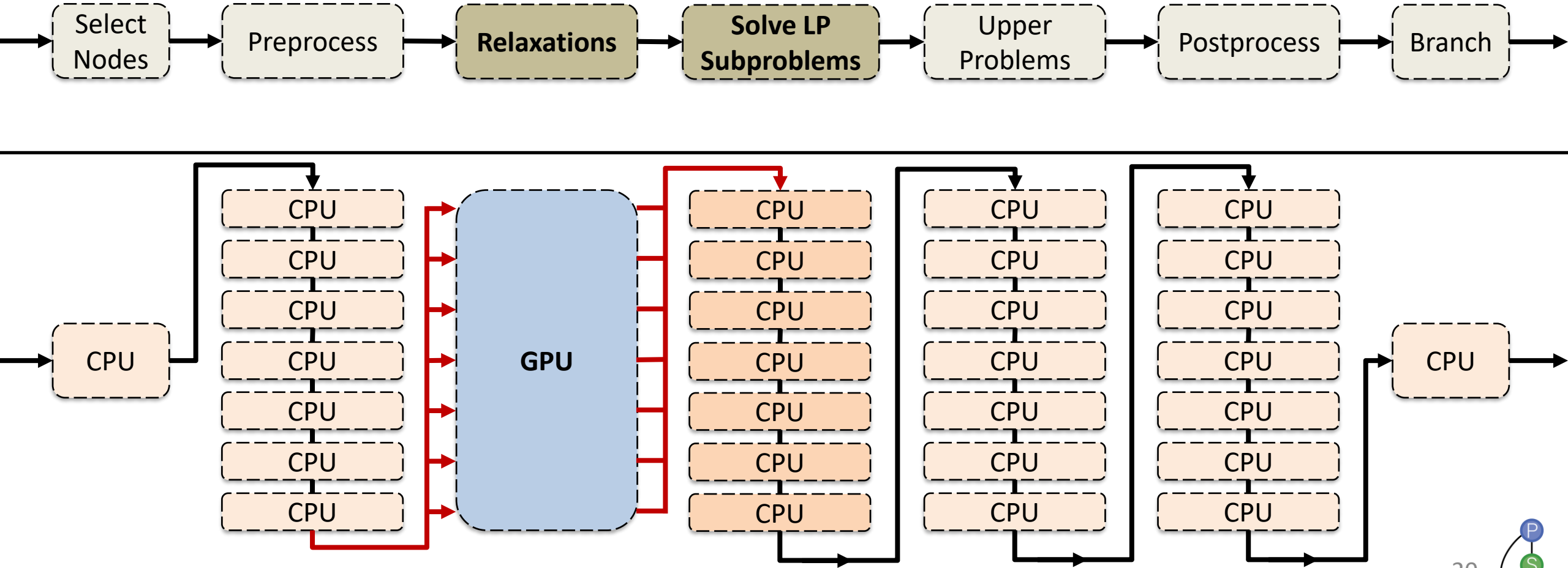
Hybrid CPU-GPU B&B



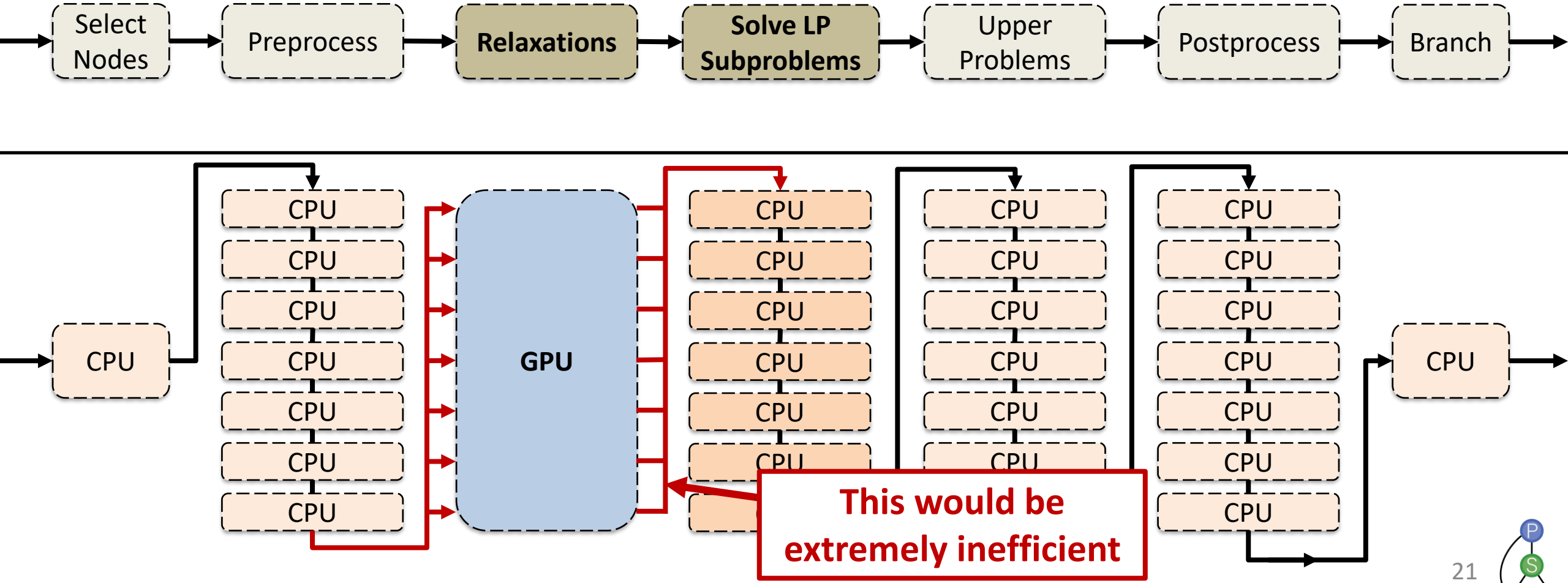
Hybrid CPU-GPU B&B



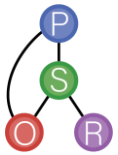
Hybrid CPU-GPU B&B



Hybrid CPU-GPU B&B



2) Solving LPs



PDLP

Primal Dual Hybrid Gradient for LP (PDLP)

- Applegate et al. (2021) developed a competitive first-order method (FOM) for solving LPs

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^\top x \\ \text{subject to:} \quad & Gx \geq h \\ & Ax = b \\ & l \leq x \leq u \end{aligned} \qquad \begin{aligned} \underset{y\in\mathbb{R}^{m_1+m_2},\lambda\in\mathbb{R}^n}{\text{maximize}} \quad & q^\top y + l^\top \lambda^+ - u^\top \lambda^- \\ \text{subject to:} \quad & c - K^\top y = \lambda \\ & y_{1:m_1} \geq 0 \\ & \lambda \in \Lambda\,, \end{aligned}$$

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

Practical Large-Scale Linear Programming using Primal-Dual Hybrid Gradient

David Applegate
Google Research
dapplegate@google.com

Mateo Díaz
California Institute of Technology*
mateodd@caltech.edu

Oliver Hinder
Google Research
University of Pittsburgh
ohinder@pitt.edu

Haihao Lu
University of Chicago†
haihao.lu@chicagobooth.edu

Miles Lubin
Google Research
mlubin@google.com

Brendan O'Donoghue
DeepMind
bodonoghue@deepmind.com

Warren Schudy
Google Research
wschudy@google.com

Abstract

We present PDLP, a practical first-order method for linear programming (LP) that can solve to the high levels of accuracy that are expected in traditional LP applications. In addition, it can scale to very large problems because its core operation is matrix-vector multiplications. PDLP is derived by applying the primal-dual hybrid gradient (PDHG) method, popularized by Chambolle and Pock (2011), to a saddle-point formulation of LP. PDLP enhances PDHG for LP by combining several new techniques with older tricks from the literature; the enhancements include diagonal preconditioning, presolving, adaptive step sizes, and adaptive restarting. PDLP improves the state of the art for first-order methods applied to LP. We compare PDLP with SCS, an ADMM-based solver, on a set of 383 LP instances derived from MIPLIB 2017. With a target of 10^{-8} relative accuracy and 1 hour time limit, PDLP achieves a 6.3x reduction in the geometric mean of solve times and a 4.6x reduction in the number of instances unsolved (from 227 to 49). Furthermore, we highlight standard benchmark instances and a large-scale application (PageRank) where our open-source prototype of PDLP, written in Julia, outperforms a commercial LP solver.

15. Applegate, D., et al. **Practical Large-Scale Linear Programming using Primal-Dual Hybrid Gradient.** *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, (2021).

PDLP

Primal Dual Hybrid Gradient for LP (PDLP)

- Applegate et al. (2021) developed a competitive first-order method (FOM) for solving LPs

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

$$x^{k+1} = \underset{X}{\mathbf{proj}}(x^k - \tau(c - K^\top y^k))$$
$$y^{k+1} = \underset{Y}{\mathbf{proj}}(y^k + \sigma(q - K(2x^{k+1} - x^k)))$$

15. Applegate, D., et al. **Practical Large-Scale Linear Programming using Primal-Dual Hybrid Gradient.** *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, (2021).

Practical Large-Scale Linear Programming using Primal-Dual Hybrid Gradient

David Applegate
Google Research
dapplegate@google.com

Mateo Díaz
California Institute of Technology*
mateodd@caltech.edu

Oliver Hinder
Google Research
University of Pittsburgh
ohinder@pitt.edu

Haihao Lu
University of Chicago†
haihao.lu@chicagobooth.edu

Miles Lubin
Google Research
mlubin@google.com

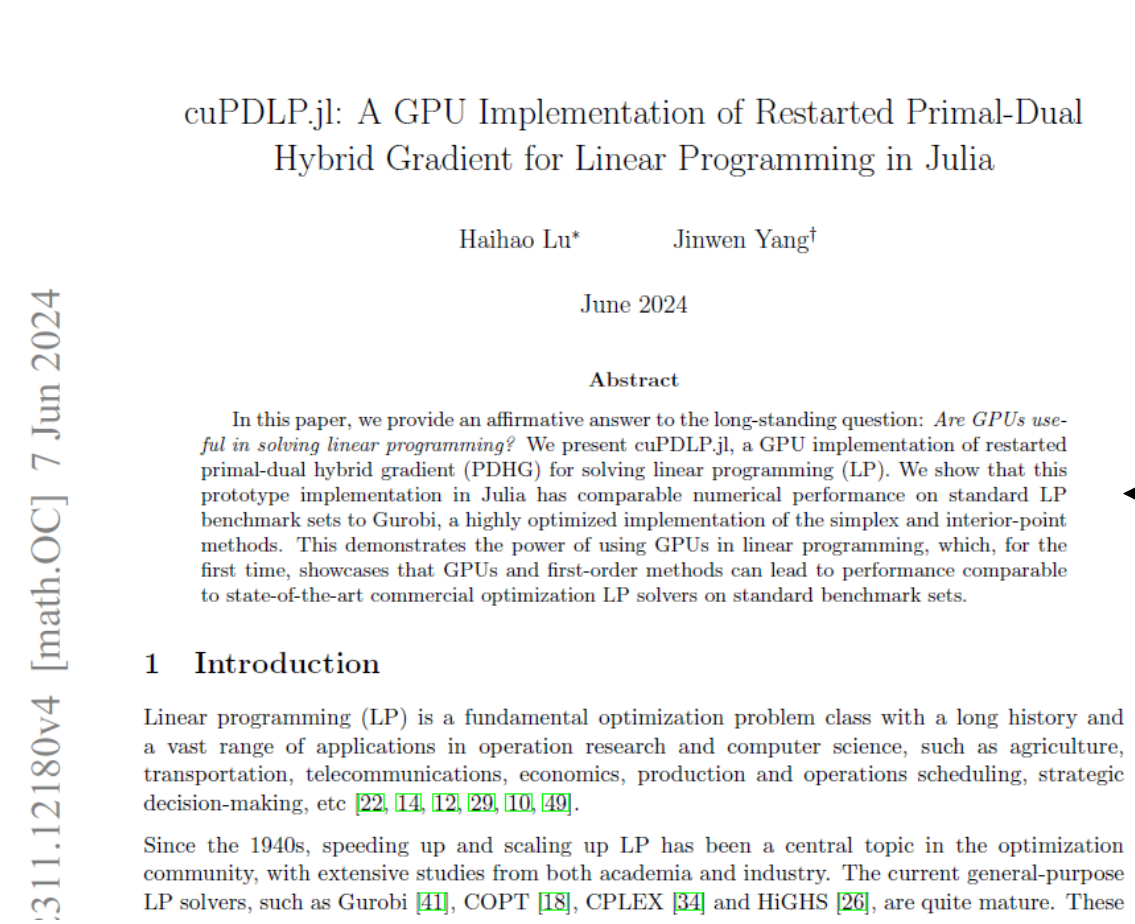
Brendan O'Donoghue
DeepMind
bodonoghue@deepmind.com

Warren Schudy
Google Research
wschudy@google.com

Abstract

We present PDLP, a practical first-order method for linear programming (LP) that can solve to the high levels of accuracy that are expected in traditional LP applications. In addition, it can scale to very large problems because its core operation is matrix-vector multiplications. PDLP is derived by applying the primal-dual hybrid gradient (PDHG) method, popularized by Chambolle and Pock (2011), to a saddle-point formulation of LP. PDLP enhances PDHG for LP by combining several new techniques with older tricks from the literature; the enhancements include diagonal preconditioning, presolving, adaptive step sizes, and adaptive restarting. PDLP improves the state of the art for first-order methods applied to LP. We compare PDLP with SCS, an ADMM-based solver, on a set of 383 LP instances derived from MIPLIB 2017. With a target of 10^{-8} relative accuracy and 1 hour time limit, PDLP achieves a 6.3x reduction in the geometric mean of solve times and a 4.6x reduction in the number of instances unsolved (from 227 to 49). Furthermore, we highlight standard benchmark instances and a large-scale application (PageRank) where our open-source prototype of PDLP, written in Julia, outperforms a commercial LP solver.

cuPDLP.jl



cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia

Haihao Lu* Jinwen Yang†

June 2024

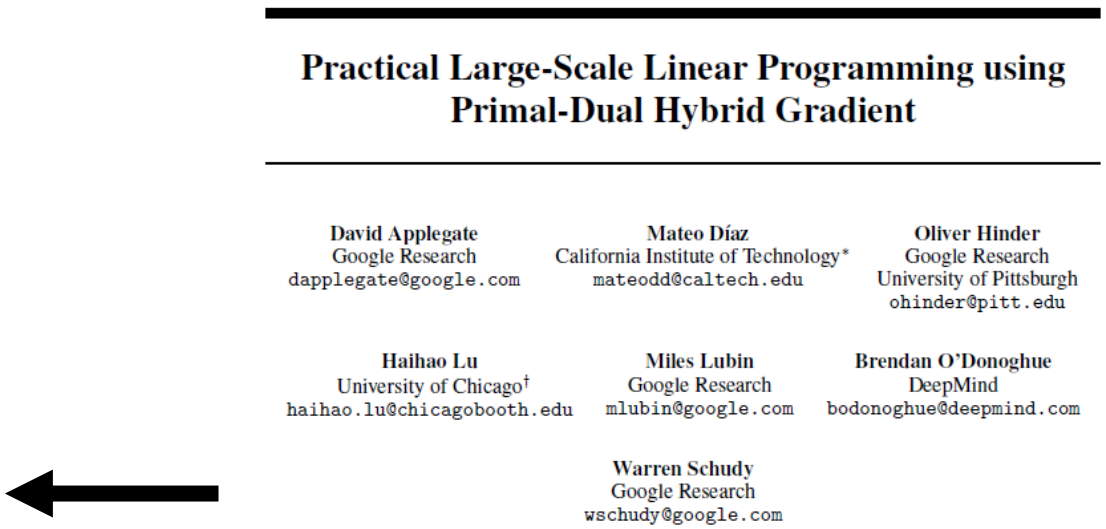
Abstract

In this paper, we provide an affirmative answer to the long-standing question: *Are GPUs useful in solving linear programming?* We present cuPDLP.jl, a GPU implementation of restarted primal-dual hybrid gradient (PDHG) for solving linear programming (LP). We show that this prototype implementation in Julia has comparable numerical performance on standard LP benchmark sets to Gurobi, a highly optimized implementation of the simplex and interior-point methods. This demonstrates the power of using GPUs in linear programming, which, for the first time, showcases that GPUs and first-order methods can lead to performance comparable to state-of-the-art commercial optimization LP solvers on standard benchmark sets.

1 Introduction

Linear programming (LP) is a fundamental optimization problem class with a long history and a vast range of applications in operation research and computer science, such as agriculture, transportation, telecommunications, economics, production and operations scheduling, strategic decision-making, etc [22, 14, 12, 29, 10, 49].

Since the 1940s, speeding up and scaling up LP has been a central topic in the optimization community, with extensive studies from both academia and industry. The current general-purpose LP solvers, such as Gurobi [41], COPT [18], CPLEX [34] and HiGHS [26], are quite mature. These



Practical Large-Scale Linear Programming using Primal-Dual Hybrid Gradient

David Applegate
Google Research
dapplegate@google.com

Mateo Díaz
California Institute of Technology*
mateodd@caltech.edu

Oliver Hinder
Google Research
University of Pittsburgh
ohinder@pitt.edu

Haihao Lu
University of Chicago†
haihao.lu@chicagobooth.edu

Miles Lubin
Google Research
mlubin@google.com

Brendan O'Donoghue
DeepMind
bodonoghue@deepmind.com

Warren Schudy
Google Research
wschudy@google.com

Abstract

We present PDLP, a practical first-order method for linear programming (LP) that can solve to the high levels of accuracy that are expected in traditional LP applications. In addition, it can scale to very large problems because its core operation is matrix-vector multiplications. PDLP is derived by applying the primal-dual hybrid gradient (PDHG) method, popularized by Chambolle and Pock (2011), to a saddle-point formulation of LP. PDLP enhances PDHG for LP by combining several new techniques with older tricks from the literature; the enhancements include diagonal preconditioning, presolving, adaptive step sizes, and adaptive restarting. PDLP improves the state of the art for first-order methods applied to LP. We compare PDLP with SCS, an ADMM-based solver, on a set of 383 LP instances derived from MIPLIB 2017. With a target of 10^{-8} relative accuracy and 1 hour time limit, PDLP achieves a 6.3x reduction in the geometric mean of solve times and a 4.6x reduction in the number of instances unsolved (from 227 to 49). Furthermore, we highlight standard benchmark instances and a large-scale application (PageRank) where our open-source prototype of PDLP, written in Julia, outperforms a commercial LP solver.

16. Lu, H. and Yang, J. **cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia.** *arXiv:2311.12180v4* (2024).

cuPDLP.jl

- Entire GPU used for **each step** (maps threads to variables)
  - Useful for **large problems**

Algorithm 1: Restarted PDHG (after preconditioning)

Input: Initial point $z^{0,0}$;

Initialize outer loop counter $n \leftarrow 0$, total iterations $k \leftarrow 0$, step-size $\hat{\eta}^{0,0} \leftarrow 1/\|K\|_\infty$, primal weight $\omega^0 \leftarrow \text{InitializePrimalWeight}(c, q)$;

repeat

  $t \leftarrow 0$;

  repeat

    $z^{n,t+1}, \eta^{n,t+1}, \hat{\eta}^{n,t+1} \leftarrow \text{AdaptiveStepPDHG}(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$;

    $\bar{z}^{n,t+1} \leftarrow \frac{1}{\sum_{i=1}^{t+1} \eta^{n,i}} \sum_{i=1}^{k+1} \eta^{n,i} z^{n,i}$;

    $z_c^{n,t+1} \leftarrow \text{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1})$;

    $t \leftarrow t+1$, $k \leftarrow k+1$;

  until restart or termination criteria holds;

  restart the outer loop $z^{n+1,0} \leftarrow z_c^{n,t}$, $n \leftarrow n+1$;

  $\omega^n \leftarrow \text{PrimalWeightUpdate}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$;

until termination criteria holds;

Output: $z^{n,0}$.

16. Lu, H. and Yang, J. **cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia.** *arXiv:2311.12180v4* (2024).

cuPDLP.jl

- Entire GPU used for **each step** (maps threads to variables)
  - Useful for **large problems**

Algorithm 1: Restarted PDHG (after preconditioning)

Input: Initial point $z^{0,0}$;

Initialize outer loop counter $n \leftarrow 0$, total iterations $k \leftarrow 0$, step-size $\hat{\eta}^{0,0} \leftarrow 1/\|K\|_\infty$, primal weight $\omega^0 \leftarrow \text{InitializePrimalWeight}(c, q)$;

repeat

  $t \leftarrow 0$;

  repeat

    $z^{n,t+1}, \eta^{n,t+1}, \hat{\eta}^{n,t+1} \leftarrow \text{AdaptiveStepPDHG}(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$;

    $\bar{z}^{n,t+1} \leftarrow \frac{1}{\sum_{i=1}^{t+1} \eta^{n,i}} \sum_{i=1}^{k+1} \eta^{n,i} z^{n,i}$;

    $z_c^{n,t+1} \leftarrow \text{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1})$;

    $t \leftarrow t+1$, $k \leftarrow k+1$;

  until restart or termination criteria holds;

  restart the outer loop $z^{n+1,0} \leftarrow z_c^{n,t}$, $n \leftarrow n+1$;

  $\omega^n \leftarrow \text{PrimalWeightUpdate}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$;

until termination criteria holds;

Output: $z^{n,0}$.

16. Lu, H. and Yang, J. **cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia.** *arXiv:2311.12180v4* (2024).

cuPDLP.jl

- Entire GPU used for **each step** (maps threads to variables)
  - Useful for **large problems**

Algorithm 1: Restarted PDHG (after preconditioning)

Input: Initial point $z^{0,0}$;

Initialize outer loop counter $n \leftarrow 0$, total iterations $k \leftarrow 0$, step-size $\hat{\eta}^{0,0} \leftarrow 1/\|K\|_\infty$, primal weight $\omega^0 \leftarrow \text{InitializePrimalWeight}(c, q)$;

repeat

  $t \leftarrow 0$;

  repeat

    $z^{n,t+1}, \eta^{n,t+1}, \hat{\eta}^{n,t+1} \leftarrow \text{AdaptiveStepPDHG}(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$;

    $\bar{z}^{n,t+1} \leftarrow \frac{1}{\sum_{i=1}^{t+1} \eta^{n,i}} \sum_{i=1}^{k+1} \eta^{n,i} z^{n,i}$;

    $z_c^{n,t+1} \leftarrow \text{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1})$;

    $t \leftarrow t+1$, $k \leftarrow k+1$;

  until restart or termination criteria holds;

  restart the outer loop $z^{n+1,0} \leftarrow z_c^{n,t}$, $n \leftarrow n+1$;

  $\omega^n \leftarrow \text{PrimalWeightUpdate}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$;

until termination criteria holds;

Output: $z^{n,0}$.

16. Lu, H. and Yang, J. **cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia.** *arXiv:2311.12180v4* (2024).

cuPDLP.jl

- Entire GPU used for **each step** (maps threads to variables)
  - Useful for **large problems**

Algorithm 1: Restarted PDHG (after preconditioning)

Input: Initial point $z^{0,0}$;

Initialize outer loop counter $n \leftarrow 0$, total iterations $k \leftarrow 0$, step-size $\hat{\eta}^{0,0} \leftarrow 1/\|K\|_\infty$, primal weight $\omega^0 \leftarrow \text{InitializePrimalWeight}(c, q)$;

repeat

  $t \leftarrow 0$;

  repeat

    $z^{n,t+1}, \eta^{n,t+1}, \hat{\eta}^{n,t+1} \leftarrow \text{AdaptiveStepPDHG}(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$;

    $\bar{z}^{n,t+1} \leftarrow \frac{1}{\sum_{i=1}^{t+1} \eta^{n,i}} \sum_{i=1}^{k+1} \eta^{n,i} z^{n,i}$;

    $z_c^{n,t+1} \leftarrow \text{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1})$;

    $t \leftarrow t+1$, $k \leftarrow k+1$;

  until restart or termination criteria holds;

  restart the outer loop $z^{n+1,0} \leftarrow z_c^{n,t}$, $n \leftarrow n+1$;

  $\omega^n \leftarrow \text{PrimalWeightUpdate}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$;

until termination criteria holds;

Output: $z^{n,0}$.

16. Lu, H. and Yang, J. **cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia.** *arXiv:2311.12180v4* (2024).

cuPDLP.jl

- Entire GPU used for **each step** (maps threads to variables)
  - Useful for **large problems**

Algorithm 1: Restarted PDHG (after preconditioning)

Input: Initial point $z^{0,0}$;

Initialize outer loop counter $n \leftarrow 0$, total iterations $k \leftarrow 0$, step-size $\hat{\eta}^{0,0} \leftarrow 1/\|K\|_\infty$, primal weight $\omega^0 \leftarrow \text{InitializePrimalWeight}(c, q)$;

repeat

  $t \leftarrow 0$;

  repeat

    $z^{n,t+1}, \eta^{n,t+1}, \hat{\eta}^{n,t+1} \leftarrow \text{AdaptiveStepPDHG}(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$;

    $\bar{z}^{n,t+1} \leftarrow \frac{1}{\sum_{i=1}^{t+1} \eta^{n,i}} \sum_{i=1}^{k+1} \eta^{n,i} z^{n,i}$;

    $z_c^{n,t+1} \leftarrow \text{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1})$;

    $t \leftarrow t+1$, $k \leftarrow k+1$;

  until restart or termination criteria holds;

  restart the outer loop $z^{n+1,0} \leftarrow z_c^{n,t}$, $n \leftarrow n+1$;

  $\omega^n \leftarrow \text{PrimalWeightUpdate}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$;

until termination criteria holds;

Output: $z^{n,0}$.

16. Lu, H. and Yang, J. **cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia.** *arXiv:2311.12180v4* (2024).

cuPDLP.jl

- Entire GPU used for **each step** (maps threads to variables)
  - Useful for **large problems**

Algorithm 1: Restarted PDHG (after preconditioning)

Input: Initial point $z^{0,0}$;

Initialize outer loop counter $n \leftarrow 0$, total iterations $k \leftarrow 0$, step-size $\hat{\eta}^{0,0} \leftarrow 1/\|K\|_\infty$, primal weight $\omega^0 \leftarrow \text{InitializePrimalWeight}(c, q)$;

repeat

  $t \leftarrow 0$;

  repeat

    $z^{n,t+1}, \eta^{n,t+1}, \hat{\eta}^{n,t+1} \leftarrow \text{AdaptiveStepPDHG}(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$;

    $\bar{z}^{n,t+1} \leftarrow \frac{1}{\sum_{i=1}^{t+1} \eta^{n,i}} \sum_{i=1}^{k+1} \eta^{n,i} z^{n,i}$;

    $z_c^{n,t+1} \leftarrow \text{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1})$;

    $t \leftarrow t+1$, $k \leftarrow k+1$;

  until restart or termination criteria holds;

  restart the outer loop $z^{n+1,0} \leftarrow z_c^{n,t}$, $n \leftarrow n+1$;

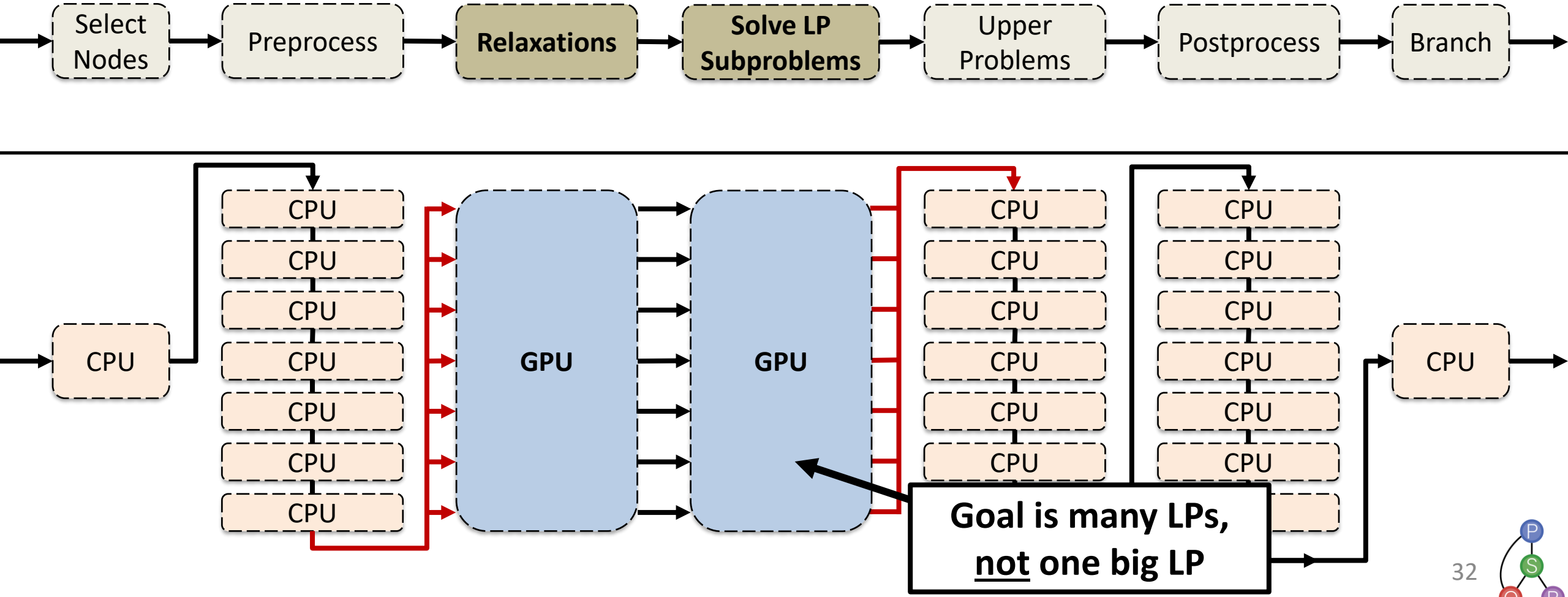
  $\omega^n \leftarrow \text{PrimalWeightUpdate}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$;

until termination criteria holds;

Output: $z^{n,0}$.

16. Lu, H. and Yang, J. **cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia.** *arXiv:2311.12180v4* (2024).

Hybrid CPU-GPU B&B



BatchPDLP

- Custom novel implementation of PDLP, written as a single CUDA kernel
- **Each LP** solved by a **portion** of the GPU

Algorithm 1: Single-Kernel PDLP

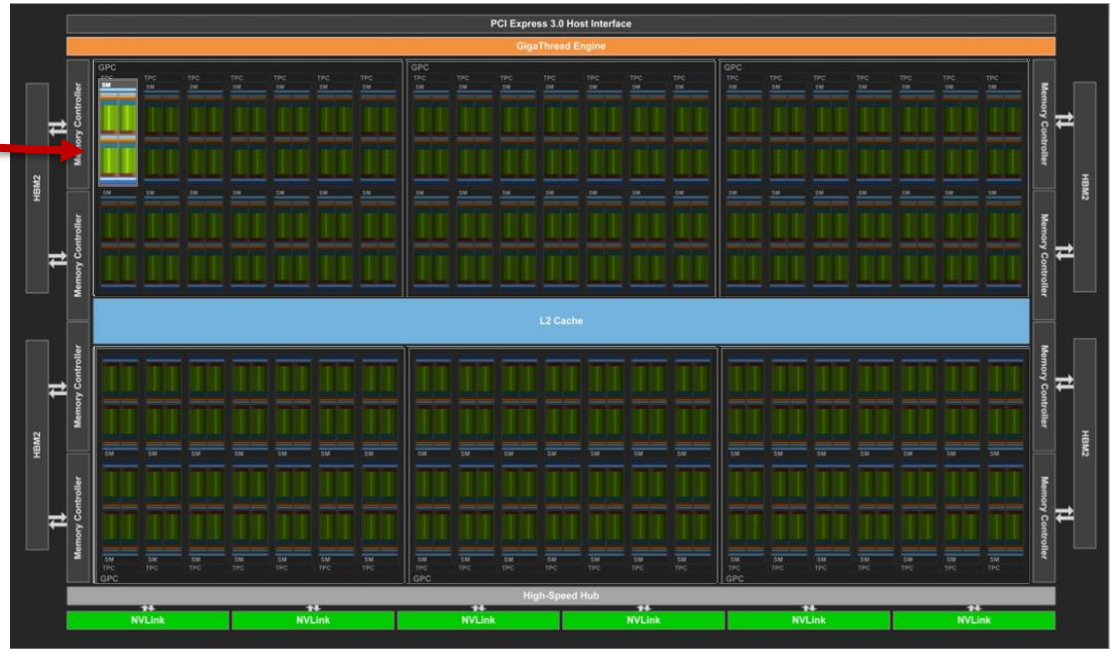
foreach *LP* **do**
  Input: An initial solution $z_i^{0,0}$;
  Initialize outer loop counter $n_i \leftarrow 0$, total iterations $k_i \leftarrow 0$, step size $\hat{\eta}_i^{0,0} \leftarrow 1/\|K_i\|_\infty$, primal weight $\omega_i^0 \leftarrow \texttt{InitializePrimalWeight}(c_i, q_i)$;
  repeat
    $t_i \leftarrow 0$;
    repeat
      $z_i^{n_i,t_i+1}, \eta_i^{n_i,t_i+1}, \hat{\eta}_i^{n_i,t_i+1} \leftarrow \texttt{AdaptivePDHGStep}(z_i^{n_i,t_i}, \omega_i^{n_i}, \hat{\eta}_i^{n_i,t_i}, k_i)$;
      $\bar{z}_i^{n_i,t_i+1} \leftarrow \frac{1}{\sum_{j=1}^{t_i+1} \eta_i^{n_i,j}} \sum_{j=1}^{t_i+1} \eta_i^{n_i,j} z_i^{n_i,j}$;
      $z_{c,i}^{n_i,t_i+1} \leftarrow \texttt{GetRestartCandidate}(z_i^{n_i,t_i+1}, \bar{z}_i^{n_i,t_i+1}, \omega_i^{n_i})$;
      $t_i \leftarrow t_i + 1, k_i \leftarrow k_i + 1$;
    until *restart or termination criteria holds*;
    Restart the outer loop. $z_i^{n_i+1,0} \leftarrow z_{c,i}^{n_i,t_i}, n_i \leftarrow n_i + 1$;
    $\omega_i^{n_i} \leftarrow \texttt{PrimalWeightUpdate}(z_i^{n_i,0}, z_i^{n_i-1,0}, \omega_i^{n_i-1})$;
  until *termination criteria holds*;
  Output: $z_i^{n_i,0}$.
end

BatchPDLP

- Custom novel implementation of PDLP, written as a single CUDA kernel
- **Each LP** solved by a **portion** of the GPU

Algorithm 1: Single-Kernel PDLP

foreach LP **do**
  Input: An initial solution $z_i^{0,0}$;
  Initialize outer loop counter $n_i \leftarrow 0$, total iterations $k_i \leftarrow 0$, step size $\hat{\eta}_i^{0,0} \leftarrow 1/\|K_i\|_\infty$, primal weight $\omega_i^0 \leftarrow \texttt{InitializePrimalWeight}(c_i, q_i)$;
  repeat
    $t_i \leftarrow 0$;
    repeat
      $z_i^{n_i,t_i+1}, \eta_i^{n_i,t_i+1}, \hat{\eta}_i^{n_i,t_i+1} \leftarrow \texttt{AdaptivePDHGStep}(z_i^{n_i,t_i}, \omega_i^{n_i}, \hat{\eta}_i^{n_i,t_i}, k_i)$;
      $\bar{z}_i^{n_i,t_i+1} \leftarrow \frac{1}{\sum_{j=1}^{t_i+1} \eta_i^{n_i,j}} \sum_{j=1}^{t_i+1} \eta_i^{n_i,j} z_i^{n_i,j}$;
      $z_{c,i}^{n_i,t_i+1} \leftarrow \texttt{GetRestartCandidate}(z_i^{n_i,t_i+1}, \bar{z}_i^{n_i,t_i+1}, \omega_i^{n_i})$;
      $t_i \leftarrow t_i + 1, k_i \leftarrow k_i + 1$;
    until *restart or termination criteria holds*;
    Restart the outer loop. $z_i^{n_i+1,0} \leftarrow z_{c,i}^{n_i,t_i}, n_i \leftarrow n_i + 1$;
    $\omega_i^{n_i} \leftarrow \texttt{PrimalWeightUpdate}(z_i^{n_i,0}, z_i^{n_i-1,0}, \omega_i^{n_i-1})$;
  until *termination criteria holds*;
  Output: $z_i^{n_i,0}$.
end



BatchPDLP

- Custom novel implementation of PDLP, written as a single CUDA kernel
- **Each LP** solved by a **portion** of the GPU

Algorithm 1: Single-Kernel PDLP

foreach *LP* **do**
  Input: An initial solution $z_i^{0,0}$;
  Initialize outer loop counter $n_i \leftarrow 0$, total iterations $k_i \leftarrow 0$, step size $\hat{\eta}_i^{0,0} \leftarrow 1/\|K_i\|_\infty$, primal weight $\omega_i^0 \leftarrow \texttt{InitializePrimalWeight}(c_i, q_i)$;
  repeat
    $t_i \leftarrow 0$;
    repeat
      $z_i^{n_i,t_i+1}, \eta_i^{n_i,t_i+1}, \hat{\eta}_i^{n_i,t_i+1} \leftarrow \texttt{AdaptivePDHGStep}(z_i^{n_i,t_i}, \omega_i^{n_i}, \hat{\eta}_i^{n_i,t_i}, k_i)$;
      $\bar{z}_i^{n_i,t_i+1} \leftarrow \frac{1}{\sum_{j=1}^{t_i+1} \eta_i^{n_i,j}} \sum_{j=1}^{t_i+1} \eta_i^{n_i,j} z_i^{n_i,j}$;
      $z_{c,i}^{n_i,t_i+1} \leftarrow \texttt{GetRestartCandidate}(z_i^{n_i,t_i+1}, \bar{z}_i^{n_i,t_i+1}, \omega_i^{n_i})$;
      $t_i \leftarrow t_i + 1, k_i \leftarrow k_i + 1$;
    until *restart or termination criteria holds*;
    Restart the outer loop. $z_i^{n_i+1,0} \leftarrow z_{c,i}^{n_i,t_i}, n_i \leftarrow n_i + 1$;
    $\omega_i^{n_i} \leftarrow \texttt{PrimalWeightUpdate}(z_i^{n_i,0}, z_i^{n_i-1,0}, \omega_i^{n_i-1})$;
  until *termination criteria holds*;
  Output: $z_i^{n_i,0}$.
end

BatchPDLP

- Custom novel implementation of PDLP, written as a single CUDA kernel
- **Each LP** solved by a **portion** of the GPU

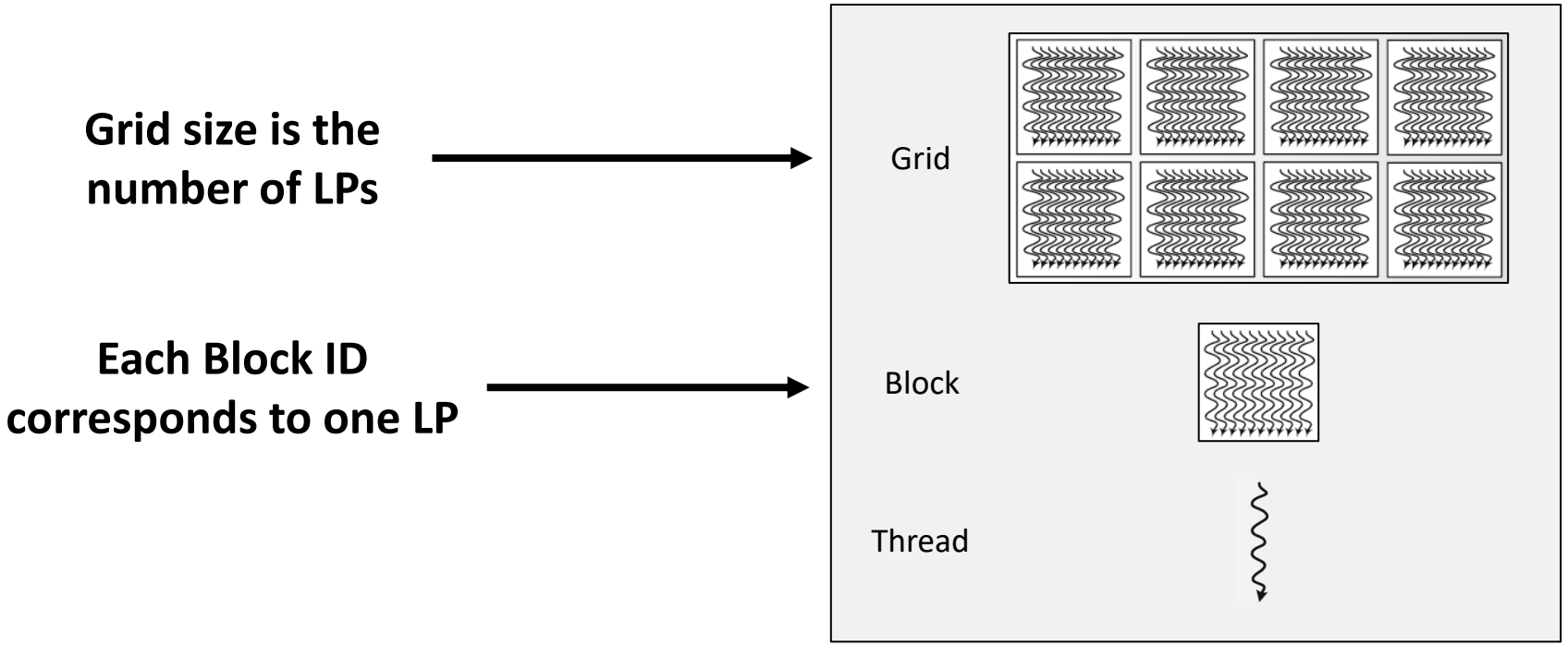
Algorithm 1: Single-Kernel PDLP

1 **foreach** *LP* **do**
2 **Input:** An initial solution $z_i^{0,0}$;
3 Initialize outer loop counter $n_i \leftarrow 0$, total iterations $k_i \leftarrow 0$, step size $\hat{\eta}_i^{0,0} \leftarrow 1/\|K_i\|_\infty$, primal weight $\omega_i^0 \leftarrow \texttt{InitializePrimalWeight}(c_i, q_i)$;
4 **repeat**
5 $t_i \leftarrow 0$;
6 **repeat**
7 $z_i^{n_i,t_i+1}, \eta_i^{n_i,t_i+1}, \hat{\eta}_i^{n_i,t_i+1} \leftarrow \texttt{AdaptivePDHGStep}(z_i^{n_i,t_i}, \omega_i^{n_i}, \hat{\eta}_i^{n_i,t_i}, k_i)$;
8 $\bar{z}_i^{n_i,t_i+1} \leftarrow \frac{1}{\sum_{j=1}^{t_i+1} \eta_i^{n_i,j}} \sum_{j=1}^{t_i+1} \eta_i^{n_i,j} z_i^{n_i,j}$;
9 $z_{c,i}^{n_i,t_i+1} \leftarrow \texttt{GetRestartCandidate}(z_i^{n_i,t_i+1}, \bar{z}_i^{n_i,t_i+1}, \omega_i^{n_i})$;
10 $t_i \leftarrow t_i + 1, k_i \leftarrow k_i + 1$;
11 **until** *restart or termination criteria holds*;
12 **Restart the outer loop.** $z_i^{n_i+1,0} \leftarrow z_{c,i}^{n_i,t_i}, n_i \leftarrow n_i + 1$;
13 $\omega_i^{n_i} \leftarrow \texttt{PrimalWeightUpdate}(z_i^{n_i,0}, z_i^{n_i-1,0}, \omega_i^{n_i-1})$;
14 **until** *termination criteria holds*;
15 **Output:** $z_i^{n_i,0}$.
16 **end**

Other LPs continue even if one finishes

High-Level Organization

- Built for **smaller LPs** (*within each block*, maps threads to variables)



17. Lindholm, E., et al. **NVIDIA Tesla: A Unified Graphics and Computing Architecture**. *IEEE Micro*, 28(2): 39–55 (2008).

Memory Organization

Global Memory

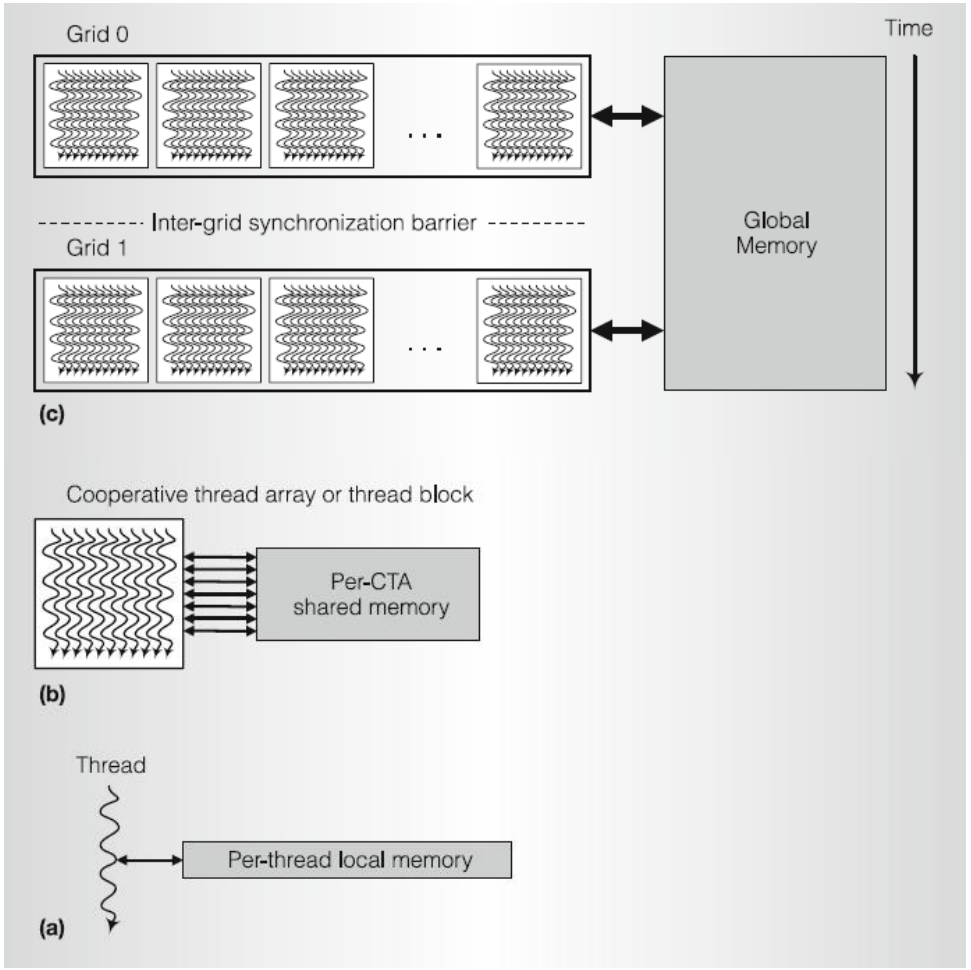
- **Problem information** (bounds, constraint matrix, right-hand side, […])
- **Problem state** (current/average/sum of primal/dual solutions, […])

Block Memory

- **Re-usable storage for parallel reductions** (sum, max)
- **Persistent data/flags** (primal weight, step size, […])

Thread Memory

- **(Thread 1) Parallel reduction results**
- **Critical information** (iteration number, […])



17. Lindholm, E., et al. **NVIDIA Tesla: A Unified Graphics and Computing Architecture**. *IEEE Micro*, 28(2): 39–55 (2008).

Memory Organization

Global Memory

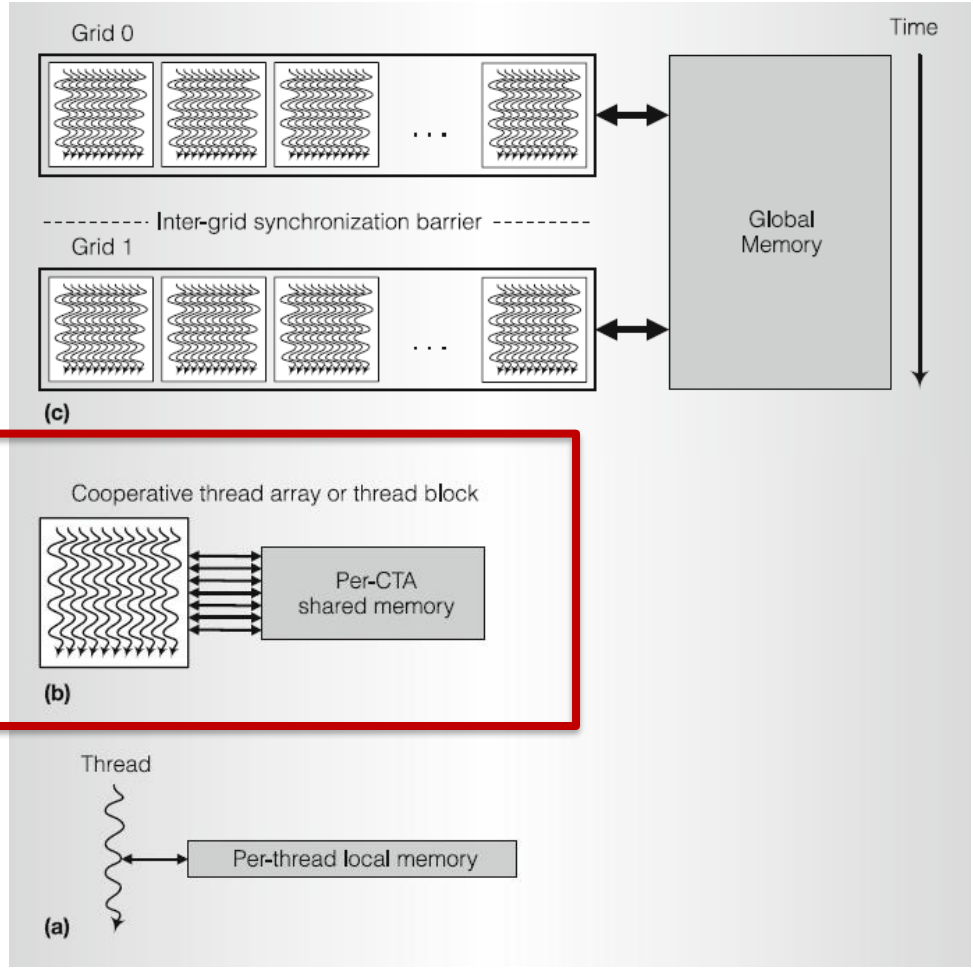
- **Problem information** (bounds, constraint matrix, right-hand side, […])
- **Problem state** (current/average/sum of primal/dual solutions, […])

Block Memory

- **Re-usable storage for parallel reductions** (sum, max)
- **Persistent data/flags** (primal weight, step size, […])

Thread Memory

- **(Thread 1) Parallel reduction results**
- **Critical information** (iteration number, […])

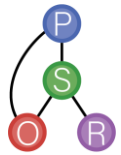


17. Lindholm, E., et al. **NVIDIA Tesla: A Unified Graphics and Computing Architecture**. *IEEE Micro*, 28(2): 39–55 (2008).

Exploiting Sparsity

Original NLP

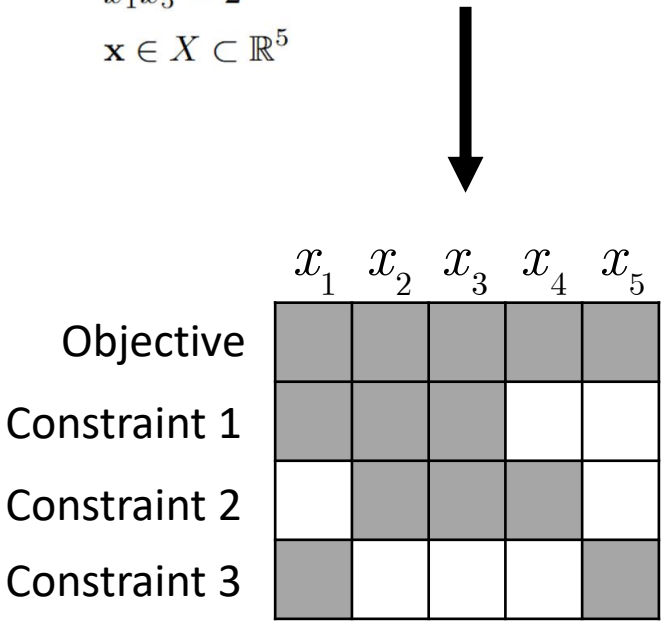
$$
\begin{aligned}
f^* = \min_{\mathbf{x}} \; & (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4 \\
\text{s.t.} \quad & x_2^2 + x_3^3 + x_1 = 6.24264068711929 \\
& -x_3^2 + x_2 + x_4 = 0.82842712474619 \\
& x_1 x_5 = 2 \\
& \mathbf{x} \in X \subset \mathbb{R}^5
\end{aligned}
$$



Exploiting Sparsity

Original NLP

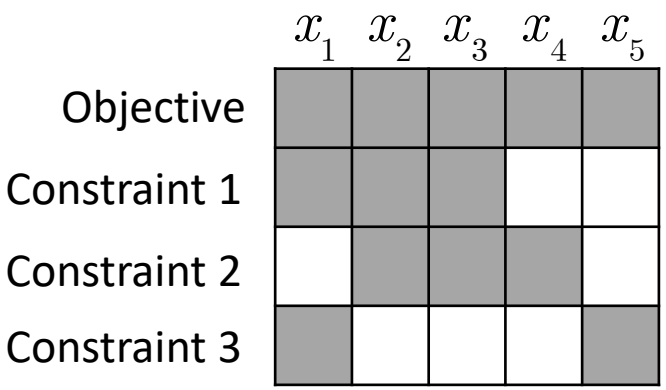
$$
\begin{aligned}
f^* = \min_{\mathbf{x}} \; & (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4 \\
\text{s.t.} \quad & x_2^2 + x_3^3 + x_1 = 6.24264068711929 \\
& -x_3^2 + x_2 + x_4 = 0.82842712474619 \\
& x_1 x_5 = 2 \\
& \mathbf{x} \in X \subset \mathbb{R}^5
\end{aligned}
$$



Exploiting Sparsity

Original NLP

$$
\begin{aligned}
f^* = \min_{\mathbf{x}} \; & (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4 \\
\text{s.t.} \quad & x_2^2 + x_3^3 + x_1 = 6.24264068711929 \\
& -x_3^2 + x_2 + x_4 = 0.82842712474619 \\
& x_1 x_5 = 2 \\
& \mathbf{x} \in X \subset \mathbb{R}^5
\end{aligned}
$$



LP from Relaxation Subgradients

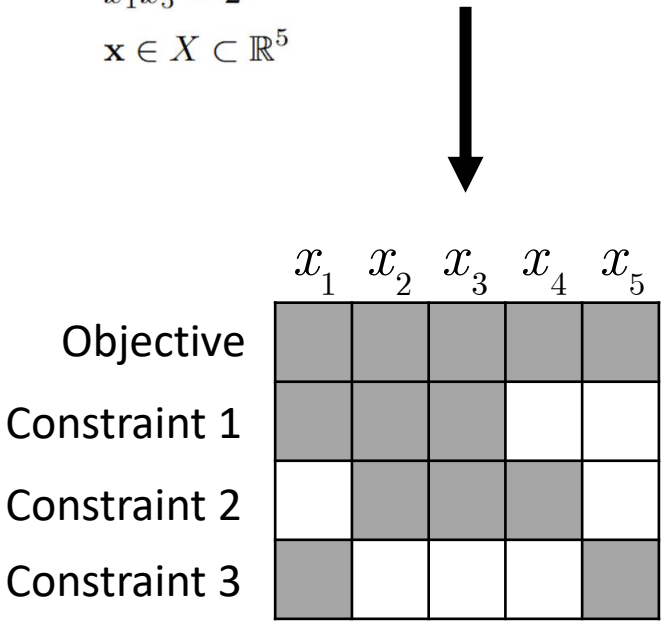
$$
\begin{aligned}
f^*_{\text{aff}} = \min_{\mathbf{x}} \; & \eta \\
\text{s.t.} \quad & \eta + \text{obj}_{\text{cv}}(x_1, x_2, x_3, x_4, x_5) \geq 0 \\
& \text{eq\_1}^{\text{aff}}_{\text{cv}}(x_1, x_2, x_3) \geq 0 \\
& \text{eq\_1}^{\text{aff}}_{\text{cc}}(x_1, x_2, x_3) \geq 0 \\
& \text{eq\_2}^{\text{aff}}_{\text{cv}}(x_2, x_3, x_4) \geq 0 \\
& \text{eq\_2}^{\text{aff}}_{\text{cc}}(x_2, x_3, x_4) \geq 0 \\
& \text{eq\_3}^{\text{aff}}_{\text{cv}}(x_1, x_5) \geq 0 \\
& \text{eq\_3}^{\text{aff}}_{\text{cc}}(x_1, x_5) \geq 0 \\
& \mathbf{x} \in X \subset \mathbb{R}^5
\end{aligned}
$$

Exploiting Sparsity

Original NLP

$$f^* = \min_{\mathbf{x}} (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4$$

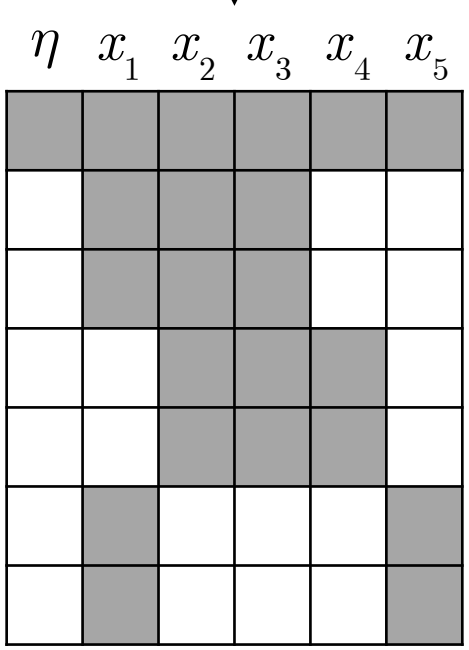
$$\text{s.t.} \quad x_2^2 + x_3^3 + x_1 = 6.24264068711929$$
$$-x_3^2 + x_2 + x_4 = 0.82842712474619$$
$$x_1 x_5 = 2$$
$$\mathbf{x} \in X \subset \mathbb{R}^5$$



LP from Relaxation Subgradients

$$f^*_{\text{aff}} = \min_{\mathbf{x}} \eta$$

$$\text{s.t.} \quad \eta + \text{obj}_{\text{cv}}(x_1, x_2, x_3, x_4, x_5) \geq 0$$
$$\text{eq\_1}^{\text{aff}}_{\text{cv}}(x_1, x_2, x_3) \geq 0$$
$$\text{eq\_1}^{\text{aff}}_{\text{cc}}(x_1, x_2, x_3) \geq 0$$
$$\text{eq\_2}^{\text{aff}}_{\text{cv}}(x_2, x_3, x_4) \geq 0$$
$$\text{eq\_2}^{\text{aff}}_{\text{cc}}(x_2, x_3, x_4) \geq 0$$
$$\text{eq\_3}^{\text{aff}}_{\text{cv}}(x_1, x_5) \geq 0$$
$$\text{eq\_3}^{\text{aff}}_{\text{cc}}(x_1, x_5) \geq 0$$
$$\mathbf{x} \in X \subset \mathbb{R}^5$$



Exploiting Sparsity

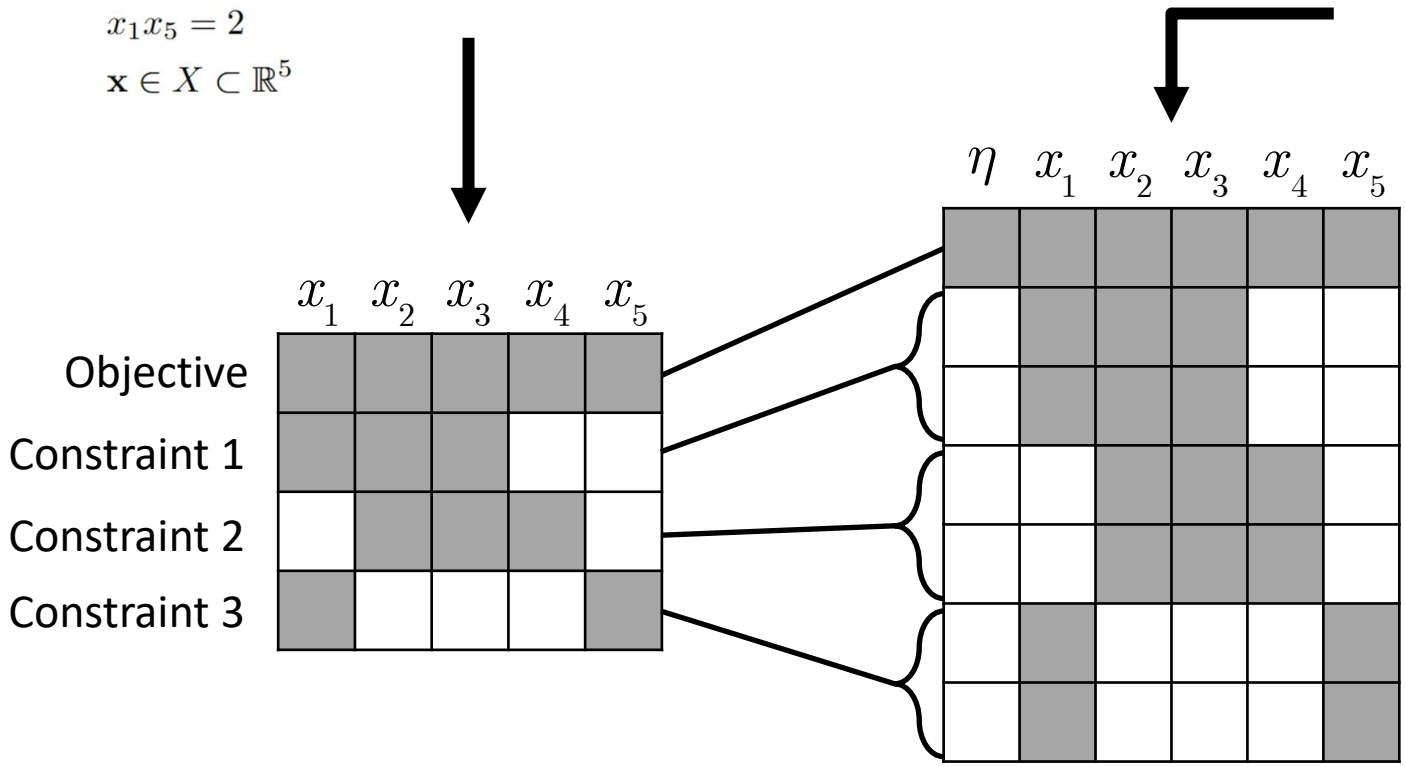
Original NLP

$$f^* = \min_{\mathbf{x}} (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4$$

$$\text{s.t.} \quad x_2^2 + x_3^3 + x_1 = 6.24264068711929$$
$$-x_3^2 + x_2 + x_4 = 0.82842712474619$$
$$x_1 x_5 = 2$$
$$\mathbf{x} \in X \subset \mathbb{R}^5$$

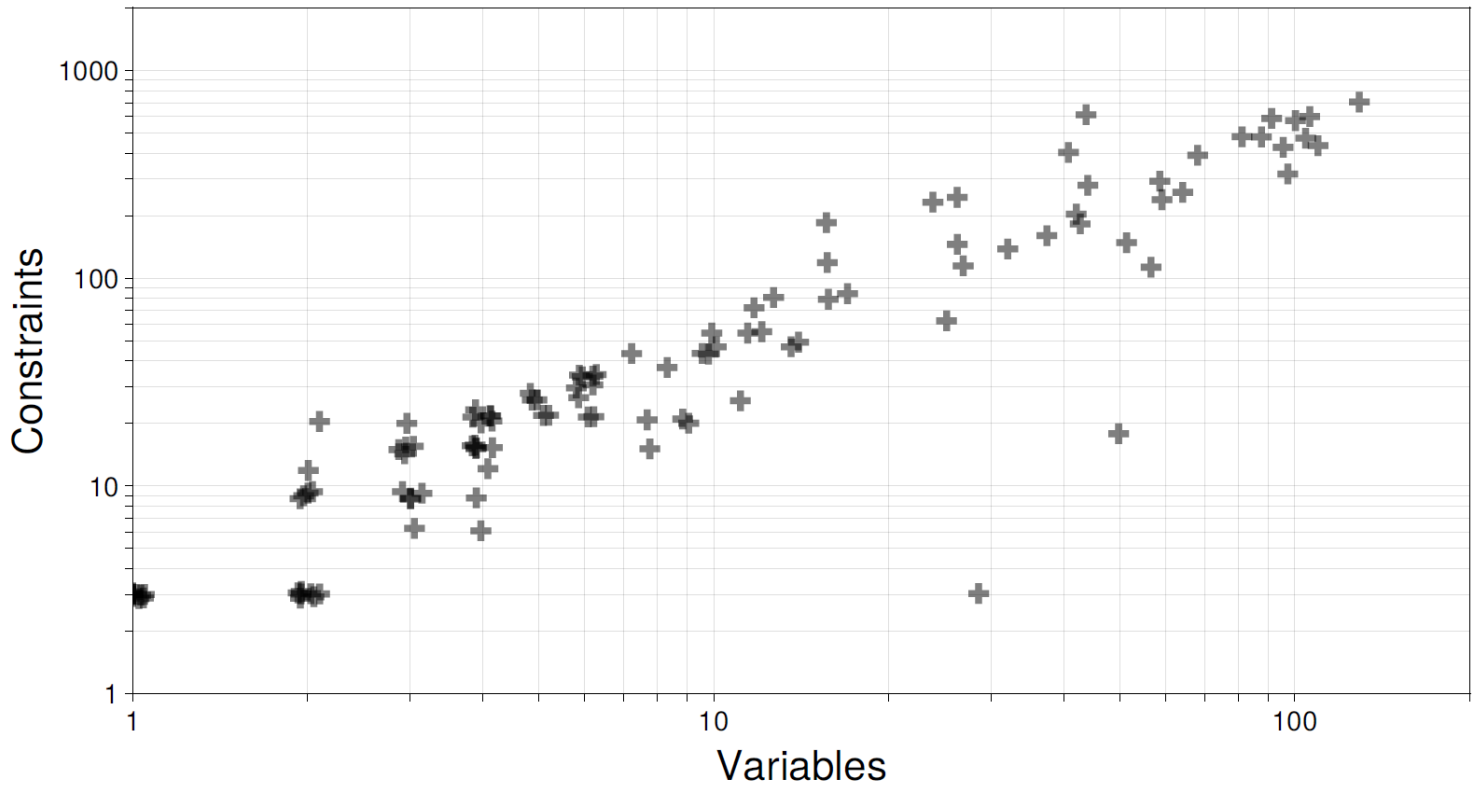
LP from Relaxation Subgradients

$$f^*_{\text{aff}} = \min_{\mathbf{x}} \eta$$
$$\text{s.t.} \quad \eta + \text{obj}_{\text{cv}}(x_1, x_2, x_3, x_4, x_5) \geq 0$$
$$\text{eq\_1}^{\text{aff}}_{\text{cv}}(x_1, x_2, x_3) \geq 0$$
$$\text{eq\_1}^{\text{aff}}_{\text{cc}}(x_1, x_2, x_3) \geq 0$$
$$\text{eq\_2}^{\text{aff}}_{\text{cv}}(x_2, x_3, x_4) \geq 0$$
$$\text{eq\_2}^{\text{aff}}_{\text{cc}}(x_2, x_3, x_4) \geq 0$$
$$\text{eq\_3}^{\text{aff}}_{\text{cv}}(x_1, x_5) \geq 0$$
$$\text{eq\_3}^{\text{aff}}_{\text{cc}}(x_1, x_5) \geq 0$$
$$\mathbf{x} \in X \subset \mathbb{R}^5$$

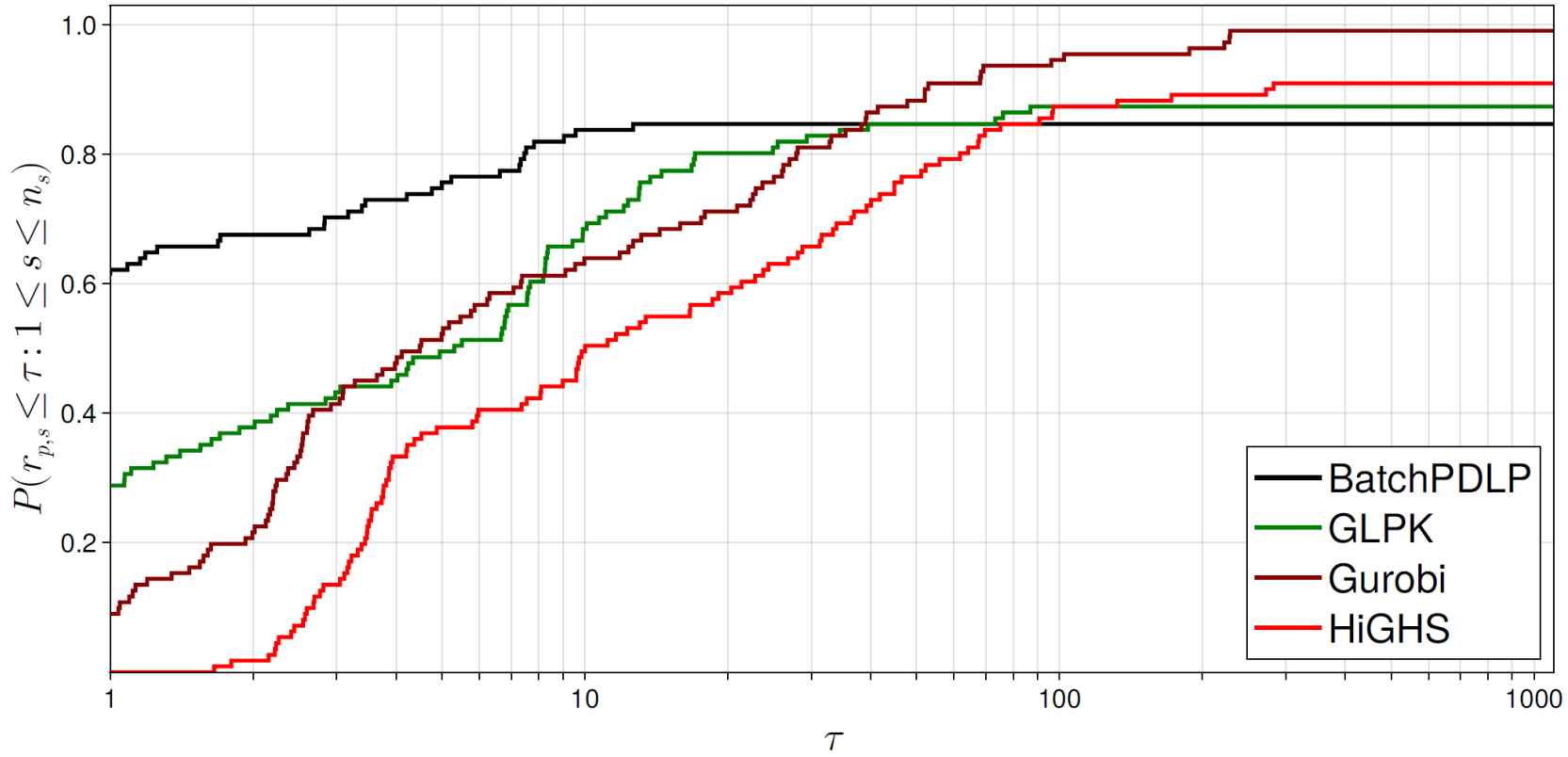


Benchmark Tests

- Subset of “small” NLPs from MINLPLib
- Original domain partitioned into 10000 subdomains
- Evaluated subgradients at 3 points per subdomain to generate LP constraints



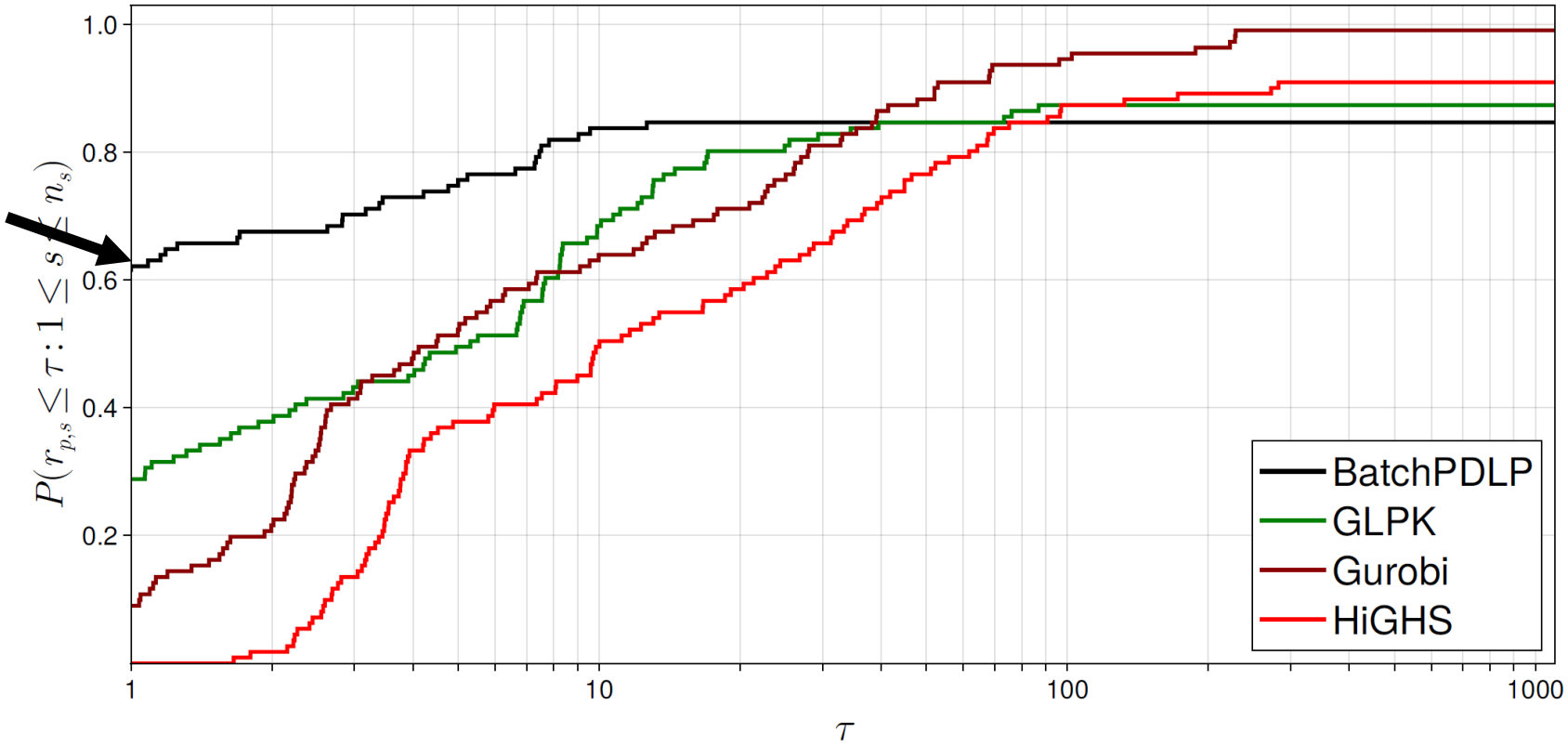
Performance Profile



18. Dolan, E and Moré, J. **Benchmarking optimization software with performance profiles.** *Math. Program. Ser. A*, 91: 201–213 (2002).

Performance Profile

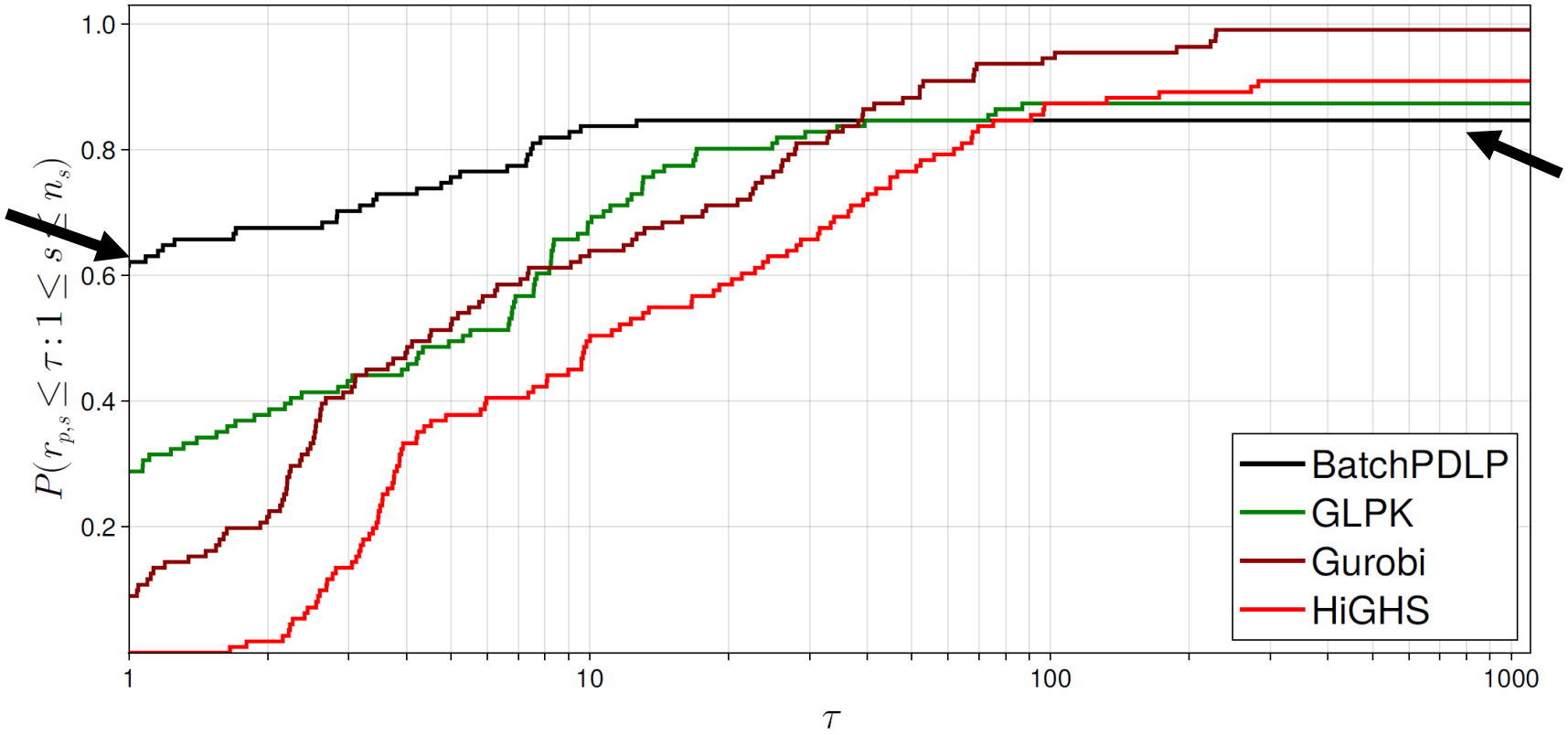
Fastest Solver in
61% of Problems



18. Dolan, E and Moré, J. **Benchmarking optimization software with performance profiles.** *Math. Program. Ser. A*, 91: 201–213 (2002).

Performance Profile

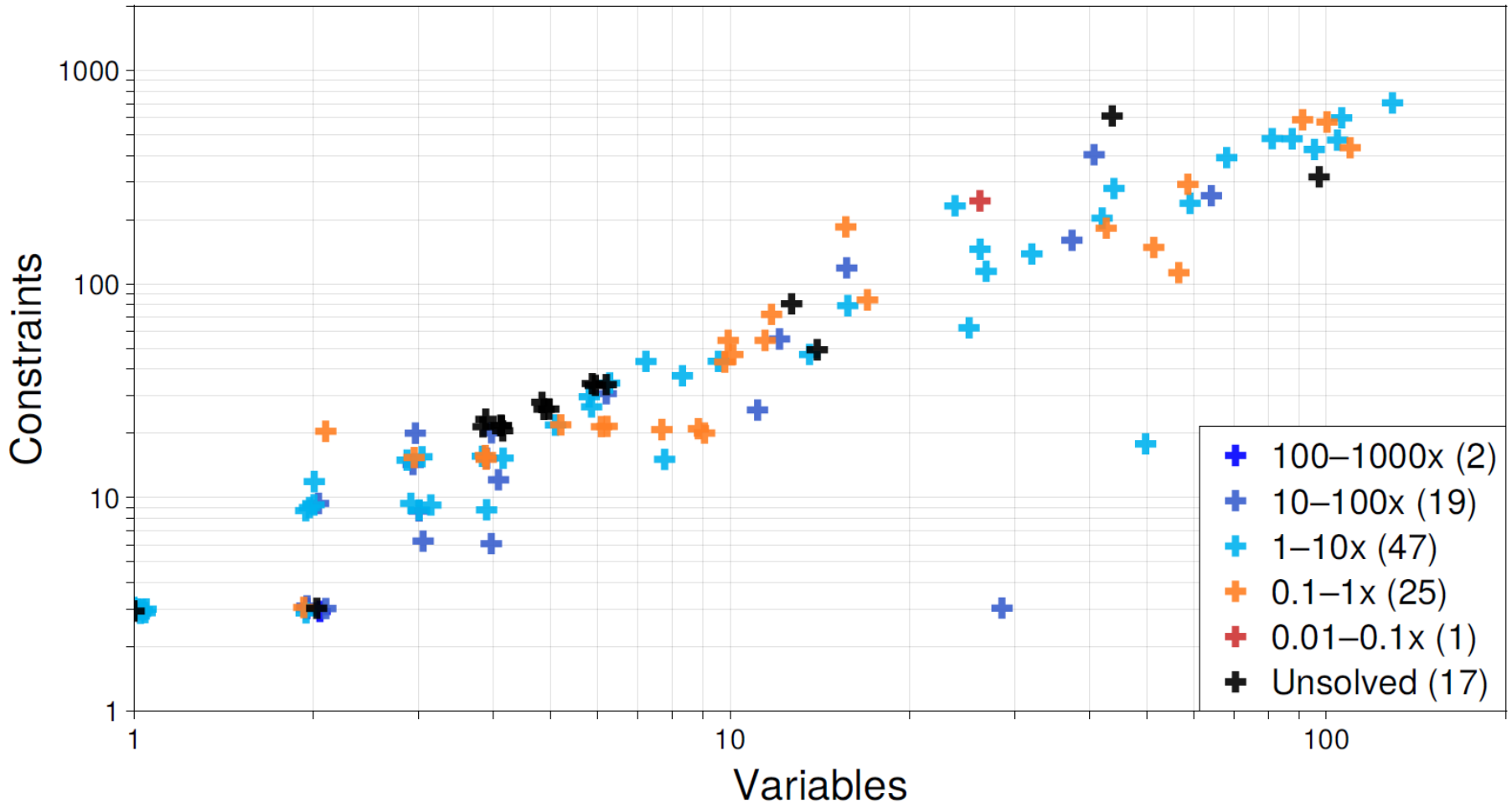
Fastest Solver in 61% of Problems



Fewest Problems Solved (85%) (FOMs are more sensitive to numerical issues)

18. Dolan, E and Moré, J. **Benchmarking optimization software with performance profiles.** *Math. Program. Ser. A*, 91: 201–213 (2002).

Results Distribution



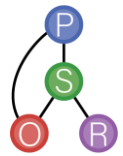
LP Solver Problem Dependence

Instance	Glop Primal Simplex	Glop Dual Simplex	Gurobi Barrier with Crossover	Gurobi Barrier without Crossover	PDLP High Precision	PDLP Low Precision
ex10	>1200	>1200	79.7	63.5	8.2	2.7
nug08-3rd	>1200	252.8	144.6	3.2	1.1	0.9
savsched1	>1200	>1200	156.0	22.6	46.4	32.4
wide15	>1200	20.8	>1200	>1200	916.4	56.3
buildingenergy	178.4	267.5	12.8	12.3	>1200	157.2
s250r10	12.1	520.6	15.2	16.4	>1200	>1200
Solver Version	OR-Tools 9.3	OR-Tools 9.3	Gurobi 9.0	Gurobi 9.0	OR-Tools 9.3	OR-Tools 9.3
`solver_specific_parameters`	(defaults)	`use_dual_simplex: true`	`Method 2, Threads 1`	`Method 2, Crossover 0, Threads 1`	`termination_criteria { eps_optimal_absolute: 1e-8 eps_optimal_relative: 1e-8 }`	`termination_criteria { eps_optimal_absolute: 1e-4 eps_optimal_relative: 1e-4 }`

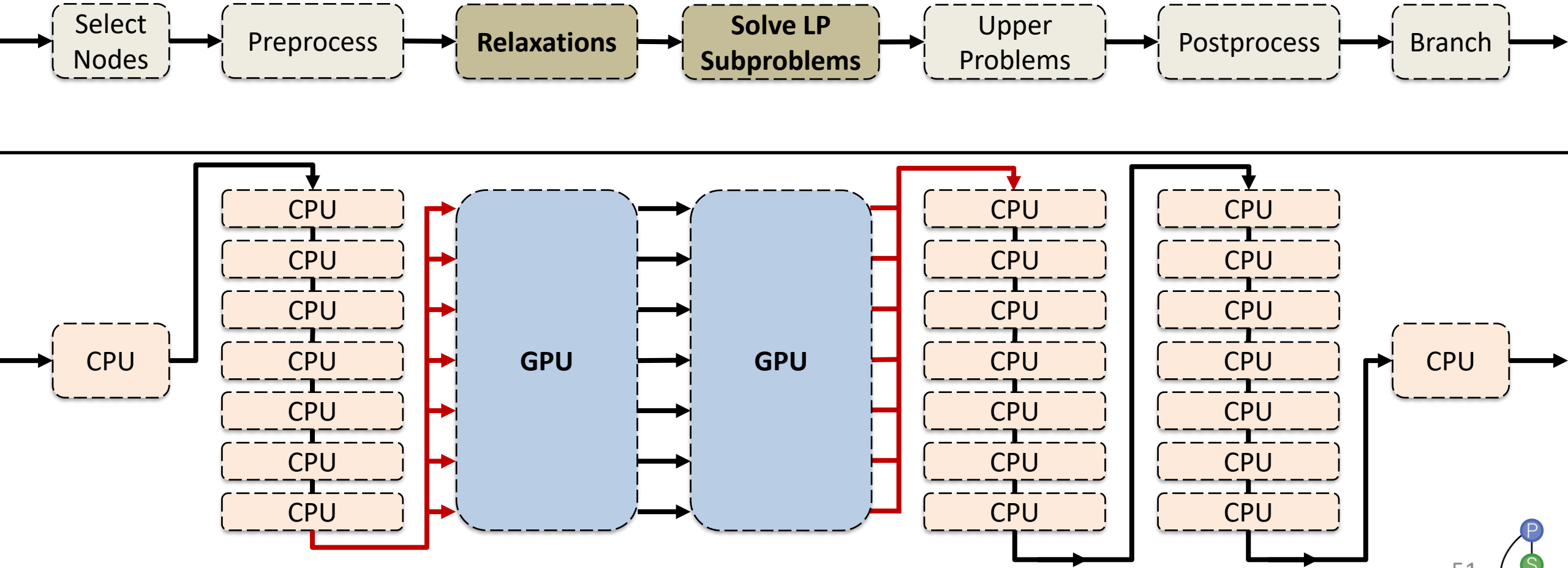
Fastest

Close to Fastest

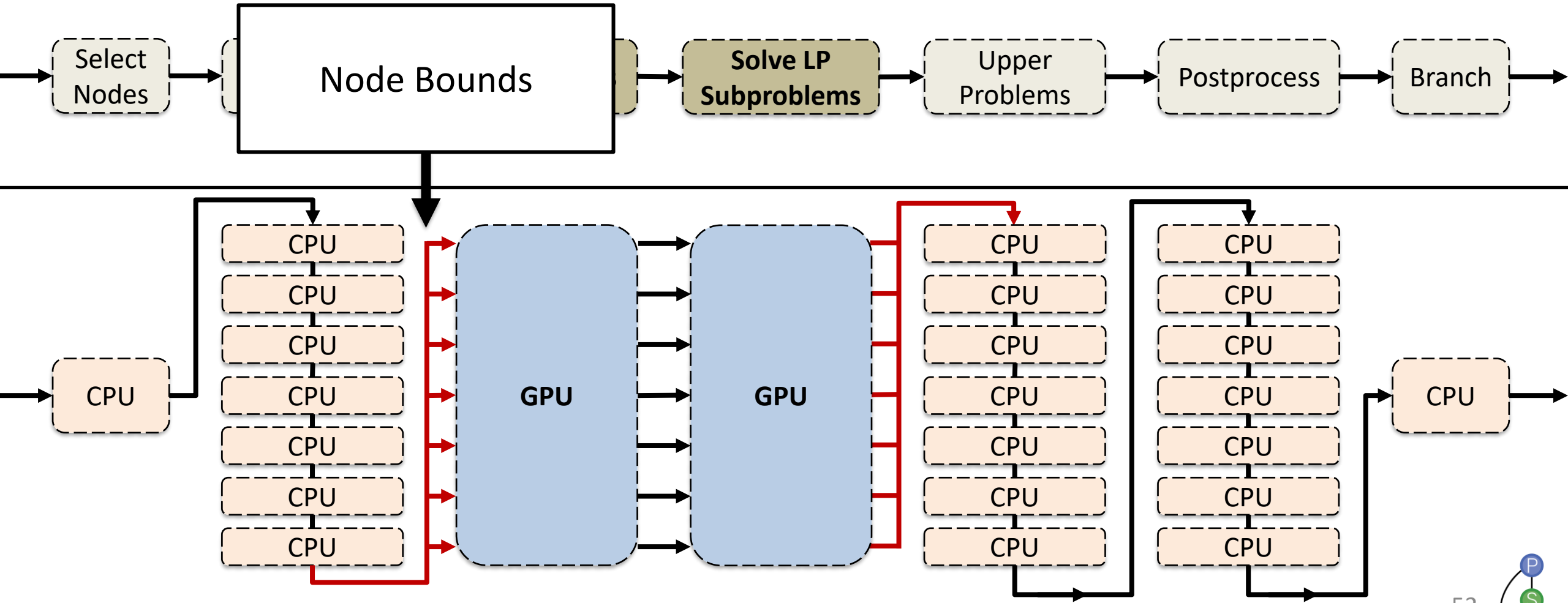
19. Google OR-Tools. **Advanced LP Solving.** Accessed 2025-08-22. https://developers.google.com/optimization/lp/lp_advanced



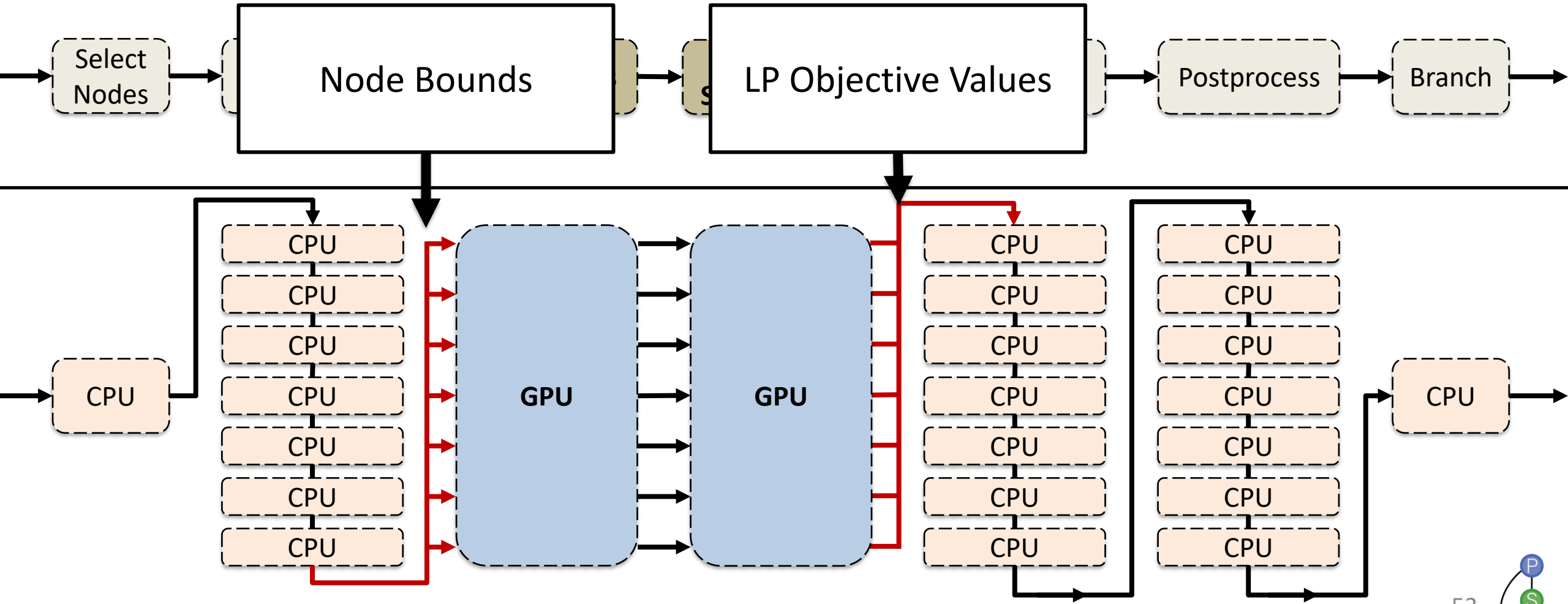
Integration into B&B



Integration into B&B



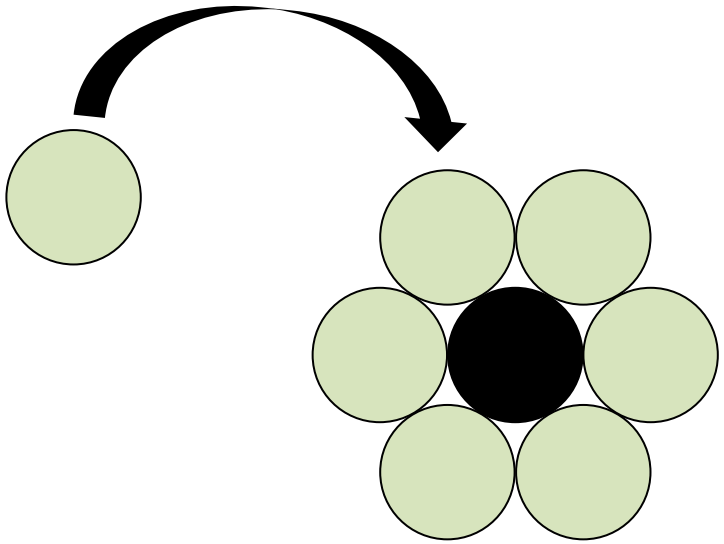
Integration into B&B



Numerical Example

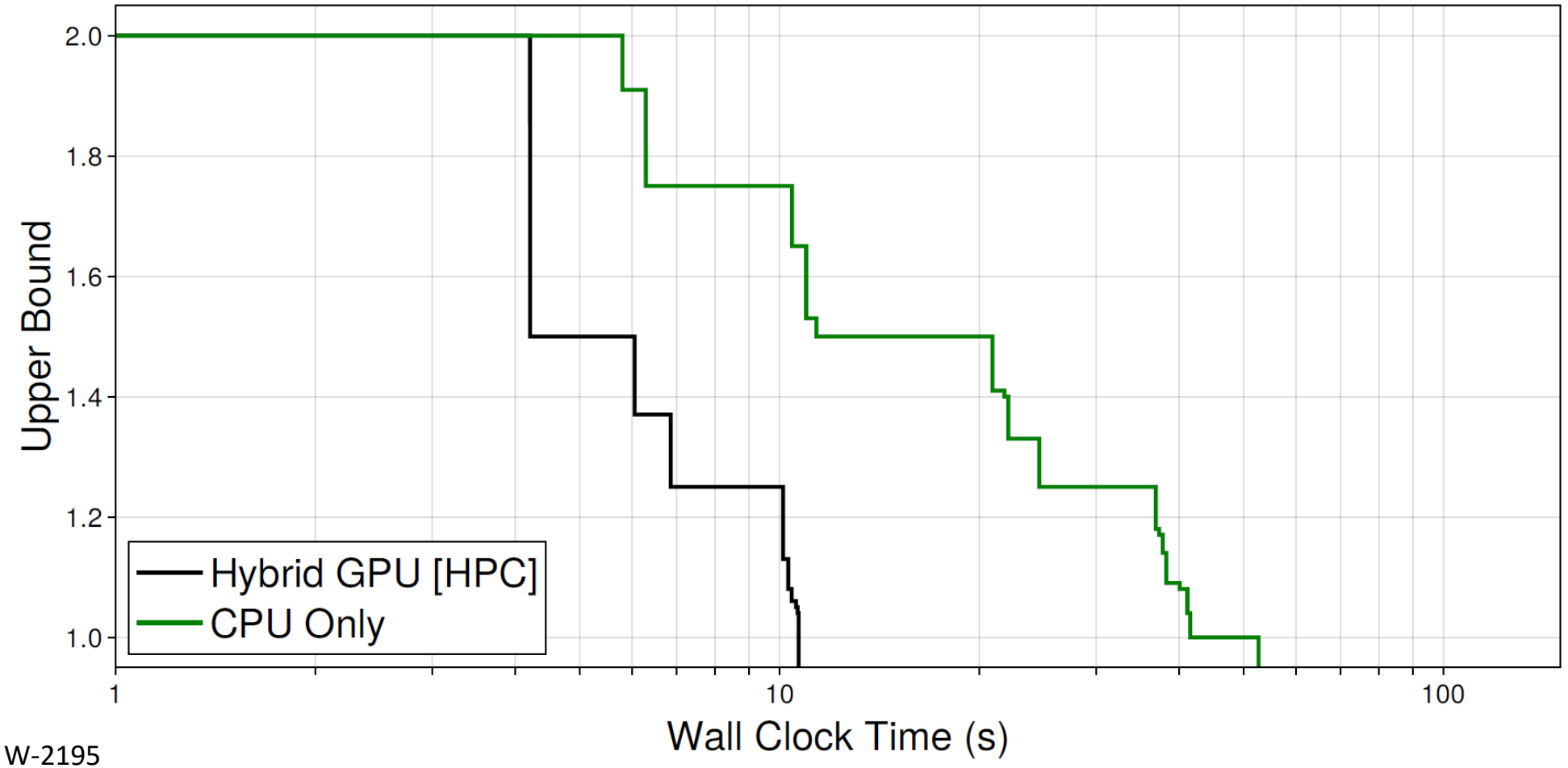
$$\max_{\mathbf{x},\alpha} \alpha$$

$$\begin{aligned}
\text{s.t.} \quad & \|\mathbf{x}^i\|^2 = 4 && \forall i \leq 7 \\
& 2\alpha + \sum_{k=1}^{2} x_k^i x_k^j \leq 4 && \forall i < j \leq 7 \\
& \alpha \geq 1 \\
& x_1^1 = -2 \\
& x_1^i \leq x_1^{i+1} && \forall i \leq 6 \\
& \mathbf{x}^i \in [-2, 2]^2 && \forall i \leq 7
\end{aligned}$$



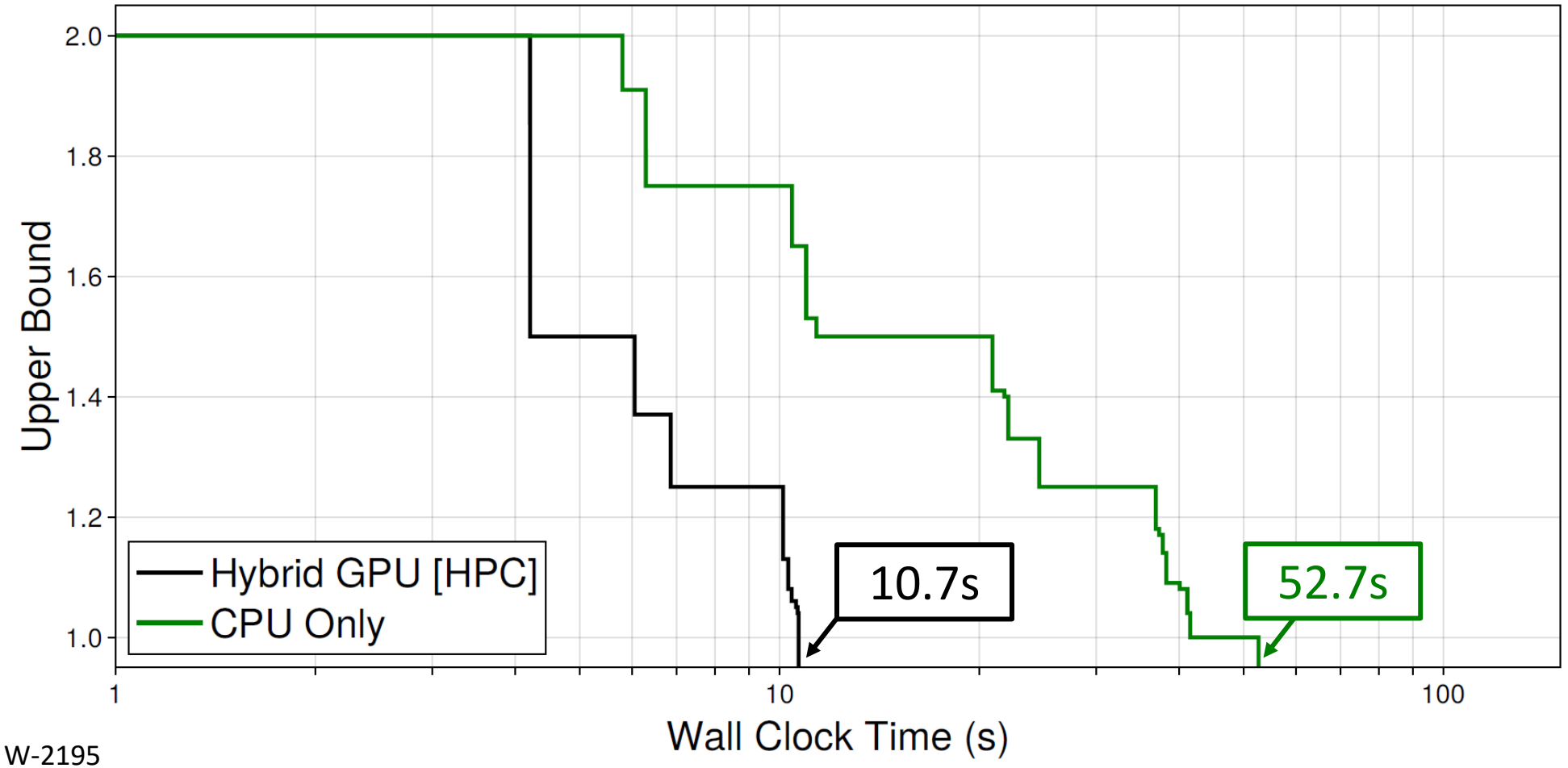
20. Liberti, L., et al. **New formulations for the Kissing Number Problem**. *Discrete Applied Mathematics*, 155(4): 1837–1841 (2007).

Example Convergence Plot



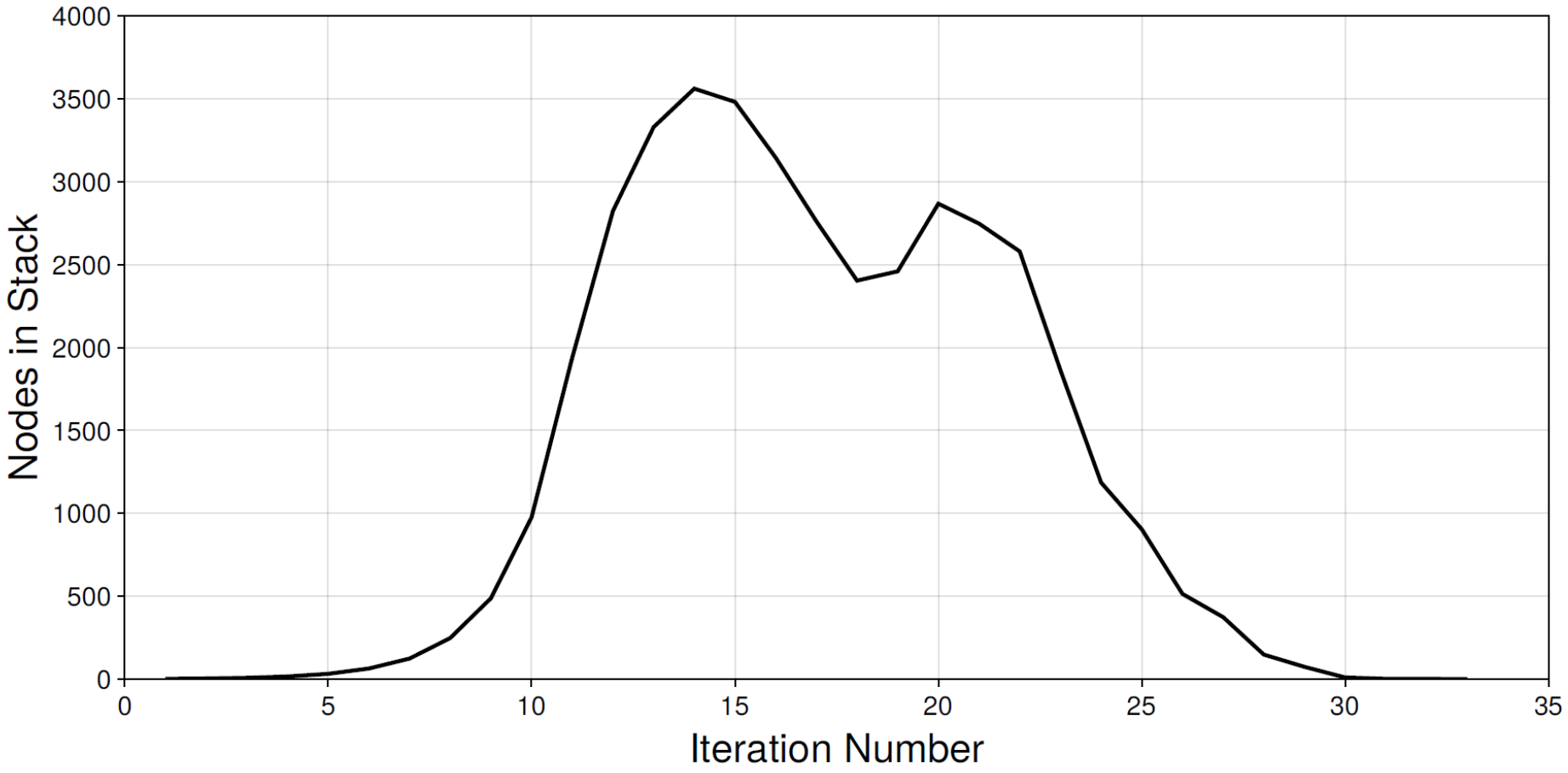
CPU: Intel Xeon W-2195
GPU [HPC]: NVIDIA Quadro GV100

Example Convergence Plot

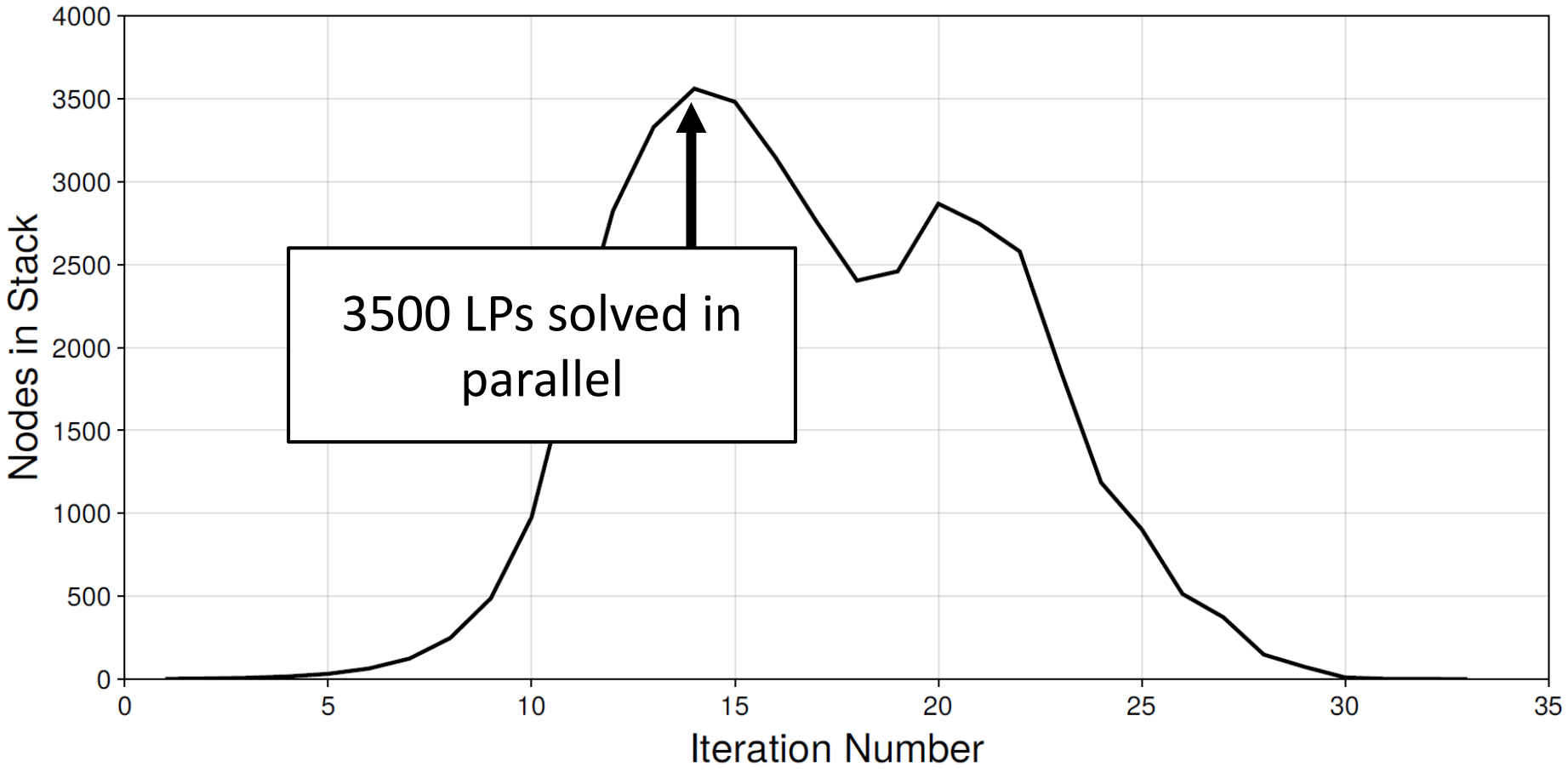


CPU: Intel Xeon W-2195
GPU [HPC]: NVIDIA Quadro GV100

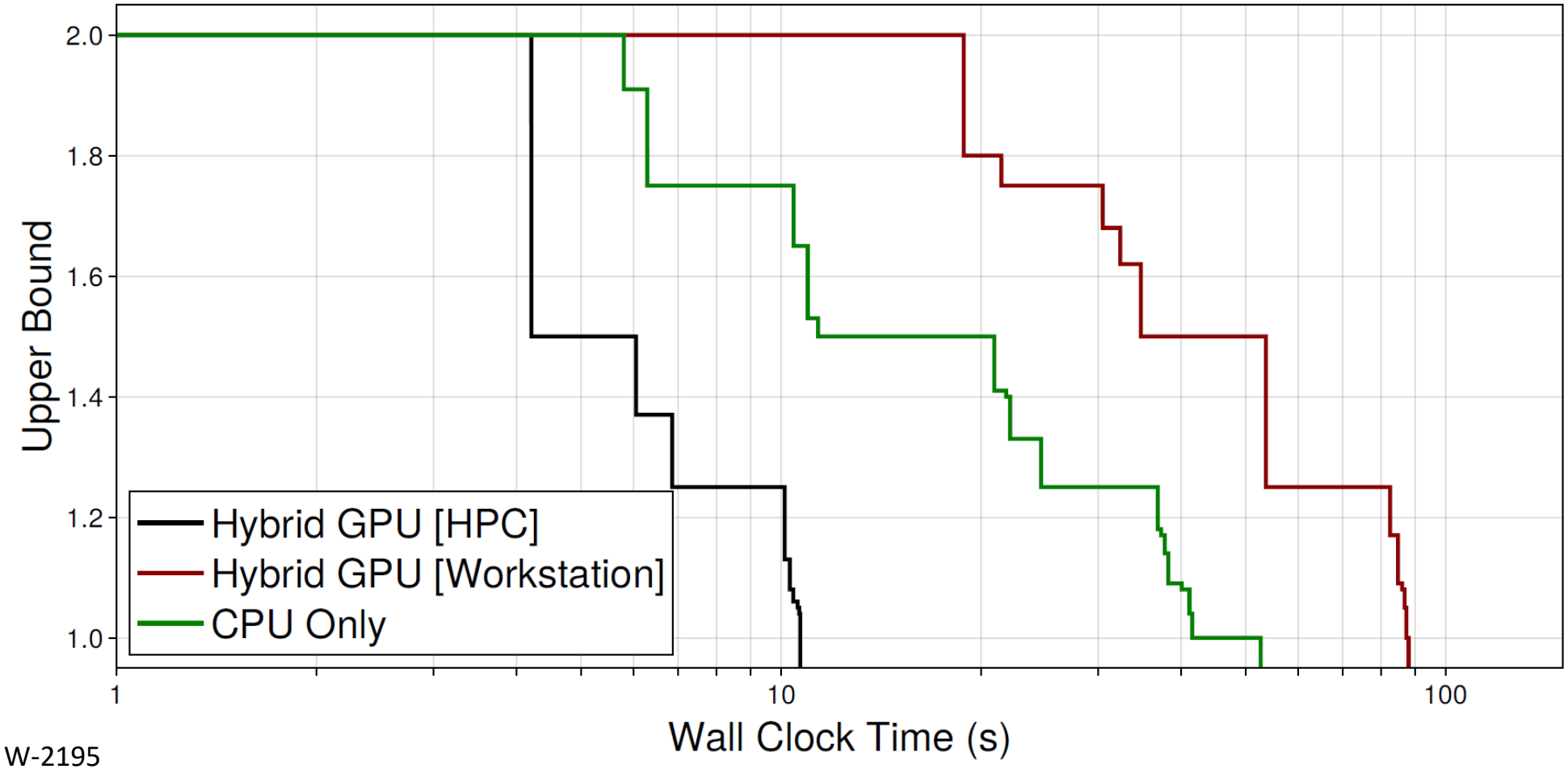
B&B Progress


Nodes in Stack
4000
3500
3000
2500
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
Iteration Number

B&B Progress

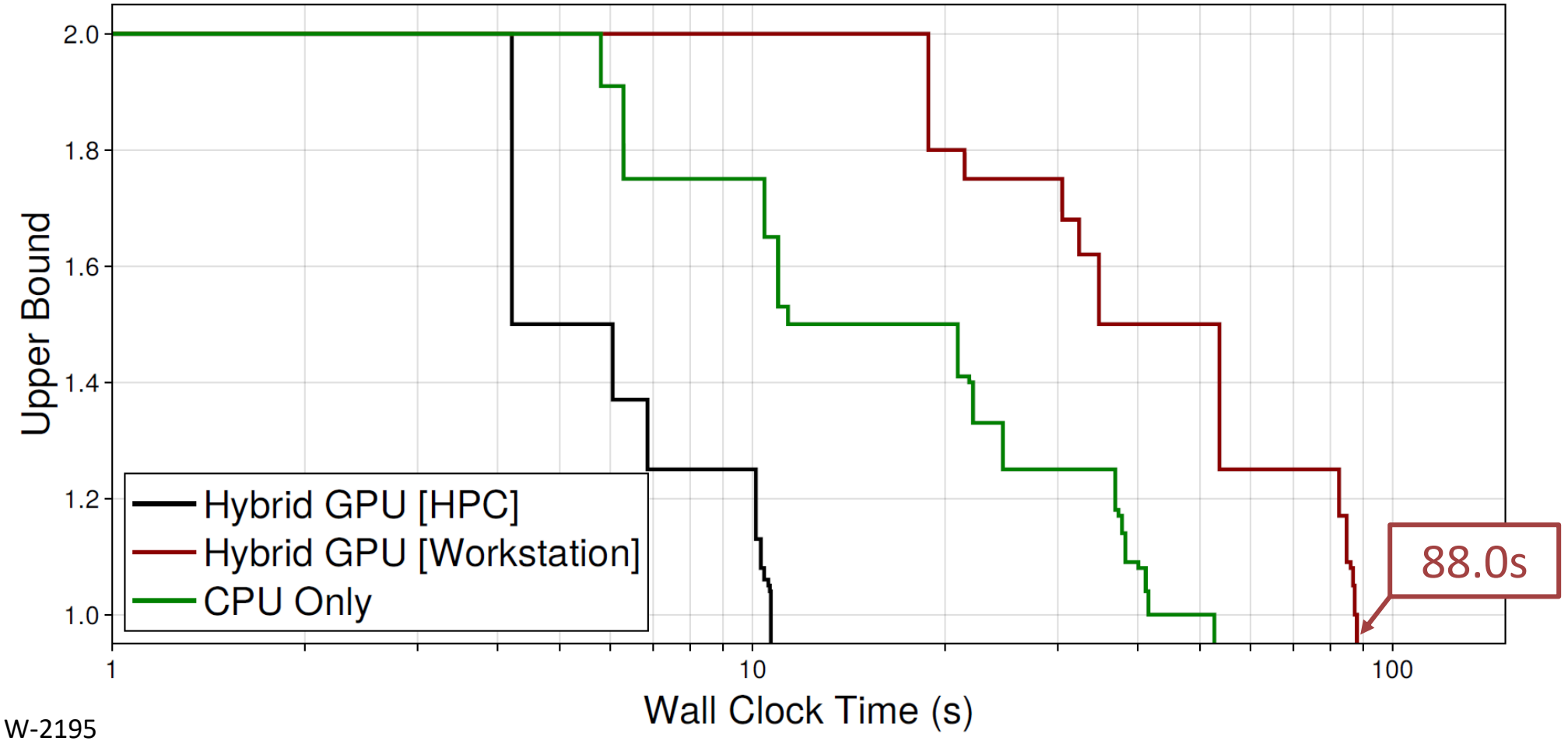

Nodes in Stack
4000
3500
3000
2500
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
Iteration Number
3500 LPs solved in parallel

Example Convergence Plot



CPU: Intel Xeon W-2195
GPU [HPC]: NVIDIA Quadro GV100
GPU [Workstation]: NVIDIA Quadro T2000

Example Convergence Plot



CPU: Intel Xeon W-2195
GPU [HPC]: NVIDIA Quadro GV100
GPU [Workstation]: NVIDIA Quadro T2000

Wrapping Up

- Applying GPU resources to address individual B&B nodes
  - Code generation for relaxations
  - Custom LP solver (BatchPDLP)
- Integration within EAGO allows minimization of CPU-GPU data transfer
- Integration within EAGO enables **any** nonconvex NLP to be solved using GPUs

SourceCodeMcCormick.jl Public

https://github.com/PSORLab/SourceCodeMcCormick.jl

Acknowledgements

Members of the Process Systems and Operations Research Laboratory at the University of Connecticut (https://psor.uconn.edu/)

Process Systems and Operations Research Laboratory

https://www.psor.uconn.edu

Funding:

National Science Foundation, Award No.: **2330054**

Pratt & Whitney Endowed Professorship

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the United States Government.