Multimed Tools Appl (2018) 77:30035–30050
https://doi.org/10.1007/s11042-018-5947-z

Revised simplex algorithm for linear programming on GPUs with CUDA

Lili He[1,3] · Hongtao Bai[1,2,3] · Yu Jiang[1,3] ·
Dantong Ouyang[1,3] · Shanshan Jiang[1,3]

Received: 22 June 2017 / Revised: 9 October 2017 / Accepted: 26 March 2018 /
Published online: 18 April 2018

Abstract The revised simplex algorithm (RSA) is a typical algorithm for solving linear programming problems. Many theoretical modifications have been done to make the algorithm more efficient, but almost all of them were based on single-instruction single-data architecture processors (CPUs), which could not make full use of the inherent parallel characteristics of RSAs. We propose a novel single-instruction multiple-data architecture processor (GPU) based on the RSA in this paper. The intensive matrix manipulations of a traditional RSA are offloaded to the GPU, which helps to make full use of its powerful parallel processing ability. We implemented the GPU-based RSA on compute unified device architecture (CUDA). Numerical experiments on randomly generated linear programs show that the GPU-based RSA can not only find the correct optimal solutions, but can also reach a speed of up to 100 times as fast as that of a CPU-based RSA: it also runs 3 to 11 times as fast as MATLAB.

Keywords CUDA · GPU · Revised simplex algorithm · SIMD

✉ Hongtao Bai
baihongtao@263.net

✉ Yu Jiang
jiangyuyou@126.com

✉ Dantong Ouyang
ouyd@jlu.edu.cn

1 College of Computer Science and Technology, Jilin University, Changchun 130012, China

2 Center for Computer Fundamental Education, Jilin University, Changchun 130012, China

3 Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun 130012, China

1 Introduction

The simplex algorithm and its revised version, the revised simplex algorithm (RSA) [11, 30] for linear programming (LP) problems [4, 6, 9], are perhaps the most well-studied algorithms in the optimization literature because of their good performance in solving LP problems of small or medium size. Luh and Tsaih [17] proposed using the interior point method (IPM) to replace constructing the initial basis of the simplex; Paparrizos et al. [21] proposed a different approach that integrates the IPM and simplex methods. Junior and Lins [13] used a cosine criterion to obtain an improved initial basis. Those efforts all aimed at finding a better initial basis to reduce the number of iterations of the algorithm, not to speed the iterations themselves. Moreover, almost all those studies were based on single instruction stream, single data stream CPUs, which could not make full use of the inherent parallel characteristics of RSAs.

The appearance of programmable graphics processing units (GPUs) [3, 5, 12, 23, 24] and their wide availability on PCs make it possible to parallelize an RSA on a single PC. The GPUs have many powerful single-instruction multiple-data (SIMD) processors supporting parallel data processing and high-precision computation. The performance and precision of GPUs are constantly increasing.

A novel GPU-based RSA is proposed in this paper. To accelerate the intensive matrix manipulations of traditional RSAs, we offloaded key matrix computations of an RSA to the GPU to fully use the GPU multiprocessors. To decrease the data delay between the GPU and the GPU's memory, we designed optimized threads and blocks of NVIDIA's CUDA so the kernel could do intensive arithmetic calculations and conceal the latency. To reduce the data transfer overhead, we introduce 2 strategies. First, in this algorithm we made very large arrays reside in the global memory of the GPU without being downloaded from the GPU to the CPU. Second, only 2 small vectors were downloaded in the iteration in direct memory access (DMA) mode to minimize data traffic. To summarize, the GPU was responsible for intensive computations, whereas the CPU answered for iteration control and minimum choosing, which the GPU is not good at. As a result, a high-performance but low-cost GPU-based RSA was achieved.

We implemented this GPU-based RSA on the newest generation GPU with CUDA. Numerical experiments on randomly generated LPs show that the proposed GPU-based RSA can not only find the correct optimal solution, but also reaches speeds several hundred times as fast as a CPU-based RSA, and 3 to 11 times as fast as the latest MATLAB release.

This paper is organized as follows: Section 2 reviews the background of RSAs. Section 3 presents the concept of and related works on general-purpose computing on graphics processing units (GPGPU). Section 4 describes GPU-based RSAs and some implementation details. A performance analysis of our approach is reported in Section 5. Finally, conclusions are drawn in Section 6.

2 Revised simplex algorithms

An LP problem can be described by matrix notation as follows:

Find a vector $x \in R^n$ that will

$$\begin{aligned} &\max z = c^T x \\ &s.t. \begin{cases} Ax = b \\ x \geq 0 \end{cases} \end{aligned} \qquad \text{(LP problem)}$$

$A \in R^{m \times n}$, $b \in R^m$, and $c \in R^n$. The equations represent the constraints. The function z is the objective function.

The RSA starts from a known extreme point and proceeds to an adjacent extreme point. An extreme point is near another extreme point if the points have all but one basic variable in common. For the solution to "move" from one extreme point to its adjacent point, one basic variable has to be replaced by a nonbasic variable. The nonbasic variable that will cause the greatest immediate increase in the objective function's result is selected and is known as the entering variable. The basic variable to be replaced is known as the departing variable. The departing variable is selected so that it will decrease the objective function the least. This process continues until either an optimal solution is reached or the problem is determined to have no optimal solution.

An RSA consists of 7 steps:

STEP1: Translate the LP problem into a canonical form by using some slack or artificial variables so that an initialized base feasible solution is obtained by the formula

$$\begin{bmatrix} x_B \\ x_N \end{bmatrix} = \begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix} \tag{1}$$

where the initialized base matrix B would be an identity matrix I_m, as would B^{-1}. Afterward, the simplex multiplier $Y = {c_B}^T B^{-1}$ is calculated.

STEP2: Calculate $\delta_A = c_B^T B^{-1} A - c^T$. If $\delta_A \geq 0$, an optimal solution has been achieved, so stop; else, go to the next step.

STEP3: Determine the entering variable x_k by choosing the variable that will cause the greatest increase in the objective function. This is equivalent to selecting the k that gives the smallest $c_B^T B^{-1} A_k - c_k$.

STEP4: Calculate $B^{-1}A_k$. If $B^{-1}A_k \leq 0$, the problem does not have an optimal solution, so stop; else, go to the next step.

STEP5: Determine the departing variable x_l by choosing the variable with the smallest nonnegative $\theta - ratio$, $\theta = \min\left\{ \frac{(B^{-1}b)_i}{(B^{-1}A_k)_i} \middle| (B^{-1}A_k)\ _i > 0 \right\} = \frac{(B^{-1}b)_i}{(B^{-1}A_k)_i}$

STEP6: Determine the new B_{new}^{-1} from the previous B_{old}^{-1} and the departing and entering variables. The formulas are $B_{new}^{-1} = EB_{old}^{-1}$, $E = (e_1, \ldots, e_{l-1}, \xi, e_{l+1}, \ldots, e_m)$, $I_m = (e_1, e_2, \ldots, e_m)$, and

$$\xi = \begin{bmatrix} -a_{1k}/a_{lk} \\ -a_{2k}/a_{lk} \\ \vdots \\ 1/a_{lk} \\ \vdots \\ -a_{mk}/a_{lk} \end{bmatrix} \leftarrow l' th$$

STEP7: Determine the new x_B and Y, and then continue with STEP2.

Obviously, an RSA includes many matrix operations. On a CPU, the elements of a matrix must be computed one by one on one core, whereas on a GPU they can be computed simultaneously because SIMD processors can execute the same operations (instructions) on different data.

3 General-purpose GPUs

Viewed only by cost, GPUs are probably today's most powerful computational hardware. The rapid increase in the performance of graphics hardware, coupled with recent improvements in its programmability, have made graphics hardware a compelling platform for computationally demanding tasks in a wide variety of application domains. Many research projects and implementations based on GPUs, such as ocean surface simulation [28], image processing [14], computational biology [8, 15], and pattern recognition [10, 19] have been presented in recent years for general-purpose computing, an effort known collectively as GPGPU (general-purpose computing on GPUs).

Currently, the peak performance of state-of-the-art GPUs is approximately 10 times as fast as that of comparable CPUs. Furthermore, the growth rate of the number of transistors used on GPUs is greater than for microprocessors [20]. Consequently, GPUs will soon become an even more attractive alternative for high-performance computing.

However, before a new architecture's GPU became available, accessing all the computational power of the GPU and efficiently leveraging it for nongraphics applications remained tricky:

- The GPU could be programmed only through a graphics API, imposing a high learning curve on the novice and the overhead of an inadequate API on the nongraphics application.
- The GPU DRAM could be read in a general way—GPU programs can gather data elements from any part of DRAM—but could not be written in a general way.

CUDA is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API [25]. Compared to previous programming interfaces such as Cg, CUDA provides more flexibility to efficiently map a computing problem onto the hardware architecture. CUDA applications consist of 2 parts. The first executes on the GPU (the device) and is called a "kernel." Kernels are implemented in the CUDA programming language, which is basically the C programming language extended with several keywords. The second part executes on the host CPU (the host) and provides control over data transfers between the CPU and the GPU and the execution of kernels.

A kernel program is run by multiple threads that run on the GPU. We call a group of threads a block. Threads contained in the same block communicate with each other using shared memory and cannot communicate with threads in another block. Calculations on the GPU are started by specifying the number of blocks to execute and the number of threads that each block contains. The total number of threads is the product of the two.

For now, CUDA is available for the NVIDIA GeForce GTX series, the Tesla solutions, and some Quadro solutions. The NVIDIA GeForce GTX hardware architecture defines a hierarchical memory structure where each level has a different size, access restrictions, and access speed, as shown in Fig. 1 [26]. Generally, accessing the largest type of memory is flexible but slow, whereas accessing the smallest type of memory is restrictive but fast. This memory structure is directly exposed by the CUDA programming framework. The challenge in mapping a computing problem efficiently on the GPU through CUDA is to store frequently used data items in the fastest memory, while keeping as much of the data on the device as possible.

In a word, GPUs are especially well suited to address problems that can be expressed as data-parallel computations, where the same program is executed on many data elements in parallel, with high arithmetic intensity—the ratio of arithmetic operations to memory operations. Many applications that process large data sets such as arrays can use a data-parallel programming model to increase computation speed. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control; and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. In fact, during the short period of 1 year after CUDA appeared, many algorithms outside the field of image rendering and processing were accelerated by CUDA, from digital investigation [18] or physics simulation [2] to molecular dynamics [1, 16].

4 GPU-based RSAs

4.1 Overview of GPU-based RSAs

The main characteristic of a GPU-based RSA is that the data-parallel, compute-intensive portions of a traditional RSA are offloaded to the device as a series of CUDA kernels. The GPU-based RSA is shown in Fig. 2.

We did not offload the entire RSA to the device, because its transistors are mostly devoted to data processing, making it weak at flow control and data caching. In this algorithm, all matrix computations such as basic revised matrix B^{-1}, simplex multiplier Y, and the θ and δ sequence are put on the device, which can indisputably use the entire device. The host handles

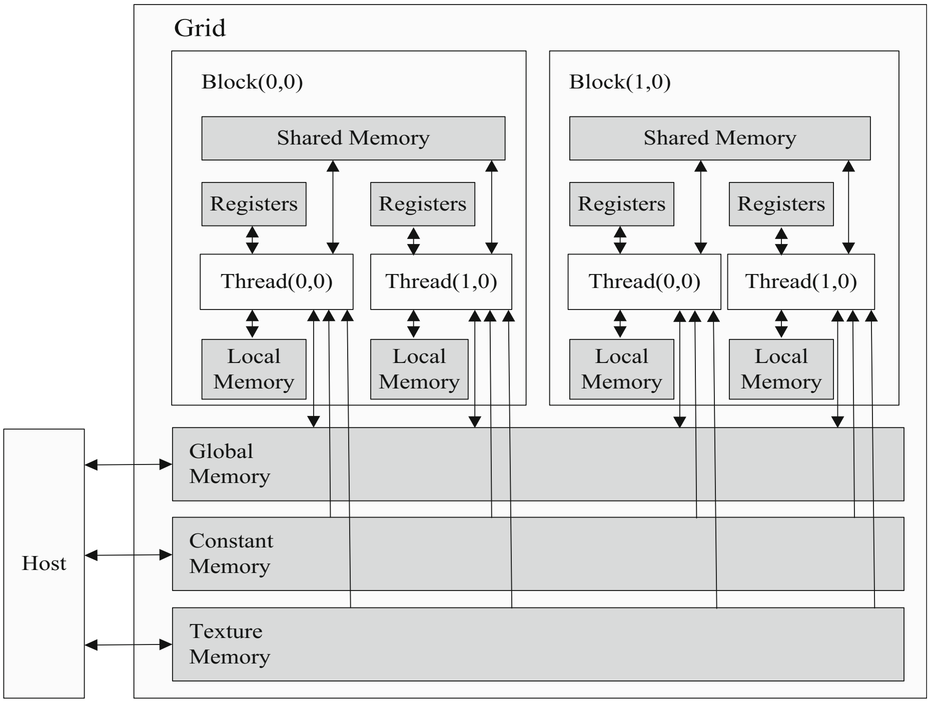


Fig. 1 Hardware architecture of NVIDIA GeForce GTX series

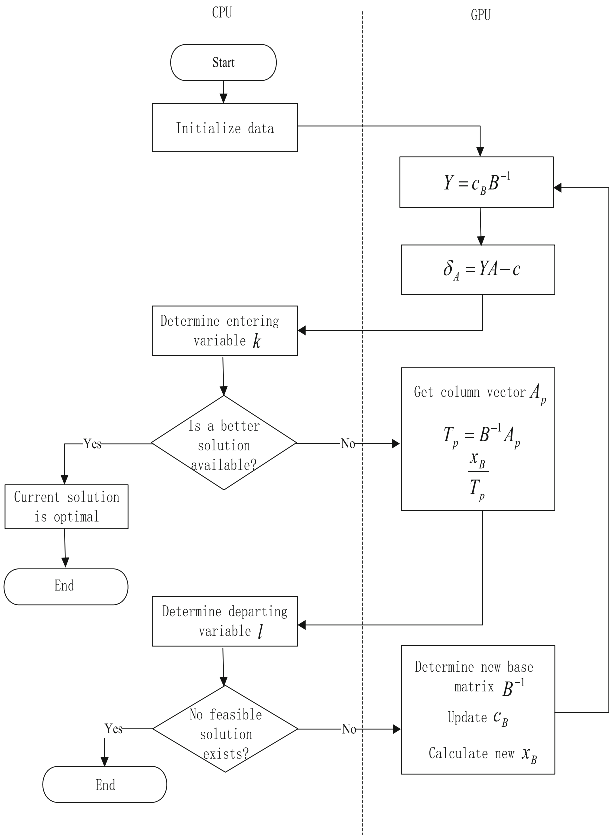

CPU
GPU
Start
Initialize data
$Y = c_B B^{-1}$
$\delta_A = YA - c$
Determine entering variable k
Is a better solution available?
Yes
No
Current solution is optimal
End
Get column vector A_p
$T_p = B^{-1} A_p$
$\frac{x_B}{T_p}$
Determine departing variable l
No feasible solution exists?
Yes
No
End
Determine new base matrix B^{-1}
Update c_B
Calculate new x_B

Fig. 2 The flowchart of the GPU-based RSA

both entering and departing variable choosing, which are not suitable for thread-level parallelism, and iteration control for judging whether an optimal solution is obtained or no feasible solution exists.

The data transfer between the host and device's memories is an extra cost to the GPU-based RSA compared to traditional RSAs. We strove to minimize data transfer between the host and the device. In the initialization phase, parameters of GPU-based RSA such as A, b, c are uploaded once from the host to the device. In the iteration phase, δ and θ are downloaded from the device to the host.

The data access speed between the GPU and its global memory is much slower than that between the CPU and its cache. Threads and blocks of CUDA are optimized for winning the device's power. The details are discussed in Section 4.2. The flow of the GPU-based RSA, which is applicable to not only CUDA but also other mainstream GPGPU environments such as data parallel virtual machines (DPVMs) [22], was determined by the architectures of the CPUs and GPUs.

4.2 Implementation details

The matrices in the GPU-based RSA include coefficient matrix A, resource vector b, value vector c, and other assistant structures such as E, B^{-1}, and x_B. All are stored as dynamic arrays on the device. The device has 3 kinds of memories shared by all the threads in one kernel: constant memory, texture memory, and global memory. We put all parameters in global memory because both constant memory and texture memory are read-only and 64 KB, which is insufficient to store data for large LPs. In the each iteration, the host issues a succession of kernel invocations to the device, namely Kernel2, Kernel3, and so on up to Kernel9. Each kernel is executed as a batch of threads organized by a grid of blocks to parallelize each formula shown in the right side of Fig. 2. According to the suggestion in the CUDA manual, we set blocks as 2-dimensional arrays of (D_x, D_y) by (16, 16) to give enough threads to every multiprocessor in the device. Thus, the number of threads per block, named *numth*, was $D_x{}^*D_y$, which equaled 256.

In the following 3 sections, we specify only 3 representative kernels: Kernel1, Kernel2, and Kernel3.

4.2.1 Kernel1

Both the host and the device maintain their own DRAM, referred to as the host memory and the device memory respectively. Any kernel can read from or write to only the device memory, so that data must reside in the device memory persistently in the iteration of the GPU-based RSA. We used 2 different approaches for data preparation. LP parameters input by the host were uploaded from the host memory to the device memory through optimized API calls that used the device's high-performance DMA engines, as the bottom left area of Fig. 3 shows. Other assistant structures were fulfilled by Kernel1. As shown in the bottom right area of Fig. 3, each thread of Kernel1 was responsible for one column of the $m \times m$ matrix and one pixel of the m vector. To make our GPU-based RSA manage arbitrary LP sizes and to satisfy the programming mode of the device, we expanded m to $\lceil m/numth \rceil * numth$ and n to $\lceil n/numth \rceil * numth$, which corresponded to adding extra slack variables. It worth maximizing the use of the available computing resources of the device by sacrificing some space, especially when an LP is very large. This expansion makes the device's memory controller handle data from 16× starting addresses, which can increase the device memory access speed. Another gain is that we did not have to judge the boundary.

Thus, we obtained all data in the device memory. Note that the "4-channel (R, G, B, A)"-based data model usually used in earlier GPUs was not required, because 1-D scalar quantity is optimized by CUDA.

4.2.2 Kernel2

Kernel2 was the core program for B^{-1} by $B^{-1}_{new} = EB^{-1}_{old}$. Every element λ'_{ij} of B^{-1}_{new} may be calculated by a thread based on $\sum\limits_{k=1}^{m} e_{ik}{}^*\lambda_{kj}$, as shown in Fig. 4, if B^{-1} and E are 2 general matrices. The computing for an element in the $i'th$ row and $j'th$ column of the product matrix requires only 2 vectors, the $i'th$ row of the left matrix and the $j'th$ column of the right matrix. Such matrix multiplication can be implemented much faster on GPUs with parallel massive threads than on CPUs. The CUDA manual gives 2 examples of matrix multiplication sped up by submatrix and shared memory. Nonetheless, that was written for clarity of exposition to illustrate various CUDA programming principles, not with the goal of providing a high-performance kernel for generic matrix multiplication. In this paper, matrix multiplication is not constructed that way.

We notice that E is a unit matrix, except for lth column, and l is the sequence number of the entering variable. As a result, we give a revised formula for B_{new}^{-1} below, if $E=(e_1,\ldots e_{l-1},\xi,e_{l+1},\ldots,e_m)$, $\xi=(\xi_1,\xi_2,\ldots,\xi_m)^T$, $B_{old}^{-1}=\left[\lambda_{ij}\right]_{m\times m}$, and $B_{new}^{-1}=\left[\lambda_{ij}^{'}\right]_{m\times m}$:

$$\lambda_{ij}^{'}=\left\{\begin{array}{ll}\xi_i\times\lambda_{lj}+\lambda_{ij} & i\neq l\\ \xi_i\times\lambda_{lj} & i\neq l\end{array}\right\} \tag{2}$$

Based on Eq. (2), we implement Kernel2 for computing B_{new}^{-1}. As schematically illustrated in Fig. 5, a thread produces a whole column of B_{new}^{-1}, as opposed to the one-element mode of Fig. 4, to maintain intensive computation and avoid too frequent switches to other threads. The total number of threads of Kernel2 is m under this mode. ξ is copied from global memory into the shared memory of every multiprocessor by threads within one block using a synchronization mechanism to speed memory access. A shared memory capacity of 16 KB in every multiprocessor can support an m of not more than 4000.

Early GPUs have some inconveniences for general-purpose computing, owing to their low-level mechanism of graphics API used for programming. One of them is that input and output regions, usually as textures, cannot be the same to a shader (equivalent to a kernel in this paper). To solve that problem, ping-pong and texture ID swap techniques are commonly used. Fortunately, CUDA supports reading from and writing to the same arrays in global memory without extra cost. Thus, we can use a single B^{-1} array as long as Kernel2 updates $\lambda_{ij}^{'}$ last for each column.

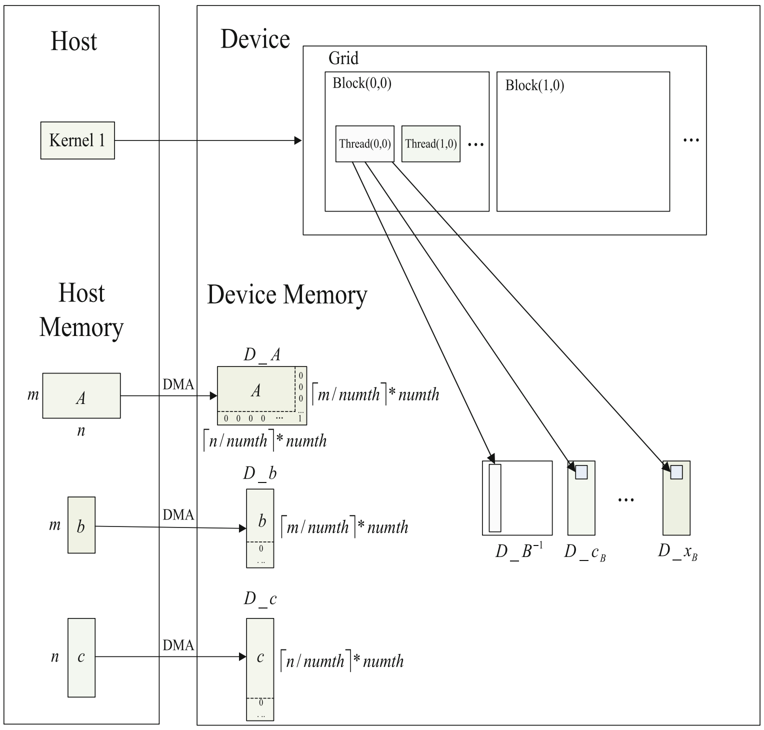


Fig. 3 Data initialization in the device memory

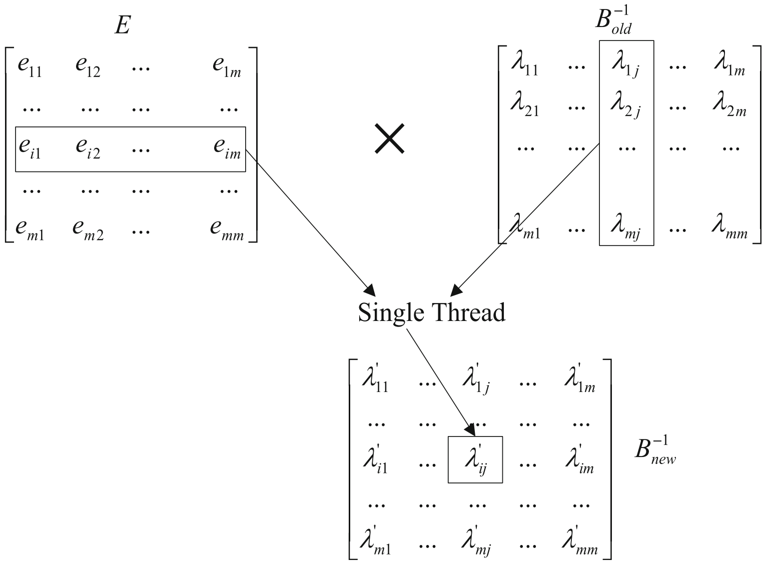


Fig. 4 Multiplication of 2 general matrices on the device

4.2.3 Kernel3

Kernel3 is the program for δ_A. Every thread is responsible for one element of δ_A, as Fig. 6 shows. $YA - C$, the mixture manipulation of multiplication and subtraction, integrated in one kernel proves that CUDA has greater programming flexibility and is more practical for general-purpose computation.

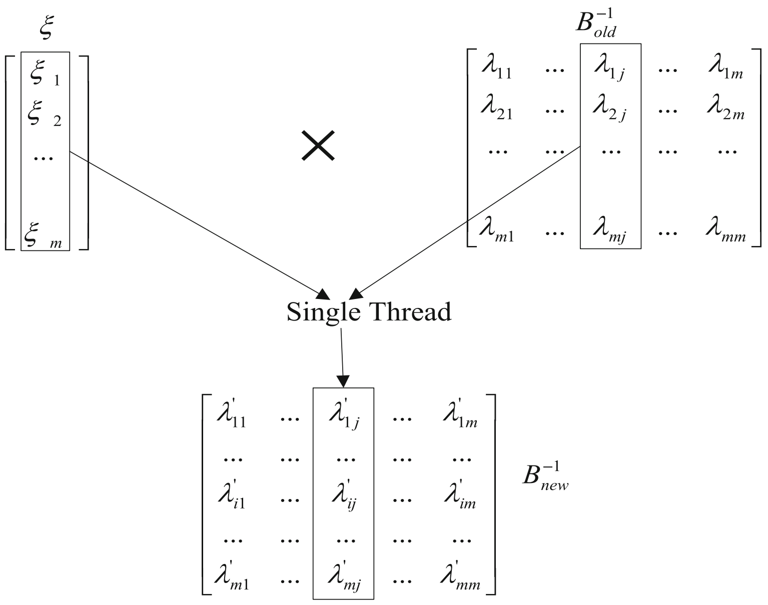


Fig. 5 ξB^{-1} on the device

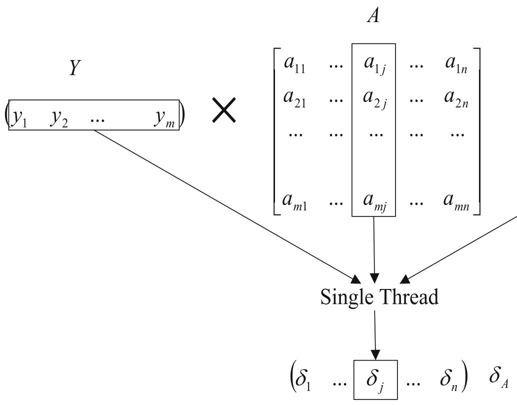


Fig. 6 δ_A on the device

5 Experiment results

In this section we compare GPU-based RSAs with CPU-based RSAs and MATLAB. All experiments were done on a Dell-compatible PC with an Intel Core i5–6500 CPU 3.2 GHz, 16 GB of main memory, a Geforce GTX 750 Ti graphics card, a 1.15 GHz engine clock speed, 2 GB of device RAM, and 640 stream processors organized into 20 multiprocessors.

5.1 Functionality test

At the time of our study, GPUs could support only single-precision binary floating-point arithmetic with some deviations, compared with CPUs. To validate the correctness of the GPU-based RSA, we produced samples consisting of random 32-bit floating-point numbers between 0 and 1. In the iterations, because CUDA follows the IEEE-754 standard, we simulated 64-bit floating-point operations using the Kahan summation formula to achieve double-type precision. The LP problem was stored in a text file in the standard form, with its size being the number of constraints m ($m < n$).

Table 1 GPU-based RSA correctness test

LP size (m)	MATLAB	CPU-based RSA		GPU-based RSA	
	Optimal Value	Iterations	Optimal Value	Iterations	Optimal Value
100	0.022	10	0.022	10	0.022
200	0.053	16	0.053	16	0.053
300	0.040	12	0.040	12	0.040
500	0.037	14	0.037	14	0.037
600	0.013	17	0.013	17	0.013
800	0.012	19	0.012	19	0.012
1000	0.006	10	0.006	10	0.006
1500	0.007	17	0.007	17	0.007
2000	0.008	18	0.008	18	0.008
3000	0.004	–	–	26	0.004
4000	/	–	–	24	0.008

[a] "/" indicates no result; the host was out of memory

[b] "-" indicates that the CPU-based RSA did not stop within 5 h (18,000 s)

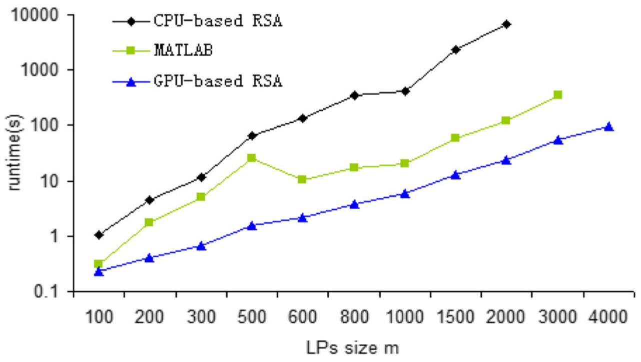


Fig. 7 Performance comparison between a CPU-based RSA, MATLAB, and the proposed GPU-based RSA

The optimal value and iterations are listed in Table 1. In the table, "/" indicates that MATLAB cannot get any result because the host is out of memory, and "-" indicates that the CPU-based RSA did not stop within 5 h (18,000 s).

From Table 1, we see that the 3 algorithms obtained the same optimal solutions at a certain precision: 3 valid digits after the decimal point. In addition, the GPU-based RSA maintained iterations identical to those of the CPU-based RSA. Iterations of MATLAB were not used as the benchmark because it can automatically adopt the best algorithm, not certain RSA-class algorithms, according to the LP's size and some other characteristics. For example, MATLAB will choose LIPSOL for large problems and a projective method for medium problems. Through the above comparisons, we found that our GPU-based RSA was effective, and GPUs may be applied to the optimization problem.

5.2 Performance test

Both the LP's size *m* and iterations will affect the total runtime. We produced 10 samples for each LP size *m*. The average runtime was used for performance comparison.

As shown in Fig. 7, the GPU-based RSA outperformed both the CPU-based RSA and the latest release of MATLAB. The GPU-based RSA ran up to 100 times as fast as the CPU-based RSA, and ran 3 to 11 times as fast as MATLAB under LP sizes of

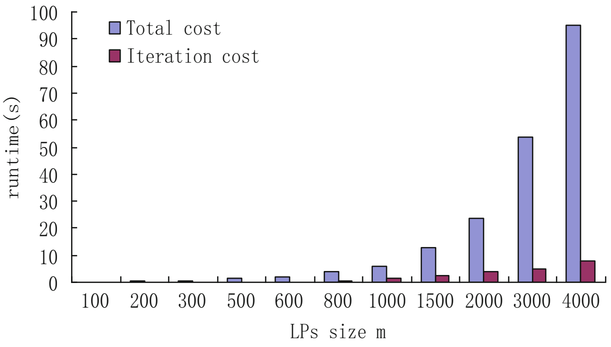


Fig. 8 Total cost compared with iteration cost of the GPU-based RSA

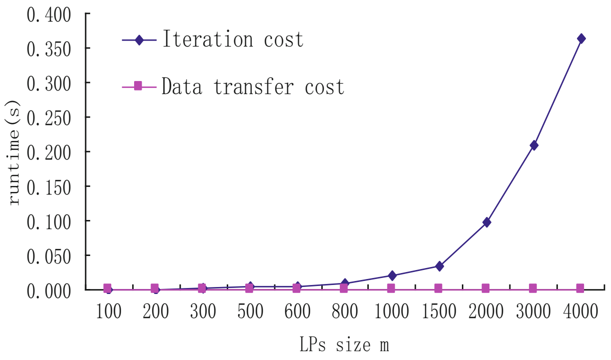


Fig. 9 Data transfer cost compared with iteration cost for one iteration of the GPU-based RSA

100 to 4000. In addition, when m reached 3000 the CPU-based RSA could not end in 5 h, and when m reached 4000 neither the CPU-based RSA nor MATLAB obtained an optimal solution, whereas the GPU-based RSA did, in under 100 s. At the same time, we note that MATLAB generally runs several dozens of times as fast as a CPU-based RSA, because it adopts just-in-time and vectorization technologies with far greater efficiency than does C or C++ hand coding.

From the workflow of the GPU-based RSA, we know that the device mainly participates in iteration; that is to say, the GPU-based RSA improves only computation performance. For example, in the work of loading LP parameters from a .txt file there are no differences between the GPU-based RSA and a CPU-based RSA, or even between the GPU-based RSA and MATLAB. The iteration cost of an RSA is usually most of the total cost. However, is it the same in our approach? We compared iteration cost with total cost of the GPU-based RSA in Fig. 8.

To our surprise, we conclude that this traditional rule did not apply: iteration cost was far lower than total cost. If we consider merely iteration cost, the GPU-based RSA would achieve a greater speed increase than a CPU-based RSA and MATLAB.

Data transfer is the extra cost of a GPU-based RSA. In each iteration, δ_A and θ must be downloaded once from the device memory to the host memory. We also compared the data transfer cost with the iteration cost for one iteration, as shown in Fig. 9.

Figure 9 shows that the data transfer cost of all samples was zero (0.000000 s), so may be completely ignored. The data transfer rate between the host memory and the device memory under PCI-E16x is approximately 4 GB/s, so it bears little influence on the total cost of the GPU-based RSA.

6 Conclusion

The sizes of the LPs that many practical applications and science computations must process are growing rapidly, and the current linear optimization tools are already struggling to deal with even modest-sized targets. In addition, the main body of every algorithm for solving LPs is matrix manipulation. That puts a severe strain on the computation resources of a single workstation. As a result, operation researchers must use every means available to increase performance. Some of the possible means include paying critical attention to designing more efficient software, developing software that can take advantage of modern multicore CPUs (through multithreading), using distributed processing, and as demonstrated in this paper, using commodity GPUs to speed up certain computations.

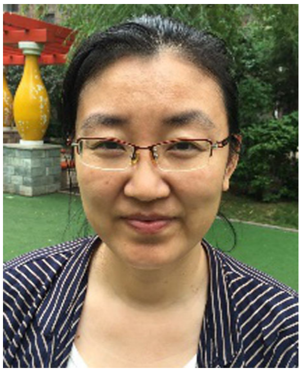
In this research, we found that nearly all complex and time-costly computations of RSAs can be sped up substantially by offloading work to the GPU. The CUDA technology used in our experiments was modern GPGPU architecture, which is adopted by many NVIDIA GPUs. As current trends indicate, future GPU designs, also based on general-purpose multiprocessors, will offer even more computational power. Our primary purpose in this research was to prove that developing GPU-aware operation research software is possible and useful. More GPU-based approaches for LPs should be further explored. We plan to extend our GPU-based RSA to hold memory download cost further. Another key direction is to use the GPU for IPMs. Finally, we also want to try to use some other ideas [7, 27, 29] to improve the effectiveness of our algorithm.

Acknowledgements This work was supported in part by the National Natural Science Foundation of China (51679105, 61672261,51409117), Jilin Province Department of Education Thirteen Five science and technology research projects [2016] No. 432, [2017] No.JJKH20170804KJ.

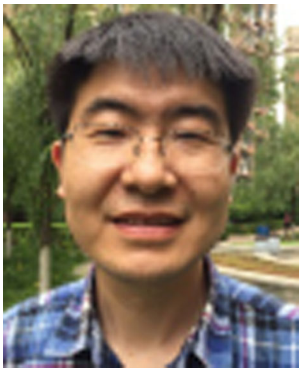
References

1. Anderson JA, Lorenz CD, Travesset A (2008) General purpose molecular dynamics simulations fully implemented on graphics processing units. J Comput Phys 227(10):5342–5359
2. Belleman RG, Bédorf J, Zwart SFP (2008) High performance direct gravitational N-body simulations on graphics processing units II: an implementation in CUDA. New Astron 13(2):103–112
3. Belloch JA, Gonzalez A, Vidal AM, Cobos M (2015) On the performance of multi-GPU-based expert systems for acoustic localization involving massive microphone arrays. Expert Syst Appl 42(13):5607–5620
4. Demirel E, Demirel N, Gökçen H (2016) A mixed integer linear programming model to optimize reverse logistics activities of end-of-life vehicles in Turkey. J Clean Prod 112:2101–2113
5. Douglas CC, Lee L, Yeung MC (2016) Hierarchical density-based clustering based on GPU accelerated data indexing strategy. Procedia Comput Sci 80:951–961
6. Ezzati R, Khorram E, Enayati R (2015) A new algorithm to solve fully fuzzy linear programming problems using the MOLP problem. Appl Math Model 39(12):3183–3193
7. Fu Z, Ren K, Shu J, et al. (2015) Enabling personalized search over encrypted outsourced data with efficiency improvement
8. Gobron S, Devillard F, Heit B (2007) Retina simulation using cellular automata and GPU programming. Mach Vis Appl 18(6):331–342
9. Guastaroba G, Mansini R, Ogryczak W, Speranza MG (2016) Linear programming models based on omega ratio for the enhanced index tracking problem. Eur J Oper Res 251(3):938–956
10. Ho TY, Lam PM, Leung CS (2008) Parallelization of cellular neural networks on GPU. Pattern Recogn 41(8):2684–2692
11. Huangfu Q, Hall JAJ (2015) Novel update techniques for the revised simplex method. Comput Optim Appl 60(3):587–608
12. Jermain CL, Rowlands GE, Buhrman RA, Ralph DC (2016) GPU-accelerated micromagnetic simulations using cloud computing. J Magn Magn Mater 401:320–322
13. Junior HV, Lins MPE (2005) An improved initial basis for the simplex algorithm. Comput Oper Res 32(8): 1983–1993
14. Leung CS, Wong TT, Lam PM et al (2006) An RBF-based compression method for image-based relighting. IEEE Trans Image Process 15(4):1031–1041
15. Liu W, Schmidt B, Voss G et al (2007) Streaming algorithms for biological sequence alignment on GPUs. IEEE Trans Parallel Distrib Syst 18(9):1270–1281

16. Liu W, Schmidt B, Voss G, et al. (2007) Molecular dynamics simulations on commodity GPUs with CUDA//International Conference on High-Performance Computing. Springer Berlin Heidelberg 185–196
17. Luh H, Tsaih R (2002) An efficient search direction for linear programming problems. Comput Oper Res 29(2):195–203
18. Marziale L, Richard GG, Roussev V (2007) Massive threading: using GPUs to increase the performance of digital forensics tools. Digit Investig 4:73–81
19. Oh KS, Jung K (2004) GPU implementation of neural networks. Pattern Recogn 37(6):1311–1314
20. Owens JD, Luebke D, Govindaraju N et al (2007) A survey of general-purpose computation on graphics hardware //computer graphics forum. Blackwell Publishing Ltd 26(1):80–113
21. Paparrizos K, Samaras N, Stephanides G (2003) A new efficient primal dual simplex algorithm. Comput Oper Res 30(9):1383–1399
22. Peercy M, Segal M, Gerstmann D. (2006) A performance-oriented data parallel virtual machine for GPUs// ACM SIGGRAPH 2006 Sketches. ACM. 184
23. Richter C, Schöps S, Clemens M. (2016) Multi-GPU Acceleration of Algebraic Multigrid Preconditioners// Scientific Computing in Electrical Engineering. Springer International Publishing. 83–90
24. Rocha P, Rodrigues R, Gomes A M, et al. (2015) GPU-Based Computing for Nesting Problems: The Importance of Sequences in Static Selection Approaches//Operations Research and Big Data. Springer International Publishing 195–202
25. Sanders J, Kandrot E. (2010) CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional
26. URL: http://docs.nvidia.com/cuda/optimus-developer-guide/index.html, 2017
27. Xia Z, Wang X, Sun X, Wang Q (2016) A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. IEEE Trans Parallel Distrib Syst 27(2):340–352
28. Yang X, Pi X, Zeng L, et al. (2005) GPU-based real-time simulation and rendering of unbounded ocean surface//Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05). IEEE. 6 pp
29. Zhangjie F, Xingming S, Qi L et al (2015) Achieving efficient cloud search services: multi-keyword ranked search over encrypted cloud data supporting parallel computing. IEICE Trans Commun 98(1):190–200
30. Zhong Z, Feng M, Liu D (2015) Parallelization of revised simplex algorithm on GPUs //network and information Systems for Computers (ICNISC), 2015 international conference on. IEEE. 349–353



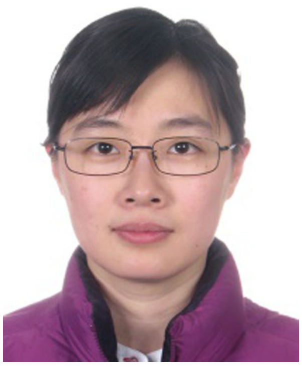
Lili He received her B.S., M.E. and Ph.D. degrees from the College of Computer Science and Technology, Jilin University, China, in 1998, 2001 and 2007, respectively. Now she is a lecturer in the College of Computer Science and Technology, Jilin University, China. Her research fields include wireless sensor network and theorem proving system.



Hongtao Bai received his B.S., M.E. and Ph.D. degrees from the College of Computer Science and Technology, Jilin University, China, in 1998, 2003 and 2010, respectively. Now he is a vice professor in the Center for Computer Fundamental Education, Jilin University, China. His research fields include high performance computing and wireless sensor network.

Yu Jiang received his M.E. and Ph.D. degrees from the College of Computer Science and Technology, Jilin University, China, in 2005 and 2011, respectively. Now he is a lecturer in the College of Computer Science and Technology, Jilin University, China. His research fields include wireless sensor network and data mining.

Dantong Ouyang received Ph.D. from Department of Computer Science, Jilin University in 1998. Now she is a professor and supervisor for Ph.D. candidates in the College of Computer Science and Technology, Jilin University, China. Her research fields include model-based diagnosis and theorem proving system.



Shanshan Jiang received her B.S. and M.E. degrees from the College of Computer Science and Technology, Jilin University, China, in 2005 and 2008, respectively. Her research fields include high performance computing and network optimization.