Technical University of Denmark

MPC Toolbox with GPU Accelerated Optimization Algorithms

Gade-Nielsen, Nicolai Fog; Jørgensen, John Bagterp; Dammann, Bernd

Published in:
The 10th European Workshop on Advanced Control and Diagnosis (ACD 2012)

Publication date:
2012

Document Version
Publisher's PDF, also known as Version of record

Citation (APA):
Gade-Nielsen, N. F., Jørgensen, J. B., & Dammann, B. (2012). MPC Toolbox with GPU Accelerated Optimization Algorithms. In The 10th European Workshop on Advanced Control and Diagnosis (ACD 2012) Technical University of Denmark (DTU).

The 10th European Workshop on Advanced Control and Diagnosis (ACD 2012)
Technical University of Denmark, Kgs. Lyngby, Denmark, Nov. 8-9, 2012

MPC Toolbox with GPU Accelerated Optimization Algorithms

Nicolai Fog Gade-Nielsen [*] **John Bagterp Jørgensen** [*,**]
Bernd Dammann [*]

[*] *DTU Informatics, Technical University of Denmark, 2800 Kgs Lyngby, Denmark (e-mail: {nfga,jbj,bd}@imm.dtu.dk)*
[**] *Center for Energy Resources Engineering, Technical University of Denmark*

Abstract: The introduction of Graphical Processing Units (GPUs) in scientific computing has shown great promise in many different fields. While GPUs are capable of very high floating point performance and memory bandwidth, its massively parallel architecture requires algorithms to be reimplemented to suit the different architecture. Interior point method can be used to solve convex optimization problems. These problems often arise in fields such as in Model Predictive Control (MPC), which may have real-time requirements for the solution time. This paper presents a case study in which we utilize GPUs for a Linear Programming Interior Point Method to solve a test case where a series of power plants must be controlled to minimize the cost of power production. We demonstrate that using GPUs for solving MPC problems can provide a speedup in solution time.

Keywords: Linear Programming, Interior Point Methods, Model Predictive Control, Graphical Processing Unit

1. INTRODUCTION

The Graphical Processing Unit (GPU) is the processing chip on modern graphic cards. Due of the increasing demand for advanced graphics, the GPU has developed into a massively parallel chip with very fast floating point computation and high memory bandwidth. These properties make them attractive for not only graphics rendering, but also many different scientific computations.

The use of GPUs for solving linear programming problems have been looked at before in Spampinato and Elster (2009), Bieling et al. (2010) and Lalami et al. (2011b) and even extended to use multiple GPUs in Lalami et al. (2011a). These papers focused on the use of the simplex method and all showed achieved speedups with the use of GPUs. While the simplex algorithm usually has good practical solve time, its worst-case is bounded in exponential time.

Unlike the simplex method, interior point methods are bounded by polynomial time. Interior point methods on GPUs have been looked at in Hyuk et al. (2007), which used GPGPU as it predates CUDA and OpenCL. A more recent use of GPUs for interior point methods is Smith et al. (2012), which uses a matrix-free interior point method and the GPU is used for sparse matrix operations in their preconditioned conjugate gradient solver when computing the step.

In this paper, we describe the implementation of a Mehrotra predictor-corrector interior point method using GPUs to lower the computation time. We test the implementation on a linear programming problem from Model Predictive Control (MPC). The test case is highly structured, which allows us to exploit the structure of the matrices to further improve the performance of the matrix computations. The GPU is utilized to do all the matrix operations as well as the factorization and backsolve.

The solvers presented in this paper are a part of a MPC toolbox, which aims to contain many different tools used in MPC. Such tools include state space realization and efficient solvers which utilizes structure and GPUs.

This paper is organized as follows. The test case used to test the implementations is introduced in Section II. The implemented algorithm is briefly introduced in Section III. The structure of the matrices in the test case are described in Section IV. The different implementations are described in Section V and performance results of these are presented in Section VI. Finally a look into the future of the toolbox is described in Section VII and a conclusion is presented in Section VII.

1.1 GPU computing

GPU computing has shown itself to be extremely efficient at certain numerical computations. GPUs provide a significant increase in computing performance and memory bandwidth compared to traditional CPUs (Central Processing Units). The cost of these benefits is a more complicated programming model and certain restrictions on the type of workload which can fully utilize the performance. Not all numerical computations are equally suited for the architecture, due to the massive parallelism and restrictions on how memory can be accessed to achieve optimal performance.

There are currently two major programming models available for programming GPUs, CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language). CUDA is the programming model developed by NVIDIA and is only available on NVIDIA GPUs, while OpenCL is an open standard for heterogeneous programming, which works on GPUs across vendors and even on different architectures such as CPUs. While both of these technologies are available today, CUDA is currently more mature with several advanced libraries available. The implementation described in this paper is done using CUDA and utilizes CUBLAS and MAGMA to simplify the implementation. CUBLAS is a BLAS (Basic Linear Algebra Subprograms) implementation for CUDA developed by NVIDIA, which contains many basic vector and matrix operations such as matrix-vector and matrix-matrix multiplication. MAGMA is a LAPACK (Linear Algebra PACKage) implementation for heterogeneous systems, which contains more advanced algorithms such as Cholesky factorization.

2. TEST CASE

Our test case is a control problem from Model Predictive Control (MPC) related to power production. It is a simplified version of the problem in Edlund et al. (2009), where a series of power producers must be controlled in order to satisfy the power demand. Given a reference which is the power demand, the objective of the control problem is to ensure sufficient power is produced to satisfy demand while keeping production cost to the minimum. The different power producers each have several real-world constraints, such as bounds on the maximum power output of a power plant and reaction rate. For instance, a coal power plant may be cheap to use but also slow to react to changes in its control signal.

The objective function of the control problem is to minimize the cost of power production over a prediction horizon of N time steps. The objective function of such a problem can be written as

$$\phi = \sum_{k=0}^{N-1} c'_{u,k} u_k$$

where $c_{u,k}$ is the production cost and u_k is the control signal for each power plant at a given time step. Due to the constraints of the power producers, it may not always be possible to meet the power demand. Slack variables with a high cost are introduced to make the problem feasible and can be considered as purchasing the remaining power from another supplier, such as another power company.

The objective function with slack variables introduced is

$$\phi = \sum_{k=0}^{N-1} c'_{u,k} u_k + c'_{s,k+1} s_{k+1}$$

where $c_{s,k+1}$ is the cost of power from another supplier and s_{k+1} is the slack variables for a given time step. The entire control problem with constraints can then be written as a constrained optimization problem as

$$\begin{array}{lll}
\min\limits_{u,s} & \phi = \sum\limits_{k=0}^{N-1} c'_{u,k} u_k + c'_{s,k+1} s_{k+1} & \\
\text{s.t.} & x_{k+1} = A x_k + B u_k & k = 0, \dots, N-1, \\
 & y_k = C x_k & k = 1, \dots, N, \\
 & u_{min} \leq u_k \leq u_{max} & k = 0, \dots, N-1, \\
 & \Delta u_{min} \leq u_k \leq \Delta u_{max} & k = 0, \dots, N-1, \\
 & y_k \geq r_k - s_k & k = 1, \dots, N, \\
 & s_k \geq 0 & k = 1, \dots, N,
\end{array}$$

where x_{k+1} is the state computed from the discrete-time state space realization, y_k is the power production, u_{min} and u_{max} are the lower and upper bounds on the control signal, Δu_{min} and Δu_{max} are the rate of movement bounds on u_k and r_k is the power demand reference.

In order to solve this control problem with a standard interior point method, we reformulate the problem into the standard form

$$\begin{array}{ll}
\min\limits_{x} & g'x \\
\text{s.t.} & Ax \geq b,
\end{array}$$

where

$$g = \begin{bmatrix} c_u \\ c_s \end{bmatrix}, x = \begin{bmatrix} u \\ s \end{bmatrix}, A = \begin{bmatrix} -I & 0 \\ I & 0 \\ 0 & -I \\ -\Psi & 0 \\ \Psi & 0 \\ \Gamma & I \end{bmatrix} b = \begin{bmatrix} U_{min} \\ U_{max} \\ 0 \\ b_l \\ b_u \\ R - \Phi x_0 \end{bmatrix} \qquad (1)$$

.

as described in Edlund et al. (2009).

3. ALGORITHM

The algorithm used to solve the test case is a primal-dual interior-point method for linear programming problems with only inequality constraints in the following form,

$$\begin{array}{ll}
\underset{x}{min} & f(x) = g'x \\
s.t. & Ax \geq b
\end{array}$$

By only having inequality constraints, it is possible to reduce the problem to a symmetric positive-definite Hessian and use Cholesky factorization to solve the affine step. The factorization can be done once per iteration and reused for the corrector step. The algorithm is shown in Listing 1.

4. STRUCTURE OF MATRICES

The interior point method consists of mostly different vector operations, which are very cheap to compute. The largest computational tasks in the algorithm are matrix-matrix multiplication when computing the Hessian, H, and its subsequent Cholesky factorization as well as the matrix-vector multiplications with the constraint matrix, A.

A straight-forward implementation is to simply compute with a dense A-matrix, however by looking at the matrices

Listing 1. Primal-Dual Interior Point Method

```
% Compute residuals
$r_L = g - A'\lambda$
$r_S = s - Ax + b$
$r_{SL} = S\Lambda e$
$\mu = \frac{S'\Lambda}{m}$
while ($\mu > tol$ or $||[rH; rA; rC; rS]||_{\inf} > tol$)
    % Factorization
    $H = A'(S^{-1}\Lambda)A$
    $[L] = chol(H)$
    % Affine step
    $\bar{r} = A'(S^{-1}(r_{SL} - \Lambda r_S))$
    $\bar{g} = r_L + \bar{r}$
    $\Delta x = L'\backslash(L\backslash(-\bar{g}))$
    $\Delta s = -r_S + A\Delta x$
    $\Delta\lambda = -S^{-1}(r_{SL} + \Lambda\Delta s)$
    Compute step length $\alpha$ such that
        $\phi + \alpha\Delta\phi \geq 0$ and $s + \alpha\Delta s \geq 0$.
    % Center parameter
    $s_{aff} = s + \alpha\Delta s$
    $\lambda_{aff} = \lambda + \beta\Delta\lambda$
    $\mu_{aff} = \frac{s'_{aff}\lambda_{aff}}{m}$
    $\sigma = (\frac{\mu_{aff}}{\mu})^3$
    $\tau = \sigma\mu e$
    % Center corrector
    $r_{SL} = r_{SL} + \Delta S\Delta\Lambda e - \tau$
    % Solve center-corrector step
    $\bar{r} = A'(S^{-1}(r_{SL} - \Lambda r_S))$
    $\bar{g} = r_L + \bar{r}$
    $\Delta x = L'\backslash(L\backslash(-\bar{g}))$
    $\Delta s = -r_S + A\Delta x$
    $\Delta\lambda = -S^{-1}(r_{SL} + \Lambda\Delta s)$
    Compute step length $\alpha$ such that
        $\phi + \alpha\Delta\phi \geq 0$ and $s + \alpha\Delta s \geq 0$.
    % Take step
    $x = x + (\eta * \alpha) * \Delta x$
    $\lambda = \lambda + (\eta * \alpha) * \Delta\lambda$
    $s = s + (\eta * \alpha) * \Delta s$
    % Compute residuals
    $r_L = g - A'\lambda$
    $r_S = s - Ax + b$
    $r_{SL} = S\Lambda e$
    $\mu = \frac{S'\Lambda}{m}$
end
```

in (1), it is obvious that they have are highly structured and sparse. Most MPC systems consists of a series of bound constraints and rate of movement constraints, as well as the dynamics of the system. Bound constraints appear as simple identity matrices in the constraint matrix, while rate of movement constraints also have a highly sparse banded structure, denoted as Ψ, which can be utilized to do efficient sparse-matrix multiplication. The output constraints are computed with Γ, which is a Toeplitz matrix. While its lower triangle is dense, it contains a lot of repetition which makes it possible to store the matrix much more efficiently by simply storing the first set of columns. This is also noted in Edlund et al. (2009), where they describe efficient methods to compute with their A-matrix. These methods can be adapted to our test system, as well as other LP MPC systems with bound constraints and rate of movement constraints.

Table 1. Solver implementation

Name	Implemented in	Exploits structure	GPU
LPippd	MATLAB	No	No
LPippdGPU	MATLAB	No	Yes
LPippdExGPU	C + CUDA	No	Yes
LPippdPP	MATLAB	Yes	No
LPippdPPGPU	MATLAB	Yes	Yes
LPippdPPExGPU	C + CUDA	Yes	Yes

5. IMPLEMENTATIONS

We've implemented the solver described in listing 1 in multiple versions to study the importance of utilizing structure as well as the benefit of GPUs. The solver has been implemented in MATLAB, both with and without MATLAB GPU support, as well as a version implemented entirely with C and CUDA by using BLAS and LAPACK libraries for GPU. Additionally, each solver has been implemented as both entirely dense solvers and as a solver which utilizes the structure of the A-matrix. A complete list of the different solver implementations is shown in Table 1.

5.1 LPippd

This is a straight-forward MATLAB implementation of the algorithm using dense matrix multiplications.

5.2 LPippdGPU

LPippdGPU is the MATLAB implementation of the algorithm, which uses the MATLABs built-in GPU support for accelerating some of the operations. GPU support in MATLAB was added in version 2011a and straightforward to use on systems equipped with a CUDA-capable device. The function `GPUArray` can be used to copy data to the GPU and any operations on GPU-resident data is done automatically on the GPU if MATLAB has support for the operation. However MATLAB only supports a limited set of operations on the GPU and any operations which are not supported on GPU by MATLAB significantly reduces the performance due to memory transfers between the CPU and GPU. Additionally, only dense matrices are supported. The function `gather` can be used to copy data back from the GPU to the CPU manually.

In our initial implementation of LPippdGPU, we tried to simply copy all the matrices and vectors to GPU and do the entire algorithm on the GPU. The matrix multiplication operations and the Cholesky factorization was accelerated substantially. However the performance of computation of the step length had worsened. This is due to the lack of support for certain operations on the GPU in MATLAB, which caused MATLAB to copy data back to the CPU, compute on the CPU and then copy back to the GPU.

Since the primary workload is in the matrix-multiplication for creating the Hessian and the Cholesky factorization, we modified the LPippdGPU to simply accelerate these two parts instead. This allows MATLAB to use the CPU for the step length computation, while still accelerating the main workload with the GPU. This results in a much better performance than the previous version.

5.3 LPippdExGPU

LPippdExGPU is the implementation of the algorithm in C and CUDA. Many of its operations can be done using BLAS and LAPACK routines, while a few has required an implementation of a kernel for optimal performance.

The operations on diagonal matrices have been done using a vector to store the diagonal and doing element-wise vector operations to compute the result, such as the computation of ΛS^{-1} in the Hessian matrix multiplication. Kernels for element-wise multiplication and division have been implemented, as element-wise operations are not part of BLAS.

The computation of the step length is a specialized type of reduction, which must compute $-S(\Delta S)^{-1}$ and find the smallest diagonal value, but only for elements where $\Delta S < 0$. This is done by implementing a specialized reduction kernel based on Mark Harris' efficient reduction kernel described in Harris (2007).

All the operations are done on the GPU to avoid the data transfer between CPU and GPU.

5.4 LPippdPP variants*

The LPippdPP* variants of the solver are the ones utilizing the structure of the A-matrix. The operations for matrix-vector multiplication as well as the computation of the Hessian have been implemented as functions which are specific for our test case. When the solver needs to do operations with the A-matrix, it simply invokes these functions instead of using the A-matrix to do matrix-vector and matrix-matrix multiplication.

6. RESULTS

The implemented methods have been tested on a computer system at GPUlab at the Technical University of Denmark (DTU). The system contains a Intel Core i7 930 CPU, 12 GB RAM and a NVIDIA Tesla C2050 GPU and contains MATLAB 2011b, CUDA 4.1 and MAGMA 1.1.

The external CUDA versions, LPippdExGPU and LPippdPPExGPU, are implemented as a separate program as it wasn't possible to make MEX work properly with CUBLAS and MAGMA due to dynamic library linking problems. When calling these solvers, MATLAB writes the problem matrices to disk and invokes an external program through the `system` function. The external program then reads the problem from disk, solves the problem with the CUDA solver and writes the solution to disk, which is then read back into MATLAB. The cost of this is included in the solution time in our results, even though the actual solve time is slightly less due to the overhead of the I/O.

There are two different parameters which can be increased to produce a larger system and determine how the solvers performance scale. We can either increase the prediction horizon and keep the number of power plants fixed or we can keep a fixed prediction horizon and increase the number of power plants. The effect of these two changes on the A-matrix is very different.

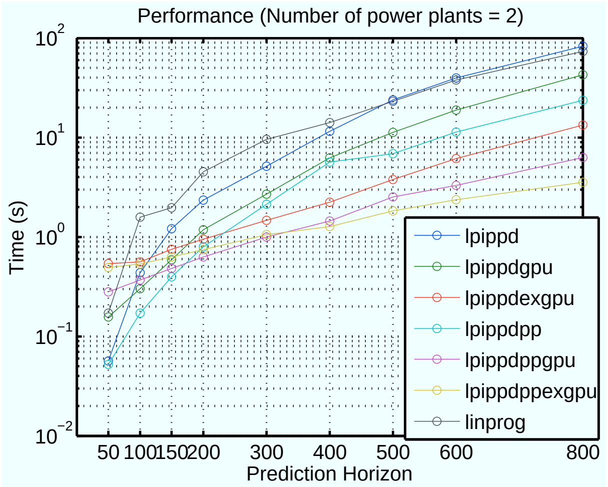

Fig. 1. Performance for variable horizon with two plants

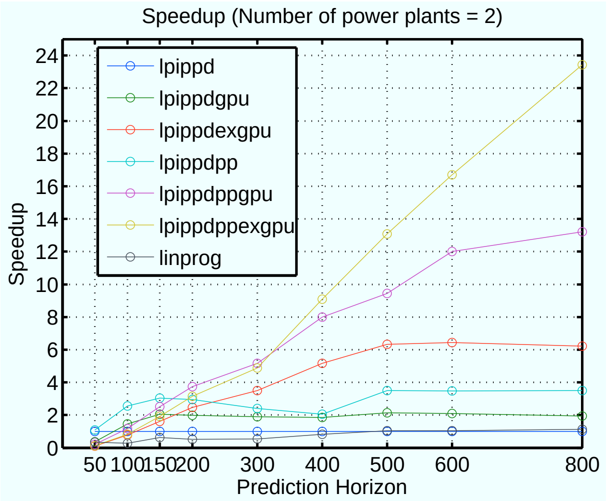

Fig. 2. Speedup for variable horizon with two plants

6.1 Increasing the prediction horizon

Increasing the prediction horizon will increase the number of dense rows in the A-matrix by increase the number of rows in Γ which represents the dynamics in the system. While the A-matrix remains sparse due to the bound and rate-of-movement constraints, the increased size of Γ increases the use of dense matrix-matrix multiplication in the solver.

The test is done by benchmarking the algorithm on a problem with two power plants with an increasing prediction horizon. The solution time of the different solvers is shown on Figure 1 and a speed-up plot with LPippd as reference is shown on Figure 2.

The speedup is approximately 2x by simply using MATLABs built-in GPU support and 6x by using own GPU implementation when the problem is large enough. MATLABs own solver linprog seems to suffer on this type of system, as it handles matrices as sparse and only achieves a performance similar to our own naive LPippd.

By utilizing the matrix structure efficiently, we can achieve a significant boost to the performance of the algorithm.

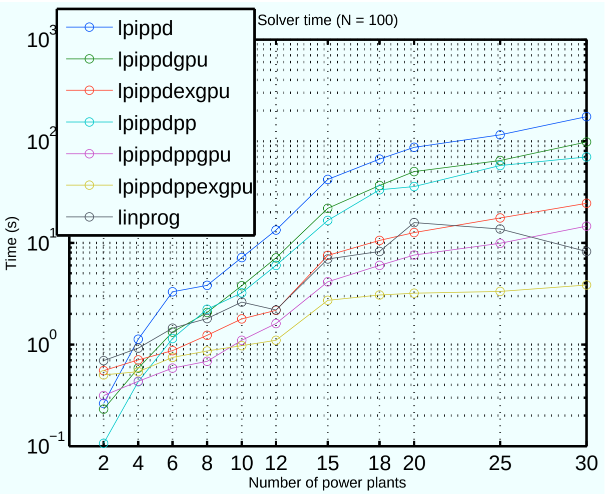

Fig. 3. Performance for fixed horizon with variable plants

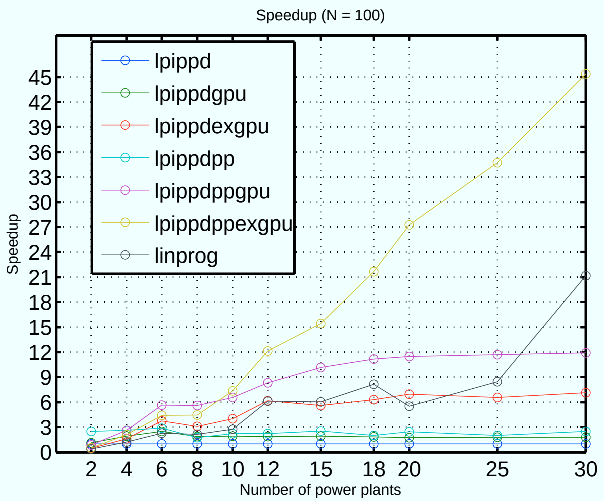

Fig. 4. Speedup for fixed horizon with variable plants

The simple MATLAB implementation achieves a 4x speedup, while the GPU version manages a 13x speedup. The speedup in our own GPU implementation outperforms all of the other solvers by achieving a speedup of up to 23x compared to LPippd, despite including the cost of calling a separate program to do the computation.

An example of the solution of the test case for prediction horizon of 500 is shown in figure 5.

6.2 Increasing the number of power plants

Increasing the number of power plants does not affect the number of rows in Γ. This results in a larger system which is highly sparse, which is particularly beneficial for MATLABs own solver, linprog, as it handles matrices as sparse internally.

The test is done by keeping the prediction horizon fixed to 100 and increasing the number of power plants. The solution time of the different solvers is shown on Figure 3 and a speed-up plot with LPippd as reference is shown on Figure 4.

As expected, MATLABs linprog achieves much better performance in this case and manages a speedup of up to 21x compared to the dense implementation in MATLAB, even outperforming GPU implementations in MATLAB which is likely due to the high sparsity of the *A*-matrix. Clearly it is necessary to exploit the structure of the matrix to achieve decent performance in this case. Our own GPU implementation which does exploit structure still manages to outperform linprog and is roughly two times faster than linprog.

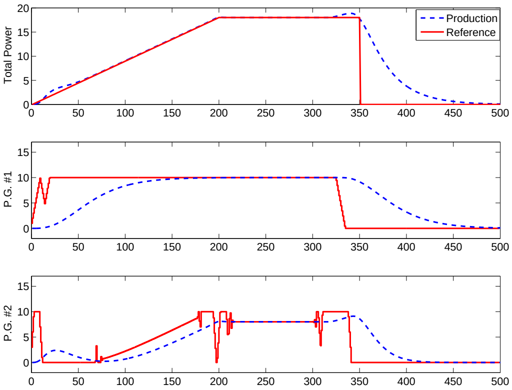

Fig. 5. Solution for N = 500 with two plants

7. FUTURE

The toolbox containing the solvers is an ongoing work and the aim is to provide an efficient toolbox for MPC without reliance on MATLAB and other expensive program packages for production use, but still providing support for these tools during the development phase. In the future, more functions will be added to simplify the use of the solvers and set up of MPC problems using the toolbox.

8. CONCLUSION

We have implemented a GPU-accelerated interior point method for solving linear programming problems with inequality constraints. The solver is implemented in both MATLAB and in C with CUDA and has been tested on a test case from MPC. Variants of the solver which exploit the structure of the particular test case have also been implemented and the importance of exploiting structure in sparse optimization problems has been demonstrated. We have shown that the use of the GPU can provide a speedup in the solution time. The speedup from using the GPU is higher when the problem is more dense, but the GPU can still provide a speedup even when problem is sparse if the matrix structures are exploited.

REFERENCES

Bieling, J., Peschlow, P., and Martini, P. (2010). An efficient gpu implementation of the revised simplex method. In *IPDPS Workshops*, 1–8. IEEE.

Edlund, K., Sokoler, L.E., and Jørgensen, J.B. (2009). A primal-dual interior-point linear programming algorithm for mpc. In *CDC*, 351–356. IEEE.

Harris, M. (2007). Optimizing parallel reduction in cuda.

Hyuk, J., Jung, and O'leary, D.P. (2007). Implementing an interior point method for linear programs on a cpu-gpu system.

Lalami, M.E., Baz, D.E., and Boyer, V. (2011a). Multi gpu implementation of the simplex algorithm. In *HPCC'11*, 179–186.

Lalami, M.E., Boyer, V., and El-Baz, D. (2011b). Efficient implementation of the simplex method on a cpu-gpu system. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, 1999–2006. IEEE Computer Society, Washington, DC, USA.

Smith, E., Gondzio, J., and Hall, J. (2012). Gpu acceleration of the matrix-free interior point method. In *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, 681–689. Springer-Verlag, Berlin, Heidelberg.

Spampinato, D.G. and Elster, A.C. (2009). Linear optimization on modern gpus. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, 1–8. IEEE.