Published in the Proceedings of the 1st International Symposium and 10th Balkan Conference on Operational Research (BALCOR 2011), 22-25 September 2011, Thessaloniki, Greece, which should be cited to refer to this work.

A multi-GPU implementation and performance model for the standard simplex method

Xavier Meyer, Paul Albuquerque
inIT, University of Applied Sciences
Geneva, Switzerland
paul.albuquerque,xavier.meyer@hesge.ch

Bastien Chopard
University of Geneva
Geneva, Switzerland
bastien.chopard@unige.ch

Abstract

The standard simplex method is a well-known optimization algorithm for solving linear programming models in operations research. It is part of software often employed by businesses for solving scheduling or assignment problems. But their always increasing complexity and size drives the demand for more computational power. In the past few years, GPUs have gained a lot of popularity as they offer an opportunity to accelerate many algorithms. In this paper we present a mono and a multi-GPU implementation of the standard simplex method, which is based on CUDA. Measurements show that it outperforms the *CLP* solver on large enough problems. We also derive a performance model and establish its accurateness. To our knowledge, only the revised simplex method has so far been implemented on a GPU.

1 Introduction

The simplex method is a well-known optimization algorithm for solving linear programming (LP) models in operations research. It is part of software often employed by businesses for finding solutions to problems such as airline scheduling problems. The original standard simplex method was proposed by Dantzig in 1947 [1]. A more efficient method named the revised simplex, was later developed. Nowadays its sequential implementation can be found in almost all commercial LP solvers. But the always increasing complexity and size of LP problems from the industry, drives the demand for more computational power. Indeed, current implementations of the revised simplex strive to produce the expected results, if any. In this context, parallelisation is the natural idea to investigate [6]. Already in 1996, Thomadakis et al. [10] implemented the standard method on a massively parallel computer and obtained an increase in performances when solving dense or large problems.

A few years back, in order to meet the demand for processing power, graphics card vendors made their graphical processing units (GPU) available for general-purpose computing. Since then GPUs have gained a lot of popularity as they offer an opportunity to accelerate algorithms having an architecture well-adapted to the GPU model. The simplex method falls into this category. Indeed, GPUs exhibit a massively parallel architecture optimized for matrix processing. To our knowledge, only the revised simplex method has so far been implemented on a GPU [3], showing encouraging results on small to mid-size LP problems.

In this paper, we present a single and a multi-GPU implementation of the standard simplex method, based on the CUDA technology of NVIDIA. We also derive the associated performance model and establish its accurateness. There are many technicalities involved in CUDA programming, in particular regarding the management of tasks and memories on the GPU. Thus fine tuning is indispensable to avoid a breakdown on performance. With respect to the algorithm, special attention must be given to numerical stability.

The paper is organized as follows. First, we give a short description of the standard simplex method, also introducing the heuristics we used. Then we cover the basics of GPU architecture. Next, we focus on the single GPU implementation and then explain how we split the LP model

in the multiple GPU version. Two sections are then devoted to describing performance models and comparing our implementations with the *CLP* solver. Finally, we summarize the results and consider new perspectives. Note that a more detailed account of this work can be found in [7].

2 The standard simplex method

The simplex method [1] applies to linear programming models. For the sake of simplicity, let us consider the canonical form of such problems :

$$\begin{array}{ll} \text{maximize} & z = \mathbf{c}^{\mathbf{T}} \cdot \mathbf{x} \\ \text{subject to} & \mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}, \quad \mathbf{x} \geq \mathbf{0} \end{array} \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{A}$ is the $m \times n$ constraint matrix. The objective function $z = \mathbf{c}^{\mathbf{T}} \cdot \mathbf{x}$ is the product of the cost vector $\mathbf{c}$ and the unknown variables $\mathbf{x}$. An element $\mathbf{x}$ is called a solution, which is said to be a feasible if it satisfies the linear constraints imposed by $\mathbf{A}$ and the bound $\mathbf{b}$.

The first step of the simplex method is to reformulate the model. So called *slack variables* x_{n+i} are added to the canonical form in order to replace the inequalities by equalities :

$$x_{n+i} = b_i - \sum_{j=1}^{n} A_{ij} x_j \quad (i = 1, 2, ..., m) \tag{2}$$

The resulting problem is called the *augmented form* in which the variables are divided into two disjoint sets: basic and nonbasic variables. Basic variables, which form the basis of the problem, are on the r.h.s. of the equations; nonbasic variables, which form the core, appear on the l.h.s.

The simplex algorithm searches for the optimal solution through an iterative process. A typical iteration consists of three operations : selecting the entering variable, choosing the leaving variable and pivoting. The entering variable is a nonbasic variable whose increase will lead to an augmentation of the objective value z. This variable must be selected with care in order to yield a substantial leap towards the optimal solution. The leaving variable is the basic variable which first violates its constraint as the entering variable increases. The choice of the leaving variable must guarantee that the solution remains feasible. Once both variables are defined, the pivoting operation makes these variables switch sets: the entering variable enters the basis, taking the place of the leaving variable which becomes nonbasic.

However, specific heuristics or methods are needed in order to improve the performance and stability of this algorithm. Our implementations use the *two-phase simplex* [1] to get hold of an initial feasible solution. Various methods for choosing the entering variable offer a compromise between computation cost and the number of iterations needed to solve the problem. Among these, the *steepest edge* [9] greatly reduces the number of iterations required. Moreover, its implementation fully benefits from the special architecture of a GPU. Stability and robustness of the algorithm depend considerably on the choice of the leaving variable. With respect to this, the *expand* method [5] proves to be very useful, particularly in helping to avoid cycles.

3 GPU architecture : CUDA

Since our programs are designed to run on a given type of GPU, we will focus on the parallel computing architecture developed by NVIDIA: Compute Unified Device Architecture (CUDA [8]). This architecture falls into the *Single Instruction Multiple Data* (SIMD) category. However, it is classified by NVIDIA under the concept of *Single Instruction Multiple Threads* (SIMT). The

key feature of this architecture is to exploit data parallelism and to offer fast context switching for the threads. When used to process large matrices, recent generation GPUs exhibit TFLOPS performances on single precision operations.

A GPU appears as a device, usually connected to the motherboard via the PCI bus, which is controlled by the CPU. The global memory of the GPU is used to share data between both processing units. Moreover, it is possible to have several GPUs in a computer. In this case, each GPU must be managed by its own CPU thread. In a standard application, the CPU manages the data and the global algorithm, while the GPU does the computing, as in a master-slave paradigm. Due to the nature of the GPU architecture, conditional branching instructions are usually delegated to the CPU to avoid sequential execution of the GPU threads.

The GPU architecture is organized around two important notions : work decomposition and memory levels. Work decomposition, describes how a computation can be decomposed using multiple threads. The parallel processing is related to the architecture of the computing units on a GPU. The latter contains several *streaming multiprocessors* (SM). Each of them has a fixed number of processing units that run instructions in parallel from a single instruction unit. Thus, each thread on a multiprocessor must execute the same instruction to ensure true parallelism.

The hierarchy of processing units leads to decomposing a computation over a multitude of threads, each of them doing mainly the same work. Consider for example the case of a matrix addition: $\mathbf{A}=\mathbf{B}+\mathbf{C}$. Each thread could load a single element B_{ij} and C_{ij} from the matrices, sum both values and store the result into A_{ij}. Threads are grouped into blocks which will be dispatched to the SMs. Inside a block, threads are grouped into so-called *warps*. Blocks are further organized into a grid of blocks, so as to provide a unique ID for each thread. The ID of a thread consists of the position of the thread in its block and the position of the latter on the grid. This ID is of crucial importance to access individual data during the computations.

There are different memory levels. A thread has its own registers on which the instructions are applied. Data accesses on registers are the fastest in this architecture. However, registers are shared inside a block, thus leading to a limited number of registers per thread. At another level, each block has its own shared memory accessible only by its own threads. This memory is nearly as fast as the registers when correctly accessed. More importantly, shared memory is the safest and fastest way to communicate between threads. Since SMs are not synchronized instruction-wise, threads on different blocks cannot communicate directly. Finally, the main memory of the GPU (GRAM) is a global memory. It is within the scope of all threads as well as the CPU. But access to this memory is costly since, in addition to the read/write instruction, it takes 400-600 cycles before the data is available from the GRAM.

A GPU architecture uses pipelines to improve performances. There is a fixed depth (D) pipeline depending on the number of processors per SM, because a *warp* consists of D batches whose threads run concurrently on the SM. Another pipeline is used to hide the latency due to GRAM accesses. The scheduler swaps *warps* waiting for data, with *warps* ready to be executed. The more warps, and thus threads, per block, the more efficiently this latency can be hidden.

4 Implementations

4.1 Single GPU

The first step performed by our implementation is done sequentially by the CPU. It consists in reading a LP model from a standard MPS file and transforming it to its augmented form. This modified problem is then entirely loaded in the GRAM. The next steps are listed in algorithm 1.

Algorithm 1 Simplex method

Require: p_{GPU} {Data structure containing the problem on the GPU}
Ensure: $feasible, infeasible$ {Feasibility of the problem}
1: **while** exists(x_{in}) **do**
2: $x_{in} \leftarrow$ find_entering(p_{GPU}), $x_{out} \leftarrow$ find_leavingVar(x_{in}, p_{GPU}) {GPU}
3: **if** not exists(x_{out}) **return** $infeasible$
4: pivoting(x_{in}, x_{out},p_{GPU}) {GPU}
5: **end while**
6: **return** $feasible$

At each iteration, the CPU launches the *kernels*[1] and checks the results returned. The first *kernel* searches for the entering variable using the *steepest edge* method. The complexity of this method is $\mathcal{O}(mn)$ and benefits thoroughly from the massively parallel architecture of the GPU. This *kernel* uses the problem loaded in the GRAM and returns only the selected entering variable. For the *expand* method used for choosing the leaving variable, it is required to use 2 or 3 *kernels* because of the different phases of this heuristic. Each phase has complexity $\mathcal{O}(m)$ and returns only one result. Finally, the pivoting operation requires a small constant amount of GRAM accesses done by the CPU in a preliminary stage. Then the pivoting is done by a *kernel* available in CUBLAS, an optimized CUDA library for basic linear algebra. Once the final optimal solution is found, the CPU has to retrieve the problem data.

This algorithm ensures that the communications between CPU and GPU are minimized. Besides loading the $m \times n$ matrix representing the problem into the GRAM and retrieving it into the CPU memory at the end of the computation, all other communications take a small and constant amount of time and can thus be neglected when dealing with large problems.

The problem is stored in the GRAM in a way to optimize data access. GPUs are able to retrieve more than a single data in one instruction whenever addresses are coalesced[2]. For example, an SM is able to read single precision variables for up to 16 threads in a single instruction, if the addresses are coalesced. Such accesses substantially increase the GRAM throughput.

4.2 Multi-GPU

The multi-GPU implementation uses the same algorithm as in section 4.1. However, some communications have to be added so as to synchronize the GPUs. The communication pattern is dependent on the way our LP problem is scattered across the GPUs. We split the constraint matrix vertically (see figure 1) in order to minimize communications.

After reading and preparing the problem, the CPU creates CPU threads that will manage the GPUs. Then, each of them initializes its GPU and loads a sub-matrix in its GRAM. During an iteration, each CPU thread launches concurrently the *steepest edge kernel* to obtain a local entering variable. The first communication intervenes here: each GPU must share with the others its local result. They then reduce the local result to elect the same best entering variable. Due to the selected splitting, only the CPU thread that holds the entering variable, launches the *expand kernels* to select the leaving variable. The column of this variable, called *pivot column*, is needed by all the other GPUs to execute the pivoting step. Thus this column, along with the leaving variable, is broadcasted. Finally, each GPU prepares and launches the pivoting *kernel* with no further communications. Note that GPUs communicate together via the CPU threads.

[1] CUDA functions are called *kernels*.
[2] CUDA concept for successive data addresses.

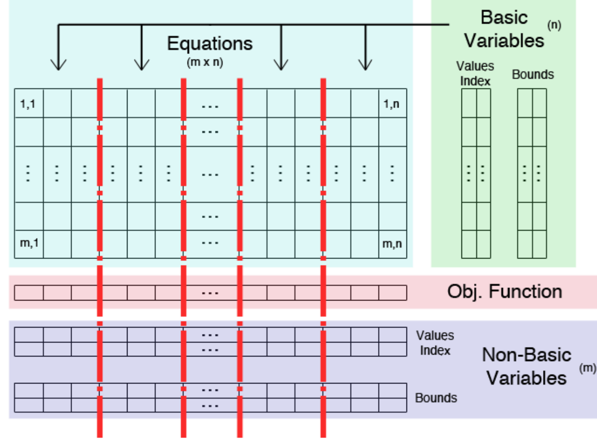

Figure 1: Vertical splitting of the LP model

5 Performance models

5.1 Modelling a *kernel*

CUDA *kernels* require a different kind of modelling than usually encountered in parallelism. Indeed, we have to take into account the GPU architecture described in section 3. The key idea is to capture in the model the decomposition of computations into threads, warps and blocks. One must also pay a particular attention to GRAM accesses and to how the pipelines reduce the associated latency. Our *kernel* model is inspired by [4].

The first step is to estimate the work done by a single thread of a *kernel* in terms of cycles. There are two different types of cycle. The first one is related to arithmetic instructions. Each instruction requires a different number of cycles in function of the operand and the operator. The second type is linked to the GRAM accesses. Since these accesses are non-blocking operation, they have to be counted apart from the arithmetic instructions.

The total work C_T done by a thread can be defined either as the sum or as the maximum of the memory access cycles and the arithmetic instructions cycles. Summing both types of cycles means we consider that latency cannot be used to hide arithmetic instructions. The maximum variant represents the opposite situation where arithmetic instructions and memory accesses are concurrent. Then only the biggest of the two represents the total work of a thread. This could occur for example when the latency is entirely hidden by the pipeline.

We must now find out how many threads are run by a processor. A *kernel* decomposes into blocks of threads. Thus each *streaming multiprocessor* (SM) must run N_B blocks. Each block itself has N_W *warps* consisting of N_T threads. An SM has N_P processors which execute batches of threads in a pipeline of depth D. Thus the total work C_{Proc} done by a processor is given by

$$C_{Proc} = N_B \cdot N_W \cdot N_T \cdot C_T \cdot \frac{1}{N_P \cdot D} \tag{3}$$

Finally, the execution time of a *kernel* is obtained by dividing C_{Proc} by the processor frequency.

5.2 A *kernel* example : *Steepest Edge*

The selection of the entering variable is done by the *steepest edge* method. This method requires the computation of the Euclidean norm of each column η_j of the constraint matrix. The coef-

ficients c_j of the variables in the objective function, are then divided by the norm $\|\eta_j\|$. The selected variable is the one having the smallest *steepest edge* ratio

$$se_l = \frac{c_l}{\|\eta_l\|} = \min_{j=1..n}\left(\frac{c_j}{\sqrt{1+\sum_{i=1}^{m} x_{ij}^2}}\right) \tag{4}$$

The processing of a column is done in a single block. Each thread of a block has to compute $N_{el} = \frac{m}{N_W \cdot N_T}$ coefficients of the column. This first computation consists of N_{el} multiplications and additions. The resulting partial sum of squared variables must then be reduced on a single thread of the block. This requires $N_{red} = \log_2(N_W \cdot N_T)$ additions. Since the shared memory is used optimally, there are no added communications. Finally, the coefficient c_j must be divided by the result of the reduction.

Each block of the *kernel* computes $N_{col} = \frac{n}{N_B \cdot N_{SM}}$ columns, where N_{SM} is the number of SMs per GPU. After processing a column, a block keeps only the minimum of its previously computed se_j ratios. Thus the number of arithmetic instruction cycles for a given thread is given by

$$C_{Arithm} = N_{col} \cdot (N_{el} \cdot (C_{add} + C_{mul}) + N_{red} \cdot C_{add} + C_{div} + C_{cmp}) \tag{5}$$

where c_{ins} denotes the number of cycles to execute instruction *ins*.

Each thread has to load N_{el} variables to compute its partial sum of squared variables. The thread computing the division also loads the coefficient c_j. This must be done for the N_{col} columns that a block has to deal with. We must also take into account that the scheduler hides some latency by swapping the *warps*, so the total latency $C_{latency}$ must be divided by the number of *warps* N_W. Thus the number of cycles relative to memory accesses is given by:

$$C_{Accesses} = \frac{N_{col} \cdot (N_{el} + 1) \cdot C_{latency}}{N_W} \tag{6}$$

At the end of the execution of this *kernel*, each block stores in the GRAM its local minimum se_l. The CPU will then have to retrieve the $N_B \cdot N_{SM}$ local minimums and reduce them. It is then profitable to minimize the number N_B of blocks per SM. With a maximum of two blocks per SM, the cost of this final CPU operation can be neglected when m and n are big.

We now take the maximimum or sum C_{Arithm} and $C_{Accesses}$ to obtain C_T. The result of equ. (3) divided by the processor frequency yields the time $T_{KSteepestEdge}$ of the *steepest edge kernel*.

5.3 Single and multi-GPU implementation models

As seen in section 4.1, the single GPU implementation requires only a few CPU-GPU communications. Since each of these communications are constant and small, they will be neglected in the model. For the sake of simplicity, we will consider the second phase of the *two-phase simplex* where we apply iteratively the three main operations: selecting the entering variable, choosing the leaving variable and pivoting. Each of these operations is computed as a *kernel*. The time for an iteration $T_{iteration}$ then amounts to the sum of all three *kernel* times

$$T_{iteration} = T_{KSteepestEdge} + T_{KExpand} + T_{KPivoting} \tag{7}$$

The times $T_{KExpand}$ and $T_{KPivoting}$ for the *expand* and pivoting *kernels* are obtained in a similar way as for the *steepest edge kernel* described in the previous section.

With the estimated time per iteration $T_{iteration}$, we can express the total time for solving a problem as $T_{prob} = T_{init} + r \cdot T_{iteration}$ where r is the number of iterations.

The model for a multi-GPU implementation is quite similar since each GPU basically executes the same *kernels* than the ones mentioned above. But there are two important differences: 1) the size of the local matrix processed by a GPU is $\frac{m \cdot n}{k}$, where k is the number of GPUs used; 2) we have to deal with the inter-GPU communications.

As seen in section 4.2, GPUs must share their locally computed entering variable so as to select one of them. Similarly the GPU choosing the leaving variable must share the *pivoting column* with the others. This depends on the number of equations m (also the size of a column) and the number of GPUs. The time of an iteration in the multi-GPU case is obtained by adding the communication times in equation (7).

6 Measurements

IBM CPLEX and *gurobi* are among the best performance proprietary solvers, whereas *GLPK*, *lpsolve* and *CLP* are well-known open source solvers. We chose *CLP* for our tests because of its excellent performances and its affiliation to the COIN-OR project. *CLP* is a sequential implementation of the revised simplex method.

We first successfully checked that our implementations were as functional as *CLP* by using the *NETLIB* [2] repository. This dataset usually serves as a benchmark for LP solvers. It has a vast variety of real and specific problems for testing stability and robustness of an implementation.

To test the performances of our implementations, a large range of problem size and density (percentage of non-zero coefficients) is needed. As none of the existing datasets provided the needed diversity of problems, we used a problem generator. It is then possible to create problems of specific size and density. Unfortunately, all the problems generated fall into the same category. Hence, generated problems of the same size and density will be solved within approximately the same number of iterations and the optimum of the objective function will only slightly fluctuate.

Our test environment is composed of a CPU server (2 Intel Xeon X5570, 2.93GHz, with 24GB DDR3 RAM) and a GPU computing system (*NVIDIA tesla S1070*). This system connects 4 GPUs to the server. Each GPU has 4GB GDDR3 graphics memory, 30 streaming multiprocessors, each holding 8 processors (1.4GHz). Such a GPU offers at peak performance up to 1TFLOPS for single precision floating point, 80GFLOPS for double precision floating point and 100GB/s memory bandwidth between threads (registers) and GRAM.

We measured the execution time for each implementation on problems of different size with density below 5%. As shown in figure 2, for large enough problems ($m > 750$ and $n > 3750$) our implementations outperform substantially *CLP*. A time limit of 30 minutes was imposed which explains why there are no measurements for *CLP* above $m = 1250$. However, on small problems *CLP* remains more efficient (see inset in fig. 2). This is mainly due to the initialisation time of a GPU which includes the time required to setup the *kernel* context.

Implementations of the revised simplex method, like *CLP*, are highly influenced by the problem density. The denser the problem, the slower *CLP* runs. Thus we mainly compared performances on problems of density lower than 5%. However, to take a deeper look into how these implementations behave under various densities, we measured the execution time on a fixed size problem (500×2000) but with increasing density. The execution time of our simplex implementation remained constant, while that of the *CLP* grew continuously. In this experiment *CLP* was outperformed when the problem density rose above 5% (see fig. 3). Moreover, we noticed that the problem density often increased significantly throughout a run.

Finally, we validated our performance models by computing the correlation between measurements and models for 1 to 4 GPUs, which was above 0.999. For big problems ($m > 5000$ and $n > 25000$), the speedup predicted by the model is nearly optimal in the range up to 40 GPUs.

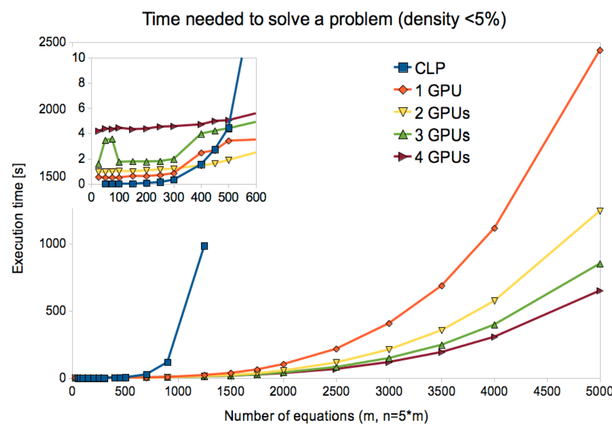

Figure 2: Implementations execution time as a function of the problem size

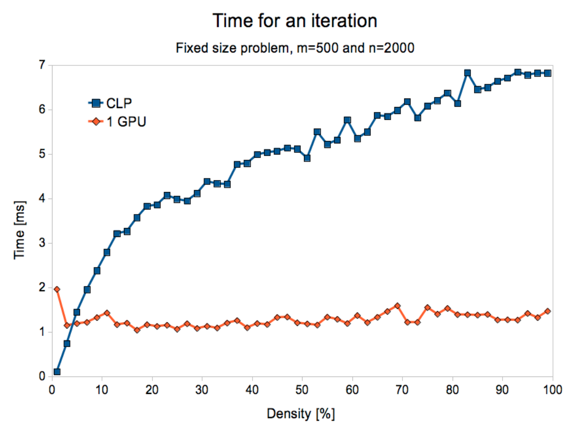

Figure 3: Time of an iteration as a function of the problem density

7 Conclusion

In this paper, we presented robust and efficient single and multi GPU implementations of the standard simplex method. These implementations outperformed a well-known open source linear programming solver, *CLP*, on mid-size to big and/or dense problems. Moreover, we derived accurate performance models for these implementations. These models actually predict a nearly optimal speedup while using up to 40 GPUs to solve large problems.

As GPUs are constantly evolving, we can expect for such implementations an improvement of performances in the near future. The latest GPU generation of NVIDIA is up to 5 times faster on double precision instructions than the one used in this work. Finally, we believe that interior point methods could also benefit from the massively parallel architecture of GPUs.

References

[1] V. Chvatal. *Linear Programming*. W.H. Freeman, 1983.

[2] J. Dongarra and E. Grosse. The NETLIB repository. http://www.netlib.org, 2010.

[3] J. Bieling et al. An efficient GPU implementation of the revised simplex method. In *Proc. of the 24th Int'l Symp. on Parallel and Distributed Processing*, pages 1–8. IEEE, 2010.

[4] K. Kothapalli et al. A performance prediction model for the CUDA GPGPU platform. Technical report, Int'l Inst. of Information Technology, Hyderabad, 2009.

[5] P.E. Gill et al. A pratical anti-cycling procedure for linearly constrained optimization. *Mathematical Programming*, 45:437–474, 1989.

[6] J.A.J. Hall. Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7:139–170, 2010.

[7] X. Meyer. Etude et implmentation de l'algorithme standard du simplexe sur GPUs. Master's thesis, University of Geneva, 2011.

[8] NVIDIA. *CUDA Compute Unified Device Architecture : Programming Guide*, 2008.

[9] A. Swietanowski. A new steepest edge approximation for the simplex method for linear programming. *Computat. Optimization and Appl.*, 10:271–281, 1998.

[10] M. Thomadakis and J.-C. Liu. An efficient steepest-edge simplex algorithm for SIMD computers. In *Proc. of the Int'l Conf. on Supercomputing, ICS'96*, pages 286–293. ACM, 1996.

View publication stats