GPU acceleration of the matrix-free interior point method

E. Smith, J. Gondzio and J. A. J. Hall*

School of Mathematics
and Maxwell Institute for Mathematical Sciences
The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ
United Kingdom.

Technical Report ERGO-11-008†

12th May 2011

Abstract

Interior point methods (IPM) with direct solution of the underlying linear systems of equations have been used successfully to solve very large scale linear programming (LP) problems. However, the limitations of direct methods for some classes of problems have led to iterative techniques being considered. The *matrix-free* method is one such approach and is so named since the iterative solution procedure requires only the results of operations $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$, where A is the matrix of constraint coefficients. Thus, in principle, it may be applied to problems where A is not known and only an oracle is available for computing $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. Since the computational cost of these operations may well dominate the total solution time for the problem, it is important that the techniques used to perform them are efficient.

This paper outlines the matrix-free interior point method and, for several classes of LP problems, demonstrates its overwhelmingly superior performance relative to the simplex method and IPM with equations solved directly. The dominant cost of the operations $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$ motivates their implementation on a GPU to yield further performance gains. Different computational schemes for these sparse matrix-vector products are discussed. A comparison of the speed-up achieved using a many-core GPU implementation with that for a multi-core CPU implementation indicates the former has better potential.

Keywords: Interior point methods, Linear programming, Matrix-free methods, Parallel sparse linear algebra

*Email: J.A.J.Hall@ed.ac.uk

†For other papers in this series see http://www.maths.ed.ac.uk/ERGO/

1 Introduction

Since they first appeared in 1984 [12], interior point methods (IPM) have been a viable alternative to the simplex method as a means of solving linear programming (LP) problems [17]. The major computational cost of IPM is the direct solution of symmetric positive definite systems of linear equations. However, the limitations of direct methods for some classes of problems have led to iterative techniques being considered [2, 4, 13]. The *matrix-free* method of Gondzio [7] is one such approach and is so named since the iterative solution procedure requires only the results of products between the matrix of constraint coefficients and a (full) vector. This paper demonstrates how the performance of the matrix-free IPM may be accelerated significantly using a Graphical Processing Unit (GPU) via techniques for sparse matrix-vector products that exploit common structural features of LP constraint matrices.

Section 2 presents an outline of the matrix-free IPM that is sufficient to motivate its linear algebra requirements. Results for three classes of LP problems demonstrate its overwhelmingly superior performance relative to the simplex method and IPM with equations solved directly. Further analysis shows that the computational cost of the matrix-free IPM on these problems is dominated by the iterative solution of linear systems of equations which, in turn, is dominated by the cost of matrix-vector products. Techniques for evaluating products with LP constraint matrices on multi-core CPU and many-core GPU are developed in Section 3. These techniques exploit commonly-occurring structural features of sparse LP constraint matrices. Results demonstrate such acceleration in the evaluation of the matrix-vector products that there is a significant speed-up in the overall solution time for the LP problems, with the GPU implementation offering better potential for further performance enhancement. Conclusions and suggestions for future work are offered in Section 4.

2 The matrix-free interior point method

The theory of interior point methods is founded on the following general primal-dual pair of linear programming (LP) problems.

$$\begin{array}{cc|cc} & \text{Primal} & & \text{Dual} \\ \min & \boldsymbol{c}^T\boldsymbol{x} & \max & \boldsymbol{b}^T\boldsymbol{y} \\ \text{s. t.} & A\boldsymbol{x} = \boldsymbol{b} & \text{s. t.} & A^T\boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c} \\ & \boldsymbol{x} \geq \boldsymbol{0} & & \boldsymbol{y} \text{ free}, \boldsymbol{s} \geq \boldsymbol{0}, \end{array} \tag{1}$$

where $A \in \mathbb{R}^{m\times n}$ has full row rank $m \leq n$, $\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{c} \in \mathbb{R}^n$ and $\boldsymbol{y}, \boldsymbol{b} \in \mathbb{R}^m$. The first order optimality conditions for these problems can be written as

$$\begin{array}{rcl} A\boldsymbol{x} & = & \boldsymbol{b} \\ A^T\boldsymbol{y} + \boldsymbol{s} & = & \boldsymbol{c} \\ XS\boldsymbol{e} & = & \boldsymbol{0} \\ (\boldsymbol{x},\ \boldsymbol{s}) & \geq & \boldsymbol{0} \end{array} \tag{2}$$

where X and S are diagonal matrices whose entries are the components of vectors $\boldsymbol{x}$ and $\boldsymbol{s}$ respectively and $\boldsymbol{e}$ is the vector of ones. The third equation $XS\boldsymbol{e} = \boldsymbol{0}$ is referred to as the complementarity condition and can be rewritten as $x_j s_j = 0$, $\forall j = \{1, 2, \ldots, n\}$. It implies that at least one of x_j and s_j must be zero at the optimal solution of (1).

Interior point methods perturb the complementarity condition, replacing $x_j s_j = 0$ by $x_j s_j = \mu$. As μ is driven to zero in the course of a sequence of iterations, the vectors $\boldsymbol{x}$ and $\boldsymbol{s}$ partition into zero and nonzero components. In each iteration, a search direction is computed that maintains the first two optimality conditions in (2) and the perturbed complementarity condition by solving

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^T & I_n \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta x} \\ \boldsymbol{\Delta y} \\ \boldsymbol{\Delta s} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\xi_p} \\ \boldsymbol{\xi_d} \\ \boldsymbol{\xi_\mu} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} - A\boldsymbol{x} \\ \boldsymbol{c} - A^T\boldsymbol{y} - \boldsymbol{s} \\ \mu\boldsymbol{e} - XS\boldsymbol{e} \end{bmatrix}. \tag{3}$$

By using the sets of equations in (3) to eliminate first $\boldsymbol{\Delta s}$, and then $\boldsymbol{\Delta x}$, the following symmetric positive definite **normal equations** system is obtained

$$(A\Theta A^T)\boldsymbol{\Delta y} = \boldsymbol{g}, \tag{4}$$

where $\Theta = XS^{-1}$ is a diagonal matrix. The normal equations matrix $A\Theta A^T$ being symmetric and positive definite, its LL^T Cholesky decomposition may be formed. In IPM, this is the usual means of solving directly for $\boldsymbol{\Delta y}$ and hence, by reversing the elimination process, $\boldsymbol{\Delta x}$ and $\boldsymbol{\Delta s}$. However, the density of $A\Theta A^T$ may be significantly higher than A, and the density of L may be higher still. For some large LP problems, the memory required to store L may be prohibitive. Following [8] test problems which exhibit this behaviour are given in Table 1. The

	Dimensions			IPM		Simplex
Problem	rows	columns	nonzeros	Cholesky nonzeros	time	time
nug20	15240	72600	304800	38×10^6	1034 s	79451 s
nug30	52260	379350	1567800	459×10^6	OoM	>28 days
1kx1k0	1025	1025	34817	0.5×10^6	0.82 s	0.38 s
4kx4k0	4097	4097	270337	8×10^6	89 s	11 s
16kx16k0	16385	16385	2129921	128×10^6	2351 s	924 s
64kx64k0	65537	65537	16908289	2048×10^6	OoM	111 h

Table 1: Prohibitive cost of solving larger QAP problems and Gruca problems using Cplex 11.0.1 IPM and dual simplex

first two problems are larger instances of quadratic assignment problems (QAP) whose solution is one of the great challenges of combinatorial optimization and the remaining problems are quantum physics models provided to us by Jacek Gruca. As problem size increases, the memory requirement of the Cholesky decomposition prevents them from being solved via standard IPM and the simplex method is seen not to be a viable alternative.

2.1 Matrix-free IPM

For some LP problems the constraint matrix may not be known explicitly due to its size or the nature of the model, but it may be possible to evaluate $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. In other problems there may be very much more efficient means of obtaining these results than evaluating them as matrix-vector products. For such problems, Gondzio [7] is developing *matrix-free* IPM techniques in which systems of equations are solved by iterative methods using only the results of $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. However, the LPs of interest that can be solved at present are those for which A is known explicitly but solution via standard IPM and the simplex method is not possible, such as the LPs in Table 1.

Since the normal equations matrix $A\Theta A^T$ is symmetric and positive definite, the method of conjugate gradients can, in theory, be applied. However, its convergence rate depends on the ratio between the largest and smallest eigenvalues of $A\Theta A^T$, as well as the clustering of its eigenvalues [9]. Recall that since there will be many indices j for which only one of x_j and s_j goes to zero as the optimal solution is approached, there will be a very large range of values in Θ. This ill-conditioning means that conjugate gradients is unlikely to converge. Within matrix-free IPM, the ill-conditioning of $A\Theta A^T$ is addressed in two ways: by modifying the standard IPM technique and by preconditioning the resulting normal equations coefficient matrix.

The technique of IPM regularization adds terms to the original primal-dual pair of LP problems (1) to give

$$\begin{array}{llll} & \text{Primal} & & \text{Dual} \\ \min & \boldsymbol{c}^T\boldsymbol{x} + \dfrac{1}{2}(\boldsymbol{x}-\boldsymbol{x}_0)^T R_p(\boldsymbol{x}-\boldsymbol{x}_0) & \max & \boldsymbol{b}^T\boldsymbol{y} + \dfrac{1}{2}(\boldsymbol{y}-\boldsymbol{y}_0)^T R_d(\boldsymbol{y}-\boldsymbol{y}_0) \\ \text{s. t.} & A\boldsymbol{x} = \boldsymbol{b} & \text{s. t.} & A^T\boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c} \\ & \boldsymbol{x} \geq \boldsymbol{0} & & \boldsymbol{y} \text{ free}, \boldsymbol{s} \geq \boldsymbol{0}, \end{array} \tag{5}$$

where the proximal terms are given by positive definite diagonal matrices R_p and R_d, and reference points $\boldsymbol{x}_0$ and $\boldsymbol{y}_0$, all of which can be chosen dynamically. Although this transforms the original LP into a quadratic programming (QP) problem, the theory and techniques of IPM for QP problems are well established and, after an elimination process similar to that applied to (3), the normal equations system corresponding to (5) has the coefficient matrix

$$G_R = A(\Theta^{-1} + R_p)^{-1}A^T + R_d, \tag{6}$$

in which R_p guarantees an upper bound on the largest eigenvalue of G_R and R_d guarantees that the spectrum of G_R is bounded away from zero. Consequently, for appropriate R_p and R_d the condition number of G_R is bounded regardless of the conditioning of Θ.

The convergence properties of the conjugate gradient method are improved by using the partial Cholesky decomposition

$$G_R = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D_L & \\ & S \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ & I \end{bmatrix} \tag{7}$$

to motivate a preconditioner

$$P = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D_L & \\ & D_S \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ & I \end{bmatrix}, \tag{8}$$

where D_S consists of the diagonal entries of S. The number of nontrivial columns in the preconditioner is $k \ll m$ so, since only the diagonal entries of S are ever computed, the preconditioner is vastly cheaper to compute, store and apply than the complete Cholesky decomposition. Each iteration of the preconditioned conjugate gradient (PCG) method requires one operation with both P^{-1} and G_R. Since D_L, D_S, Θ, R_p and R_d are all diagonal matrices, the major computational costs are the operations with the nontrivial columns of P and the matrix-vector products with A and A^T. It is seen in Table 2 that the cost of PCG dominates the cost of solving the LP problem, and that PCG is dominated by the cost of operating with P^{-1} and calculating $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. For the QAP problems the cost of applying the preconditioner is greater but, for the quantum physics problems, the cost of the matrix-vector products increasingly dominates the solution time for the LP problem. Section 3 considers how the calculation of $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$ may be accelerated by exploiting a many-core GPU and multi-core CPU.

	Percentage of solution time			
Problem	PCG	P^{-1}	$A\boldsymbol{x}$	$A^T\boldsymbol{y}$
nug20	89	55	17	15
nug30	90	54	18	17
1kx1k0	62	41	12	11
4kx4k0	89	42	19	28
16kx16k0	87	30	30	29
64kx64k0	87	19	37	34

Table 2: Proportion of solution time accounted for by preconditioned conjugate gradients, operations with P^{-1} and calculations of $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$

3 Accelerating sparse matrix-vector products

The acceleration of the following three core operations

$$\boldsymbol{y} = A\boldsymbol{x} \qquad (\text{FSAX}), \qquad \boldsymbol{x} = A^T\boldsymbol{y} \qquad (\text{FSATY}), \qquad \boldsymbol{z} = (A\Theta A^T)\boldsymbol{y} \qquad (\text{FSAAT}) \tag{9}$$

is considered for the sparse matrix $A \in I\!R^{m\times n}$, diagonal matrix $\Theta \in I\!R^{n\times n}$ and dense vectors $\boldsymbol{x} \in I\!R^n$, $\boldsymbol{y} \in I\!R^m$ and $\boldsymbol{z} \in I\!R^m$. If A is held both row-wise and column-wise, all of these operations parallelise trivially. However double storage is expensive when A is large.

3.1 Operations

Assuming that A is held column-wise, FSAX can be considered as

$$\boldsymbol{y} = \sum_{j=1}^{n} \boldsymbol{a}_j y_j, \tag{10}$$

where $\boldsymbol{a}_j$ is column j of A, and the sum divided in parallel. The reduction of partial vector sums required is praticable in multi-core (requiring $\mathcal{O}(tm)$ temporary storage for t threads) but not in a many-core environment. To avoid this, a GPU implementation can form and store a row-wise copy explicitly. This is a high one-off cost in processing effort, and an ongoing doubling of device memory requirements. The operation with the transpose, FSATY, can be calculated independently column-wise as

$$x_j = \boldsymbol{a}_j^T\boldsymbol{y}, \tag{11}$$

for $j = 1, \ldots, n$. Algebraically, FSAAT may be expressed as the compound operation consisting of FSATY, a scaling by the diagonal entries of Θ, and then FSAX. However, significant gains can be realised by integration. In the CPU case, a trick [16] may be adapted to allow each column to be read from memory only once (cache sizes permitting) by calculating

$$\boldsymbol{z} = \sum_{j=1}^{n} \boldsymbol{a}_j \theta_j (\boldsymbol{a}_j^T\boldsymbol{y}), \tag{12}$$

where θ_j for $j = 1, \ldots, n$ are the diagonal entries of Θ. In the GPU case, the vector reduction required again makes this particular algorithm impractical so it is performed as the compound operation with transfers to and from the device being reduced when the scaling is also done on the GPU.

3.2 Kernels

In the serial CPU case, the optimizations that can be applied are well known [1, 6, 10, 11] and can be briefly summarised as

- Loop unrolling to exploit the deep pipelines and multiple execution units of current superscalar architectures.
- Vectorisation using SIMD instructions to improve throughput.
- Data blocking to improve cache efficiency.

In the classes of problems considered, the last is unfortunately not possible: these matrices are not known to contain dense subblocks. Additionally, vectorising sparse arithmetic efficiently is difficult, and results are reported for non-vectorised code since this continues to be the fastest serial implementation obtained.

In the case of parallel multi-core codes, the same optimization strategies and limitations apply. Parallelisation itself is largely trivial since synchronisation overheads are overshadowed by the high cost of multiplication. Less clear is under what circumstances memory bandwidth will scale to permit speed-up, the sparse matrix vector product (SpMV) being expected to be memory bound.

A GPU is not ideally suited to sparse arithmetic, as memory performance depends critically on regular (coalesced) access to data. There has been considerable work in this area [3, 5, 14] which has identified key optimizations as:

- Reordering data for coalescing - e.g. ELLPACK [3]
- Selecting an appropriate degree of parallelism to achieve memory coalescing and to neither over- nor under-exploit the device - e.g. CSR (vector) uses a warp per row [3], ELLR-T considers fitting the ideal number of threads per row. [14]
- Identifying problem rows for special treatment - e.g. the HYB kernel [3].
- Data blocking - e.g. BCSR, BELLPACK [5], ELLR-T [14].

As for the CPU case, data blocking does not appear to be applicable to the problems examined, from quadratic assignment and quantum physics, which have at most one dense row and column. All other rows and columns have the same, small, number of nonzeroes. For such regular data, CSR is unlikely to be an optimal storage format.

The first optimization is to extract the dense rows to be treated separately. An entire block is applied to these rows, the largest unit possible per row without multiple kernel launches per matrix multiply. This is similar to the treatment of long rows in the HYB kernel of [3].

ELLPACK allows full coalescing at one thread per row, but a single thread per row underutilises the test device (a Tesla C2070). Some care allows eight threads per row, where eight warps each calculate a partial sum for thirty two rows, and these are aggregated. This improves performance but is a ceiling on the amount parallelism that can be brought to bear per row. It is beneficial for large problems to use Transpose ELLPACK, which allows multiple warps to work on each row.

3.3 Results

The following results are obtained from a test system having two AMD Opteron 2378 (Shanghai) quad-core processors, 16 GiB of RAM and a Tesla C2070 GPU with 6 GiB of RAM. Note that the processors are relatively slow in serial, though the NUMA configuration of the memory bus gives high parallel memory performance. The GPU is a significantly more highly powered unit, making raw speed characterisations of less interest than the potential for improvement with a given investment.

The GPU kernel used is DHTELL2 which consists of one block to handle any dense rows separately, then a half warp (sixteen threads) per row using a transposed ELLPACK format to store the remainder of the matrix.

	Solve time (s)			SpMV time (s)		
Problem	Serial	8 core	GPU	Serial	8 core	GPU
nug20	2.19	1.18	1.60	1.49	0.495	0.945
nug30	20.5	15.8	15.4	15.1	9.69	9.45
1kx1k0	0.244	0.177	0.217	0.0360	0.0128	0.0506
4kx4k0	3.03	2.06	2.15	1.06	0.218	0.336
16kx16k0	24.9	18.4	13.5	13.1	6.70	1.72
64kx64k0	170.0	109.0	74.0	115.0	47.4	12.2
96kx128-0	137.0	71.1	58.8	93.4	28.6	15.3
256x256-0	866.0	283.0	222.0	699.0	119.0	56.4

Table 3: Comparison of accelerated matrix-free IPM codes.

Speed-up of sparse matrix-vector kernels using all 8 cores of the test system is between two and six times, giving at most a threefold speed-up of the IPM solution time. Using the high powered GPU, speed-up of these same kernels can approach ten times, though overall solution time is reduced no more than four times. Clearly significant speed-up of matrix-free interior point, whether by many-core or multi-core parallelism, is possible.

4 Conclusions

The matrix-free approach shows promise in making some of the most difficult classes of problem tractable by interior point methods. Its focus on a small core of sparse operations makes highly optimized implementations using state of the art hardware possible without excessive difficulty.

The particular choice of many-core or multi-core acceleration depends on the hardware available. As has been noted elsewhere [15], a GPU can provide performance essentially equivalent to a small number of multi-core processors in the context of sparse problems.

References

[1] Advanced Micro Devices, Inc., *Softare Optimization Guide for AMD Family 10h and 12h Processors*, 2011.

[2] G. Al-Jeiroudi, J. Gondzio, and J. Hall, *Preconditioning indefinite systems in interior point methods for large scale linear optimization*, Optimization Methods and Software, 23 (2008), pp. 345–363.

[3] N. Bell and M. Garland, *Efficient sparse matrix-vector multiplication on CUDA*, Tech. Rep. NVR-2008-004, NVIDIA Corporation, 2008.

[4] L. Bergamaschi, J. Gondzio, and G. Zilli, *Preconditioning indefinite systems in interior point methods for optimization*, Computational Optimization and Applications, 28 (2004), pp. 149–171.

[5] J. W. Choi, A. Singh, and R. W. Vuduc, *Model-driven autotuning of sparse matrix-vector multiply on GPUs*, in Proceedings of the 15th ACM SIGPLAN Symposium on principles and practice of parallel programming, ACM, 2010, pp. 115–126.

[6] M. Chuvelev and S. Kazakov, *High performance dense linear algebra in Intel Math Kernel Library*. International Conference Dedicated to the 100th Anniversary of the Birthday of Sergei L. Sobolev, 2008.

[7] J. Gondzio, *Matrix-free interior point method*, Computational Optimization and Applications, (2010). Published online October 14, 2010: DOI 10.1007/s10589-010-9361-3.

[8] ——, *Interior point methods 25 years later*, Tech. Rep. ERGO-11-003, School of Mathematics, University of Edinburgh, King's Buildings, Edinburgh, 2011.

[9] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Res. Natl. Bur. Stand, 49 (1952), pp. 409–436.

[10] E.-J. Im and K. Yelick, *Optimizing sparse matrix computations for register reuse in SPARSITY*, in Proceedings of the International Conference on Computational Science, volume 2073 of LNCS, Springer, 2001, pp. 127–136.

[11] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2009.

[12] N. K. Karmarkar, *A new polynomial–time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373–395.

[13] A. R. L. Oliveira and D. C. Sorensen, *A new class of preconditioners for large-scale linear systems from interior point methods for linear programming*, Linear Algebra and its Applications, 394 (2005), pp. 1–24.

[14] F. Vázquez, G. Ortega, J. Fernández, and E. Garzón, *Improving the performance of the sparse matrix vector product with GPUs*, in 2010 10th IEEE Conference on Computer and Information Technology (CIT 2010), 2010, pp. 1146–1151.

[15] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringapure, *On the limits of GPU acceleration*, in Proceedings of the 2nd USENIX Conference on Hot topics in parallelism, USENIX Association, 2010.

[16] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick, *Memory hierarchy optimizations and performance bounds for sparse A^TAx*, Tech. Rep. UCB/CSD-03-1232, EECS Department, University of California, Berkeley, 2003.

[17] S. J. Wright, *Primal-Dual Interior-Point Methods*, SIAM, Philadelphia, 1997.