

Adaptive Optimization l_1 -Minimization Solvers on GPU

Jiaquan Gao¹ · Zejie Li¹ ·
Ronghua Liang¹ · Guixia He²

Received: 17 September 2015 / Accepted: 12 April 2016 / Published online: 21 April 2016
© Springer Science+Business Media New York 2016

Abstract l_1 -minimization (l_1 -min) algorithms for the l_1 -min problem have been widely developed. For most l_1 -min algorithms, their main components include dense matrix-vector multiplications such as Ax and $A^T x$, and vector operations. We propose a novel warp-based implementation of the matrix-vector multiplication (Ax) on the graphics processing unit (GPU), called the GEMV kernel, and a novel thread-based implementation of the matrix-vector multiplication ($A^T x$) on the GPU, called the GEMV-T kernel. For the GEMV kernel, a self-adaptive warp allocation strategy is used to assign the optimal warp number for each matrix row. Similar to the GEMV kernel, we design a self-adaptive thread allocation strategy to assign the optimal thread number to each matrix row for the GEMV-T kernel. Two popular l_1 -min algorithms, fast iterative shrinkage-thresholding algorithm and augmented Lagrangian multiplier method, are taken for example. Based on the GEMV and GEMV-T kernels, we present two highly parallel l_1 -min solvers on the GPU utilizing the technique of merging kernels and the sparsity of the solution of the l_1 -min algorithms. Furthermore, we design a concurrent multiple l_1 -min solver on the GPU, and optimize its performance by using new features of GPU such as the shuffle instruction and read-only data cache. The experimental results have validated high efficiency and good performance of our methods.

This work is supported by the National Science Foundation of China under Grant Number 61379017 and the Science and Technology Plan Project of Zhejiang Province, China under Grant Number 2014C33077.

✉ Jiaquan Gao
springf12@163.com

¹ College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou, Zhejiang Province 310023, China

² Zhijiang College, Zhejiang University of Technology, Hangzhou, Zhejiang Province 310024, China

Keywords l_1 -minimization · Matrix-vector multiplication · Fast iterative shrinkage-thresholding algorithm · Augmented Lagrangian multiplier method · GPU

1 Introduction

The l_1 -minimization (l_1 -min) problem can be written as:

$$\min \|x\|_1 \quad s.t. \quad Ax = b, \quad (1)$$

where $A \in \mathbb{R}^{m \times n}$ ($m \ll n$) is a full-rank dense matrix, $b \in \mathbb{R}^m$ is a pre-specified vector, and $x \in \mathbb{R}^n$ is an unknown solution. The l_1 -min solution, called the sparse representation, has proven to be sparse. Due to the sparsity of the solution, the l_1 -min optimization has been successfully applied in various fields such as signal processing [1–8], machine learning [9–11] and statistical inference [12, 13] and so on.

To solve the l_1 -min problem (1), researchers have developed many efficient algorithms. E.g., the gradient projection method [14], truncation Newton interior-point method [15], homotopy methods [16], class of iterative shrinkage-thresholding methods [17, 18], augmented Lagrange multiplier method (ALM) [19, 20], and alternating direction method of multipliers [21] and so on. A survey by Yang et al. [22] has compared and benchmarked these representative algorithms.

With the increasing scale of the problem, the execution efficiency of existing l_1 -min algorithms decreases to a large degree. An efficient way is to shift these algorithms to the distributed or multi-core architecture such as graphics processing units (GPUs). Big data processing using GPUs has drawn a lot of attention in recent years. Following the introduction of the compute unified device architecture (CUDA), a programming model that is designed to support joint CPU/GPU execution of applications, by NVIDIA in 2007 [23], GPUs have become increasingly strong competitors among the general-purpose parallel programming systems.

For most l_1 -min algorithms, their main components include dense matrix-vector multiplications such as Ax and $A^T x$, and vector operations. There have been highly efficient implementations for Ax , $A^T x$, and vector operations on the GPU in the CUBLAS library [24]. Therefore, the existing GPU-accelerated l_1 -min algorithms [25, 26] are mostly based on CUBLAS. On the NVIDIA GTX980 GPU, for the test matrices with m varying from 50 to 5000 and n being fixed at 100,000, the performance curves of the Ax and $A^T x$ implementations in CUBLAS are shown in Figs. 1a and 2a, respectively. For the test matrices with n varying from 4000 to 520,000 and m being fixed at 1000, the performance curves of the Ax and $A^T x$ implementations in CUBLAS are shown in Figs. 1b and 2b, respectively. Obviously, for the Ax and $A^T x$ implementations in CUBLAS, the performance value fluctuates as m increases or n increases, and the difference between the maximum and minimum performance values is distinct. In addition, we observe that when parallelizing the l_1 -min algorithms on the GPU, the number of kernels can be reduced by merging some operations into a single kernel, which can save time between kernel calls and avoid double loads of

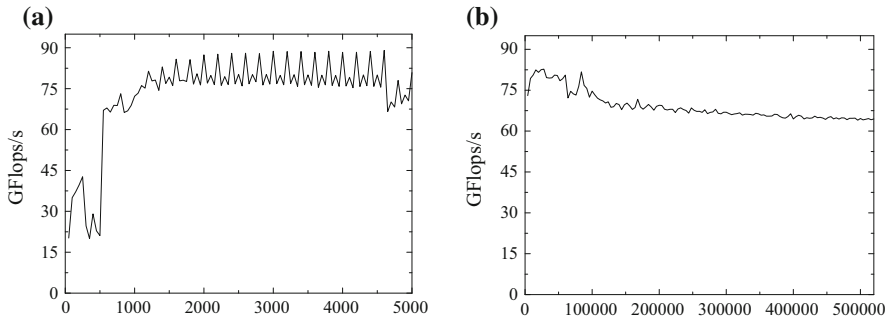


Fig. 1 The implementation of Ax in CUBLAS. **a** The performance curve with m ($n = 100,000$), **b** the performance curve with n ($m = 1000$)

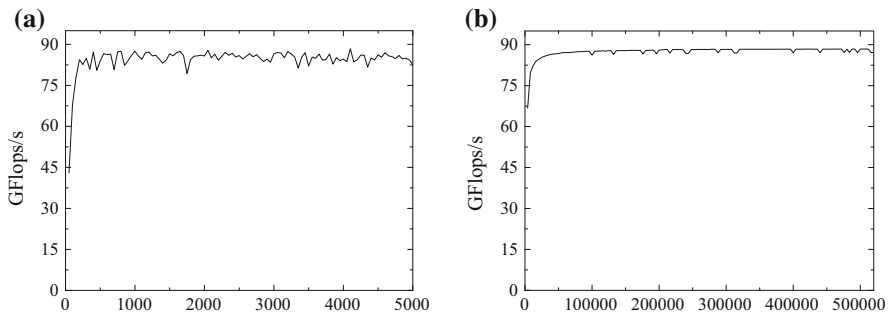


Fig. 2 The implementation of $A^T x$ in CUBLAS. **a** The performance curve with m ($n = 100,000$), **b** the performance curve with n ($m = 1000$)

vectors. However, CUBLAS does not allow merging several operations into a single kernel. At present, some new features for the NVIDIA GPU with compute capability 3.2 or higher, such as the shuffle instruction and read-only data cache, can be utilized to improve the performance of GPU-accelerated methods but not yet utilized in CUBLAS.

Therefore, these observations motivate us to further investigate the design of robust and highly parallel l_1 -min solvers on the GPU. In this study, we propose a novel warp-based implementation of Ax on the GPU, called the GEMV kernel, and a novel thread-based implementation of $A^T x$ on the GPU, called the GEMV-T kernel. For the GEMV kernel, a self-adaptive warp allocation strategy is used to assign the optimal warp number for each matrix row. Similar to the GEMV kernel, we design a self-adaptive thread allocation strategy to assign the optimal thread number to each matrix row for the GEMV-T kernel. Experimental results show that our proposed two kernels are more robust than CUBLAS, and always have high performance. In addition, two popular l_1 -min algorithms, fast iterative shrinkage-thresholding algorithm (FISTA) and augmented Lagrangian multiplier method (ALM), are taken for example. Utilizing the technique of merging kernels and the sparsity

of the solution of the l_1 -min algorithms, we propose two highly parallel l_1 -min solvers on the GPU. Furthermore, we design a concurrent multiple l_1 -min solver and optimize its performance by using the shuffle instruction and read-only data cache.

In summary, our work makes the following contributions:

- Two novel adaptive optimization GPU-accelerated implementations of the matrix-vector multiplication are proposed. The two methods are more robust than CUBLAS, and always have high performance.
- Based on the above implementations of the matrix-vector multiplication on the GPU, we present two highly parallel l_1 -min solvers on the GPU by utilizing the technique of merging kernels and the sparsity of the solution of the l_1 -min algorithms.
- Utilizing new features of GPU, we design an optimal concurrent multiple l_1 -min solver on the GPU.

The remainder of this paper is organized as follows. In Sect. 2, we describe two l_1 -min algorithms, FISTA and ALM. In Sect. 3, we introduce the CUDA architecture. Two adaptive optimization implementations of the matrix-vector multiplication on the GPU, GPU-accelerated FISTA and ALM solvers, a concurrent multiple l_1 -min solver on the GPU and some optimization strategies are proposed in Sect. 4. Experimental results are presented in Sect. 5. Section 6 contains our conclusions and points to our future research directions.

2 Two l_1 -min Algorithms

2.1 Fast Iterative Shrinkage-Thresholding Algorithm

The problem in Eq. (1) is known as the basis pursuit (BP) problem [7]. In practice, a measurement data b often contains noise (such as the measurement error ε), which is called the BPDN problem. A variant of this problem is also well known as the unconstrained BPDN problem with a scalar weight λ or the Lasso problem [27] in the statistics perspective:

$$\min \frac{1}{2} \|Ax - b\|_2^2 + \lambda \|x\|_1. \tag{2}$$

The fast iterative shrinkage-thresholding algorithm (FISTA) is a kind of accelerations, and achieves an accelerated non-asymptotic convergence rate of $O(k^2)$ by combining Nesterovs optimal gradient method [17,18]. For FISTA, it adds a new sequence $\{y_k, k = 1, 2, \dots\}$ as follows.

$$\begin{cases} x_{k+1} = \text{soft} \left(y_k - \frac{1}{L_f} \nabla f(y_k), \frac{\lambda}{L_f} \right) \\ t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2} \\ y_{k+1} = x_k + \frac{t_k - 1}{t_{k+1}} (x_k - x_{k-1}), \end{cases} \tag{3}$$

where $\text{soft}(u, a) = \text{sign}(u) \max\{|u| - a, 0\}$ is the soft-thresholding operator, $y_1 = x_0$, $t_1 = 1$ and the associated Lipschitz constant L_f of $\nabla f(\cdot)$ is given by the spectral norm of $A^T A$, denoted by $\|A^T A\|_2$. Algorithm 1 summarizes the generic FISTA algorithm.

Algorithm 1 Fast Iterative Shrinkage-Thresholding Algorithm (FISTA)

Input: $A \in R^{m \times n}$, $b \in R^m$, $\lambda \in R$
Init: $L_f = \|A^T A\|_2$, $k = 0$, $y_1 = x_0 = 0 \in R^n$, $t_1 = 1$, $a = \frac{\lambda}{L_f}$

1. **while** not converged ($k=1,2,\dots$) **do**
2. $\nabla f(y_k) = A^T (Ay_k - b)$
3. $u_k = y_k - \frac{1}{L_f} \nabla f(y_k)$
4. $x_k = \text{soft}(u_k, a) = \max(0, u_k - a) - \max(0, -u_k - a)$
5. $t_{k+1} = \frac{1\sqrt{1 + 4t_k^2}}{2}$
6. $y_{k+1} = x_k + (\frac{t_k - 1}{t_{k+1}})(x_k - x_{k-1})$
7. **end while**

Output : $x^* = x_k$

2.2 Augmented Lagrangian Multiplier Method

The augmented Lagrangian multiplier method (ALM) [19,20] combines penalty methods and the Lagrange multiplier algorithm, and its corresponding augmented Lagrangian function is

$$L_\rho(x^*, \lambda) = \|x\|_1 + \lambda^T (Ax - b) + \frac{\rho}{2} \|Ax - b\|_2^2, \tag{4}$$

where x^* is the optimal solution for the problem in Eq. (1), λ is a vector of Lagrange multipliers, $\frac{\rho}{2} \|Ax - b\|_2^2$ is the quadratic penalty function, $\rho > 0$ is a constant that determines the penalty for the infeasibility.

Equation (4) can be solved by the following iterative procedure called the method of multipliers [28].

$$\begin{cases} x_{k+1} = \min L_\rho(x, \lambda_k), \\ \lambda_{k+1} = \lambda_k + \rho_k (Ax_{k+1} - b), \end{cases} \tag{5}$$

where $\{\rho_k\}$ is a monotonically increasing positive sequence and is sufficiently large after a certain index. By the procedure shown in Eq. (5), we can simultaneously calculate the optimal solutions x^* and λ^* .

The first step of the procedure in Eq. (5), called x -min which is also a convex optimization problem, can be solved by FISTA. The generic ALM algorithm is summarized in Algorithm 2.

Algorithm 2 Augmented Lagrangian Multiplier Algorithm (ALM)**Input:** $A \in R^{m \times n}$, $b \in R^m$, $\lambda \in R$ **Init:** $L_f = \|A^T A\|_2$, $k = 0$, $x_0 = 0 \in R^n$, $\lambda_0 = 0$, $\rho > 1, \alpha > 1$

1. **While** not converged ($k=0,1,2,\dots$) **do**
 2. $l = 0, z_0 = y_1 = x_k \in R^n, t_1 = 1, a = \frac{1}{\rho L_f}$
 3. **while** not converged ($l=1,2,\dots$) **do**
 4. $\nabla f(y_l) = A^T (Ay_l - b + \frac{\lambda_k}{\rho_k})$
 5. $u_k = y_l - \frac{1}{L_f} \nabla f(y_l)$
 6. $z_l = \text{soft}(u, a) = \max(0, u_k - a) - \max(0, -u_k - a)$
 7. $t_{l+1} = \frac{1 + \sqrt{1 + 4t_l^2}}{2}$
 8. $y_{l+1} = z_l + (\frac{t_l - 1}{t_{l+1}})(z_l - z_{l-1})$
 9. **end while**
 10. $x_{k+1} = z_l$
 11. $\lambda_{k+1} = \lambda_k + \rho(Ax_{k+1} - b)$
 12. $\rho_{k+1} = \alpha \rho_k$ <optional>
 13. **end while**
- Output:** $x^* = x_{k+1}$

3 CUDA Architecture

The compute unified device architecture (CUDA) is a general purpose parallel computing platform and programming model. The developers can use provided CUDA C/C++ to define the function, called kernel, which is executed in parallel by each CUDA thread. All threads are organized into the thread blocks, then these thread blocks are organized into a grid. Both the thread block and the grid can have up to three dimensions.

A CUDA-enabled GPU has many computing cores called CUDA cores, which can collectively run thousands of threads. These CUDA cores are organized into a scalable array of streaming multiprocessors (SMs). A SM is designed to concurrently execute hundreds of threads by employing a unique architecture called SIMT (Single Instruction, Multiple-Thread). When a thread block is given to a SM, it is split in warps, each composed of 32 threads. In the best case, all 32 threads have the same execution path and the instruction is executed concurrently.

The GPU memory includes the on-chip memory, e.g., the shared memory and L1 cache, and the on-board memory, e.g., the global memory. For GPUs with different

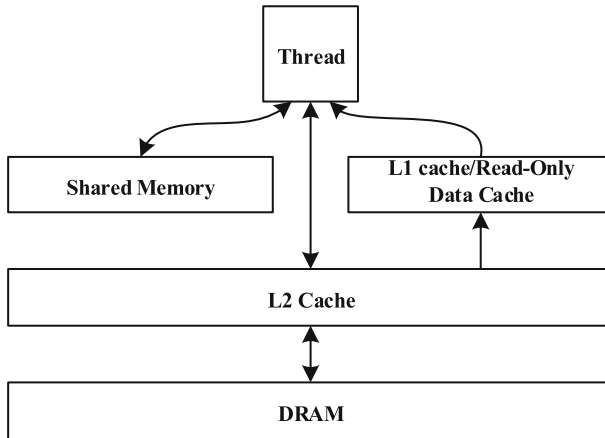


Fig. 3 The way of accessing data for threads

compute capabilities, the way of accessing data from the memory has some difference. In this study, we only consider GPUs with compute capability 3.2 or more in order to utilize new features such as the read-only data cache. Figure 3 shows the possible way of accessing data from the memory for threads. The global memory has large size and is shared by all SMs. However, since it resides on the off-chip DRAM (Dynamic Random Access Memory), the low bandwidth and large latency result in the slow access. The shared memory is used by all threads in a thread block, and provides the high bandwidth and low latency, but its size is small. Both the L1 cache and the read-only data cache are shared by all the threads within a SM, and can be accessed fast like the shared memory. Compared with the uncontrollable L1 cache, the read-only data cache is controllable and can be easily used by programmers.

Therefore, major challenges in optimizing an application on GPUs are: global memory access latency, the on-chip memory access efficiency, different execution paths in each warp, communication and synchronization between threads in different blocks and resource utilization.

4 GPU Implementation

4.1 Data Layout

We use a row major and 0-based indexing array a to store the matrix A , and utilize the padding scheme to optimize the global memory access performance, as shown in Fig. 4.

For devices with compute capability 2.0 or higher, global memory accesses by threads within a warp can be coalesced into the minimum number of L2-cache-line-sized (i.e., 32 bytes) aligned transactions. We assume that each matrix row is assigned to a warp with 32-byte memory transaction. From Fig. 4, we observe that each row of the matrix A includes 15 float-precision elements and is misaligned before padding.

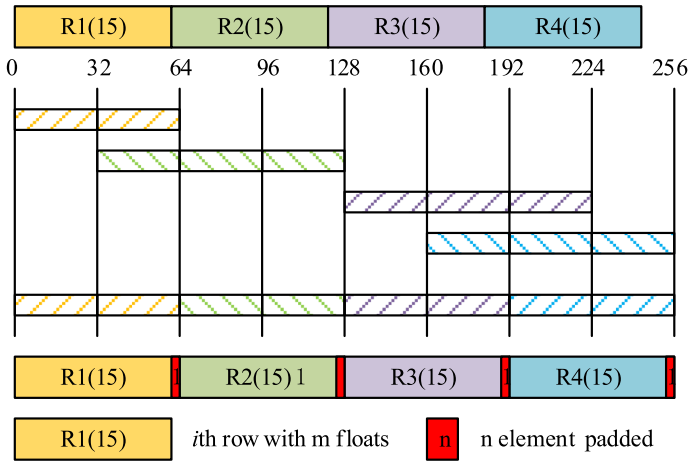


Fig. 4 Padding scheme for the array a

The misaligned case results in requiring 11 memory transactions for the first four rows of A . Padding the array a can decrease the number of memory transactions to 8. Therefore, the padding scheme can result in about 30% improvement of the global memory-access performance for this case in Fig. 4.

4.2 Robust Matrix-Vector Multiplications

The matrix-vector multiplications include Ax (GEMV) and $A^T x$ (GEMV-T), where $A \in R^{m \times n}$. In the following subsections, we propose two robust matrix-vector multiplication kernels. For the two kernels, we take full advantage of the multi-level cache hierarchy to cache vector x , thus improve the access efficiency. And they are robust and extensible for different GPU devices. The grid of our kernels is organized as a 1D array of thread blocks, and the thread block is also organized as a 1D array of threads.

4.2.1 GEMV Kernel

The GEMV Ax is composed of m dot products of x with each row of A , and these dot products can be independently computed. Thus, for our proposed GEMV kernel, we can assign one warp or multiple warps to a product dot. To optimize the GEMV kernel performance, we use the following self-adaptive warp allocation strategy to select the number of warps k for a product dot:

$$\min w = sm \times 2048/k/32, \quad s.t. \quad m \leq w, \tag{6}$$

where sm is the number of streaming multiprocessors and m is the number of the matrix rows.

Figure 5 shows the main procedure of the GEMV kernel. It is composed of two stages. The first stage includes three steps: *x-load step*, *partial-reduction step* and *warp-reduction step*.

x-load step The step is used to make threads per block parallel read elements of x into the shared memory xP . Because the size of x is large, x is segmentally read into the shared memory, and each time the size is $blockDim.x$ (size of thread blocks). By this way, the accesses to x are coalesced, and the access number is reduced by letting warps in the same thread block to share the section of elements of x .

partial-reduction step Each time after a section of elements of x is read into the shared memory, the threads in each warp group (k warps are grouped into a warp group) perform in parallel a partial-style reduction (see lines 12–5 in Fig. 5). Obviously, each thread in a warp group at most performs $\lceil n/wgSize \rceil$ times of reductions and the accesses to the global memory A are coalesced.

warp-reduction step After the threads in each warp group have completed the partial-style reductions, the fast shuffle instructions are utilized to perform a warp-style

```

Input:  $a, x, k, h, m, n$ ;
Output:  $b$ ;
1.  $\_shared\_xP[], bP[]$ ;
2.  $bVAL \leftarrow 0.0f$ ;
3.  $lane \leftarrow threadIdx.x \& 31$ ;  $wid \leftarrow threadIdx.x \gg 5$ ;
4.  $wgSize \leftarrow 32 * k$ ;  $wgid \leftarrow threadIdx.x / wgSize$ ;
5.  $hub \leftarrow threadIdx.x \& (wgSize - 1)$ ;
6.  $row \leftarrow blockDim.x * h + wgid$ ; // the assigned row index
7. for  $row$  to  $m - 1$  with  $row += h * gridDim.x$ 
   // The first stage
8.  $bVAL \leftarrow 0$ ;  $aid \leftarrow row * n + hub$ ;
9. for  $xid = threadIdx.x$  to  $n - 1$  with  $xid += blockDim.x$ 
   // The x-load step
10.  $xP[threadIdx.x] \leftarrow x[xid]$ ;
11.  $\_syncthreads()$ ;
   // The partial-reduction step
12. for  $i = hub$  to  $blockDim.x - 1$  with  $i += wgSize$ 
13.  $bVAL \leftarrow bVAL + a[aid] * xP[i]$ ;
14.  $aid \leftarrow aid + wgSize$ ;
15. end
16.  $\_syncthreads()$ ;
17. end
18. Using shuffle instructions in its warp
19. if  $lane == 0$   $bP[wid] \leftarrow bVAL$ ;
20.  $\_syncthreads()$ ;
   // The second stage
21. if  $wid == 0$ 
22.  $bVAL \leftarrow bP[lane]$ ;
23. Using shuffle instructions in its warp
24. if  $tid \% k == 0$   $b[row + threadIdx.x / k] \leftarrow bVAL$ ;
25. end
26. end

```

Fig. 5 The GEMV kernel

reduction for each warp in these warp groups. The warp-style reduction values are stored in the shared memory.

In the second stage, the warp-style reduction values in the shared memory for each warp group are reduced to an output value in parallel.

For $k = 1$, a warp group only includes a warp. From Fig. 5, we observe that it only needs the first stage and the output vector b can be obtained. Thus, we specially design a GEMV kernel for $k = 1$ in order to optimize the GEMV kernel performance (omitted here). Similarly, a GEMV kernel for $k = 32$ is also redesigned (omitted here) because it is not necessary to read x into the shared memory for this case. We can directly load x into registers.

4.2.2 GEMV-T Kernel

The GEMV-T, $A^T x$, is composed of n dot products of x with each columns of A , and these dot products can be independently computed. Comparing with the GEMV, the size of the vector x in the GEMV-T is small. Thus we assign one thread or multiple threads to a dot product in our proposed GEMV-T kernel. To optimize the GEMV-T kernel performance, we use the following self-adaptive thread allocation strategy to select the number of threads k for a product dot:

$$\min t = sm \times 2048/k, \quad s.t. \quad n \leq t, \quad (7)$$

where sm is the number of streaming multiprocessors and n is the number of the matrix columns.

Figure 6 shows the main procedure of the GEMV-T kernel. Like the GEMV kernel, it also needs two stages. In the first stage, it is only composed of *x-load step* and *partial-reduction step*. The *x-load step* has the same function as in the GEMV kernel. In the partial-reduction step, since a row major and 0-based index format is used to store the matrix A , the accesses to A will not be coalesced if the thread groups (k threads are grouped into a thread group) are constructed in an inappropriate way. For example, we assume that A is a 4×8 matrix as shown in Fig. 7, 16 threads in a thread block are launched, and two threads are assigned to a dot product in the GEMV-T. If we use the following thread groups $\{0, 1\}, \{2, 3\}, \{4, 5\}, \dots, \{14, 15\}$, the accesses to A will not be coalesced (see Fig. 7a). However, when the thread groups $\{0, 8\}, \{1, 9\}, \{2, 10\}, \dots, \{7, 15\}$ are utilized, the accesses to A are coalesced, as shown in Fig. 7b. Therefore, in the partial-reduction step, the thread groups are created according to Definition 1 below in order to ensure that the accesses to A are coalesced.

Definition 1 Assume that the size of the thread block is s , h threads are assigned to a dot product in $A^T x$, and $z = s/h$. The thread groups are created as follows: $\{0, z, \dots, (h-1)*z\}, \{1, z+1, \dots, (h-1)*z+1\}, \dots, \{z-1, 2*z-1, \dots, 2*(h-1)*z-1\}$.

For these elements of x read into the shared memory each time, the threads in each thread group perform in parallel a partial-style reduction similar to that in the GEMV kernel.

Since these threads in a thread group are usually not in the same warp, we can not use the shuffle instruction to reduce their partial-style reduction values. Therefore, in

```

Input:  $a, x, k, h, m, n$ ;
Output:  $b$ ;
1.  $\_\_shared\_\_ xP[], bP[]$ ;
2.  $bVAL$ ;
3.  $wid \leftarrow threadIdx.x \gg 5$ ;  $lane \leftarrow threadIdx.x \& 31$ ;
4.  $wgSize \leftarrow 32 * h$ ;  $wgid \leftarrow threadIdx.x / (wgSize)$ ;
   // the assigned column index
5.  $col \leftarrow blockIdx.x * k * 32 + 32 * wgid + lane$ ;
6. for  $col$  to  $n - 1$  with  $col += k * 32 * gridDim.x$ 
   // The first stage
7.  $bVAL \leftarrow 0$ ;  $aid \leftarrow col + wid \% h * n$ ;
8. for  $xid = threadIdx.x$  to  $m - 1$  with  $xid += blockDim.x$ 
   // The x-load step
9.  $xP[threadIdx.x] \leftarrow x[xid]$ ;
10.  $\_\_syncthreads()$ ;
11. // The partial-reduction step
12. for  $i = wid \% h$  to  $blockDim.x - 1$  with  $i += h$ 
13.  $bVAL \leftarrow bVAL + a[aid] * xP[i]$ ;
14.  $aid \leftarrow aid + h * n$ ;
15. end
16.  $\_\_syncthreads()$ ;
17. end
18.  $bP[tid] \leftarrow bVAL$ ;
19.  $\_\_syncthreads()$ ;
   // The second stage
20. if  $wid \% h == 0$ 
21. for  $bPid = tid$  to  $tid + 32 * h - 1$  with  $bPid += 32$ 
22.  $bVAL \leftarrow bVAL + bP[bPid]$ ;
23. end
24.  $b[col] \leftarrow bVAL$ ;
25. end
26. end
    
```

Fig. 6 The GEMV-T kernel

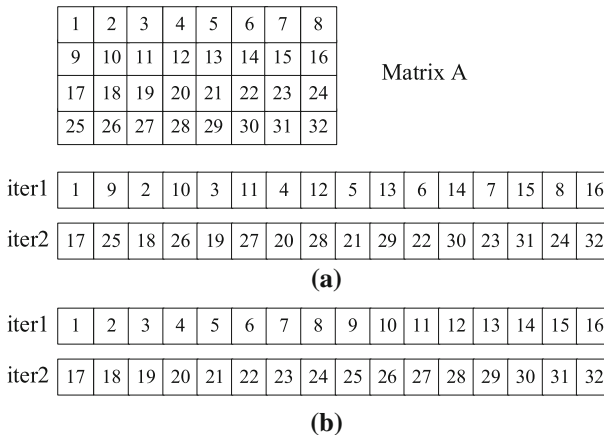


Fig. 7 Accesses to A

the second stage, we store the partial-style reduction values obtained by threads in each thread group to the shared memory, and then reduce them in the shared memory to an output value in parallel.

4.3 Parallel l_1 -min Solvers

4.3.1 FISTA and ALM Solvers

When parallelizing FISTA in Algorithm 1 on the GPU, 6 kernels, as shown in Fig. 8a, are needed. To minimize the number of kernels, save time between kernel calls, and avoid double loads of vectors, we merge these kernels by two steps. The first step merges the kernel 1 and the kernel 2 into a single kernel. In the second step, three vector operation kernels are merged into a single kernel. Thus, the total number of kernels is reduced from 6 to 3 (see Fig. 8b).

For kernels in Fig. 8b, it is easy to implement the first two kernels based on our proposed GEMV and GEMV-T implementation methods on the GPU. Although CUBLAS has shown high performance for the vector operations, CUBLAS does not allow merging several operations into a single kernel. Therefore, for the third kernel, we adopt the implementation method in [29], which supports merging several vector operations into a single kernel. The parallel FISTA on the GPU is called the FISTA solver.

For ALM in Algorithm 2, x -min can be solved by FISTA. Therefore, to parallelize ALM on the GPU, we need to design a kernel to finish the dual update, $\lambda_{k+1} = \lambda_k + \rho_k(Ax_{k+1} - b)$, besides utilizing three kernels in FISTA. Obviously, based on our proposed GEMV implementation method on the GPU, the dual update kernel is easy to be designed. The parallel ALM on the GPU is called the ALM solver.

4.3.2 Concurrent Multiple l_1 -min Solvers

In some real applications, we usually need to solve multiple l_1 -min problems concurrently. To accommodate the real requirements, here we take FISTA for example and thus design a GPU-accelerated implementation, as shown in Fig. 9, to solve multiple

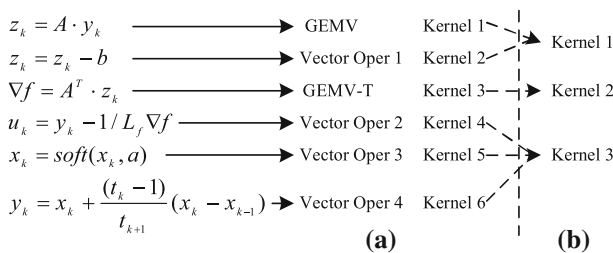


Fig. 8 Kernels in FISTA

l_1 -min problems concurrently. In this method, we assign one thread block to a l_1 -min problem, and for each thread block, the idea of constructing the parallel FISTA to solve the l_1 -min problem is similar to that of implementing FISTA on the GPU in Sect. 4.3.1. We call the concurrent multiple l_1 -min solver MFISTASOL.

In MFISTASOL, each thread block needs to access the global memory a , so we let a be cached in the read-only data cache in order to reduce the number of accesses to a . With the read-only data cache, a is shared by all thread blocks and can be accessed fast.

4.4 Optimization

When using FISTA or ALM to solve the l_1 -min problem, with the increasing number of iterations, the output vector x becomes sparser and sparser through the soft-thresholding operator as shown in Algorithms 1 and 2. Therefore, we can utilize

```

Input:  $a, b, Lf, \lambda, m, n$ ;
Output:  $x$ ;
1.  $\_shared\_yP[]$ ;
2.  $lane \leftarrow threadIdx.x \& 31$ ;  $wid \leftarrow threadIdx.x \gg 5$ ;
3.  $r \leftarrow wid$ ;  $c \leftarrow threadIdx.x$ ;  $t0 \leftarrow 1, t1 \leftarrow 0$ ;
4. for  $p = 0$  to  $numProblems - 1$  with  $p += gridDim.x$ 
5.   // map data index
6.    $y\_index \leftarrow (float *) (y + p * n)$ ;  $x\_index \leftarrow (float *) (x + p * n)$ ;
7.    $temp\_index \leftarrow (float *) (temp + p * m)$ ;  $b\_index \leftarrow (float *) (b + p * m)$ ;
8.   for  $iter = 0$  to  $numIterations - 1$  with  $iter += 1$ 
9.     // Ax-b part, one warp to a dot product
10.    for  $row = 0$  to  $m - 1$  with  $row += blockDim.x / 32$ 
11.       $bVAL \leftarrow 0$ ;  $aid \leftarrow row * n + lane$ ;
12.      for  $yid = threadIdx.x$  to  $n - 1$  with  $yid += blockDim.x$ 
13.        the x-load step
14.        the partial-reduction stepP
15.      end
16.      if  $lane == 0$   $temp\_index[row] = bVAL \leftarrow b[r]$ ;
17.    end //end of Ax
18.     $\_syncthreads()$ ;
19.    the computation of  $t1$ 
20.    //Atx fused with rest operations, one thread to a dot product in Atx
21.    for  $col = 0$  to  $n - 1$  with  $col += blockDim.x$ 
22.       $bVAL \leftarrow 0$ ;  $aid \leftarrow col$ ;
23.      for  $yid = threadIdx.x$  to  $m - 1$  with  $yid += blockDim.x$ 
24.        the x-load step
25.        the partial-reduction step
26.      end
27.       $u \leftarrow y[col] - bVAL / Lf$ ;
28.       $out \leftarrow \text{fmaxf}(0.0f, u - a) - \text{fmaxf}(0.0f, -u - a)$ ;
29.       $y\_index[col] \leftarrow out + ((t0 - 1) / t1) * (out - x\_index[col])$ ;
30.       $x\_index[col] \leftarrow out$ ;
31.    end //end of Atx
32.     $t0 \leftarrow t1$ ;
33.  end
34. end

```

Fig. 9 Concurrent multiple l_1 -min solver

the sparsity to reduce the accesses to the global memory a for the GEMV kernel in the FISTA solver, the ALM solver and MFISTASOL.

When the i th element of x is equal to zero, all elements in the i th column of A do not need to be accessed because they do not have any contribution to the output vector. With the increasing iteration in FISTA and ALM, x becomes sparser and sparser, and thus a number of columns of A are not accessed. By this way, we can improve the performance of the FISTA solver, the ALM solver and MFISTASOL by reducing accesses to the global memory a . However, since we use the row major and 0-based indexing array a to store the matrix A , the accesses to a in the above case are not coalesced. In this study, we alleviate the overhead deriving from the non-coalescence through L2 cache with 32-byte memory transactions.

5 Experimental Results

In this section, we test our proposed l_1 -min solvers on the GPU from the following four aspects : (1) analyzing the validity of using the vector sparsity to optimize the GEMV kernel performance and the read-only data cache to improve the MFISTASOL performance, (2) comparing GEMV and GEMV-T kernels with the implementation in the CUBLAS library, (3) testing the performance of our proposed parallel l_1 -min solvers, and (4) testing the performance of our proposed concurrent multiple l_1 -min solver.

The experimental environments include one machine which is equipped with an Intel Xeon Quad-Core CPU and an NVIDIA GTX980 GPU and another machine with an Intel Xeon Quad-Core CPU and an NVIDIA GTX760 GPU. Our source codes are compiled and executed using the CUDA toolkit 6.5.

The measured GPU performance for all experiments does not include the data transfer (from the GPU to the CPU or from the CPU to the GPU). The test matrices are shown in Table 1. The element values of each test matrix are randomly generated according to the normal distribution.

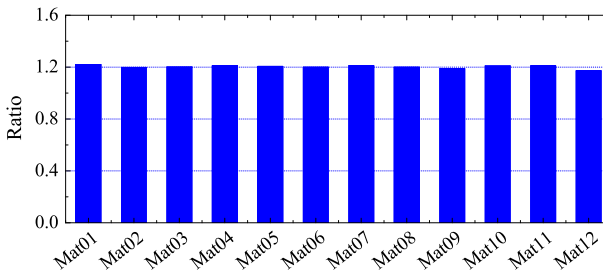
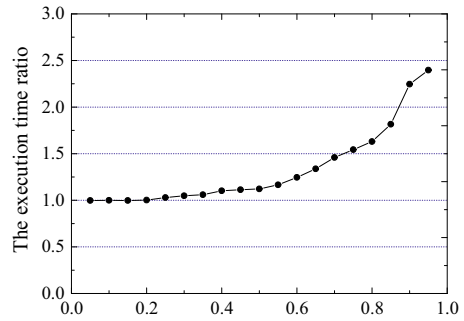
5.1 Experimental Analysis

First, we take Mat05 in Table 1 for example to test the performance influence of utilizing the sparsity of the vector x to optimize the GEMV (Ax) kernel. The GTX980 is used in this experiment. The ratio of the number of zero elements to the total number of elements in x , represented with δ , is set to 0.05, 0.10, 0.15, ..., 0.90 and 0.95, respectively. Figure 10 shows the execution time ratios of the GEMV kernel without the sparsity to the GEMV kernel with the sparsity for all δ values. We observe that the execution time ratio increases as δ increases. Therefore, we affirm that by utilizing the vector sparsity, the performance of the GEMV kernel is improved.

Second, we take the GTX980 for example to verify the validity of using the read-only data cache to improve the performance of the concurrent multiple l_1 -min solver (MFISTASOL). The test matrices are used in Table 1. 128 l_1 -min problems are concurrently calculated. In MFISTASOL, the total number of iterations is set to 10, the

Table 1 Test matrices

Seq	Matrix	Rows (m)	Columns (n)
1	Mat01	32	8,388,608
2	Mat02	50	5,368,709
3	Mat03	64	4,194,304
4	Mat04	100	2,684,350
5	Mat05	128	2,097,152
6	Mat06	200	1,342,200
7	Mat07	256	1,048,576
8	Mat08	400	671,100
9	Mat09	512	524,288
10	Mat10	800	335,850
11	Mat11	1024	262,144
12	Mat12	1600	166,900

Fig. 10 The execution time ratios for all δ values**Fig. 11** The execution time ratios for all test cases

initial x_0 is randomly generated according to the normal distribution, and $b = Ax_0$. The execution time ratios of MFISTASOL without the read-only data cache to MFISTASOL with the read-only data cache are shown in Fig. 11 for all test cases. We see that for all test cases, the execution time ratios have been sustained at around 1.2. Thus the MFISTASOL performance is improved by using the read-only data cache.

5.2 Performance Comparison of Matrix-Vector Multiplications

We compare GEMV and GEMV-T kernels with the implementation in CUBLAS. The performance is measured in terms of *Gflop/s*, which is obtained by $2 \times m \times n$ / the matrix-vector multiplication kernel execution time (the time unit is second) [30]. Figures 12 and 13 respectively show the comparison of the GEMV and GEMV-T kernels with CUBLAS. From Fig. 12, we see that on the GTX760 and GTX980, the GEMV kernel outperforms CUBLAS, and respectively obtains average speedups of $4.42 \times$ and $2.74 \times$ compared to CUBLAS. For the GEMV kernel, on the GTX760 and GTX980, the average performance values are respectively 57.69 GFlops/s and 83.81 GFlops/s , and the standard deviations are respectively 9.27 and 5.48. However, on the GTX760 and GTX980, CUBLAS only obtains the average performance values of 21.80 GFlops/s and 39.88 GFlops/s and the standard deviations of 18.73 and 20.47, respectively. This verifies that for matrices with different scales, our proposed GEMV kernel always has high performance, but CUBLAS does not. For the GEMV-T kernel, we can obtain the same conclusion as the GEMV kernel from Fig. 13.

Next, we take the GTX980 for example to further verify the above observations. The test setup is as same as in the introduction. Figure 14 shows the performance curves of the GEMV kernel and CUBLAS. Obviously, for all cases, our proposed GEMV kernel always obtains around 80 GFlops/s . However, CUBLAS has lower performance than our proposed GEMV kernel, and the difference between the maximum and minimum

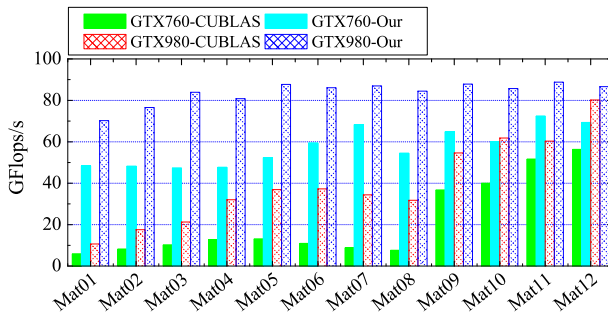


Fig. 12 The performance comparison of the GEMV kernel and CUBLAS

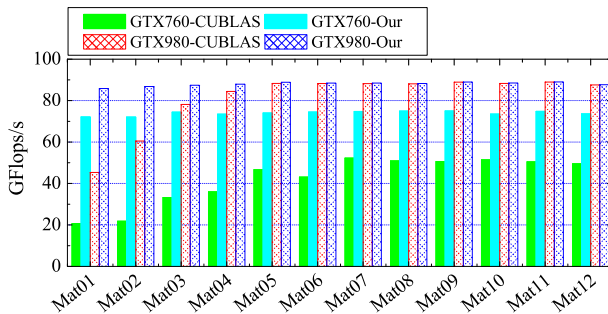


Fig. 13 The performance comparison of the GEMV-T kernel and CUBLAS

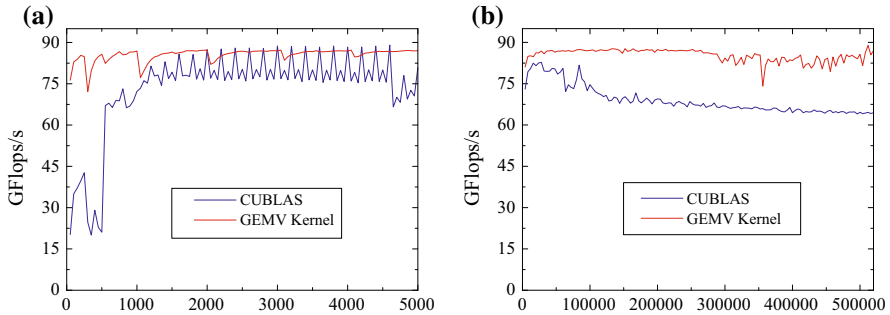


Fig. 14 GEMV. **a** The performance curves with m ($n = 100,000$), **b** the performance curves with n ($m = 1000$)

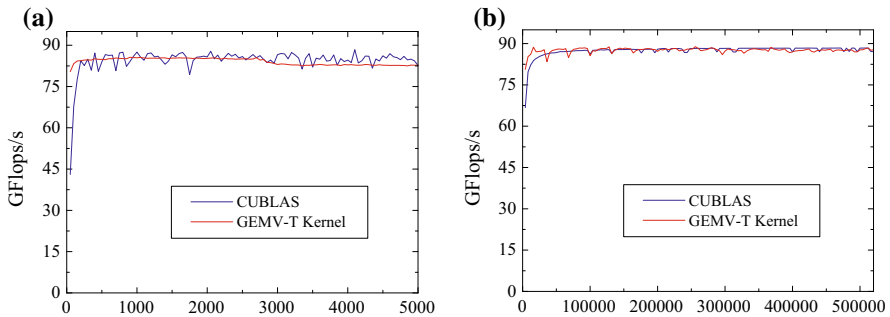


Fig. 15 GEMV-T. **a** The performance curves with m ($n = 100,000$), **b** the performance curves with n ($m = 1000$)

performance values is distinct. For the test matrices with n being set to a fixed value 100,000, the performance of our proposed GEMV-T kernel has been maintained at around 80 GFlops/s as m increases, as shown in Fig. 15a. The CUBLAS performance in general increases as m increases, and is maintained at around 80 GFlops/s only after m is more than 200. From Fig. 15b, for the test matrices with m being fixed at 1000, when n increases, we obtain the same conclusion as in Fig. 15a for the GEMV-T kernel and CUBLAS.

Therefore, we can conclude that our proposed matrix-vector multiplication implementations on the GPU are more robust than CUBLAS, and usually have high performance.

5.3 Performance of Parallel l_1 -min Solvers

We test the parallel performance of our proposed FISTA and ALM solvers by comparing them with the corresponding implementations using the CUBLAS library. The FISTA solver and corresponding implementation using the CUBLAS library are denoted as GFISTA and BFISTA, respectively. The ALM solver and corresponding

Table 2 Execution time of all algorithms

Prob	CFISTA	CALM	BFISTA	BALM	GFISTA	GALM
1	31.61	243.94	2.834	28.758	0.589	4.646
2	24.86	216.61	1.925	19.528	0.582	4.653
3	26.23	217.12	1.495	15.183	0.533	4.336
4	25.61	217.12	1.164	11.798	0.525	4.479
5	25.37	212.18	0.986	10.744	0.501	4.251
6	26.20	196.35	1.062	11.997	0.485	4.503
7	24.86	200.57	1.158	12.018	0.470	4.465
8	23.34	179.49	1.192	12.056	0.492	4.989
9	24.48	188.36	0.630	6.386	0.460	4.922
10	21.91	165.87	0.652	6.596	0.483	4.380
11	19.68	152.67	0.612	6.191	0.457	3.611
12	17.75	149.43	0.611	6.185	0.471	3.977

implementation using the CUBLAS library are denoted as GALM and BALM, respectively. CFISTA is the sequential CPU implementation corresponding to GFISTA, and CALM is the sequential CPU implementation corresponding to GALM. All experiments are conducted on the GTX980. For each l_1 -min problem, the matrix A comes from Table 1, the initial x_0 with 1024 non-zero elements is randomly generated according to the normal distribution, and $b = Ax_0$. CFISTA, BFISTA, and GFISTA stop after the number of iterations is more than 50 for all test cases. In the CALM, BALM, and GALM, the total number of iterations for the inner iteration and the outer iteration has been set to 50 and 10, respectively. Table 2 lists the execution time of all algorithms for all test cases. The speedups of BFISTA, GFISTA, BALM, and GALM are shown in Fig. 16. The time unit is second (denoted by s). From Table 2 and Fig. 16, we observe that compared to CFISTA, GFISTA obtains speedups ranging from 37.68 to 53.66 for all test cases, and the average speedup is 48.22. However, BFISTA only achieves speedups ranging from 11.15 to 38.82 for all test cases, and the average speedup is 24.05. For GALM, comparing with CALM, the maximum, minimum and average speedups are 51.21, 35.98 and 44.0, respectively, which are 2.04, 4.24, and 2.36 times faster than those that are obtained by BALM. All these results show that our proposed FISTA and ALM solvers have high parallelism, and outperform the corresponding implementations using the CUBLAS library.

In addition, we also take Mat07 and Mat12 for example to show the execution time of Kernel 1, Kernel 2, and Kernel 3 in the selected iteration steps for BFISTA and GFISTA in Fig. 17. The time unit is millisecond (denoted by ms). For each one of the two test matrices, the execution time of Kernel 2 of GFISTA and Kernel 2 of BFISTA nearly remains invariable for all selected iteration steps, and Kernel 2 of GFISTA almost has the same execution time as that of BFISTA. This observation is in accordance with that in Fig. 13. The execution time of Kernel 3 of GFISTA and Kernel 3 of BFISTA is almost invariable for all selected iteration steps, but Kernel 3 of GFISTA is nearly 2.08 times faster than that of BFISTA due to the merging of kernels.

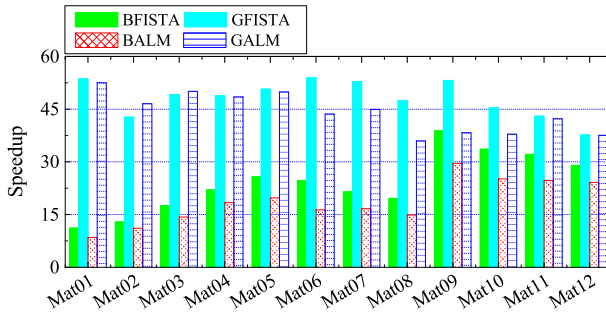


Fig. 16 Speedups of all algorithms

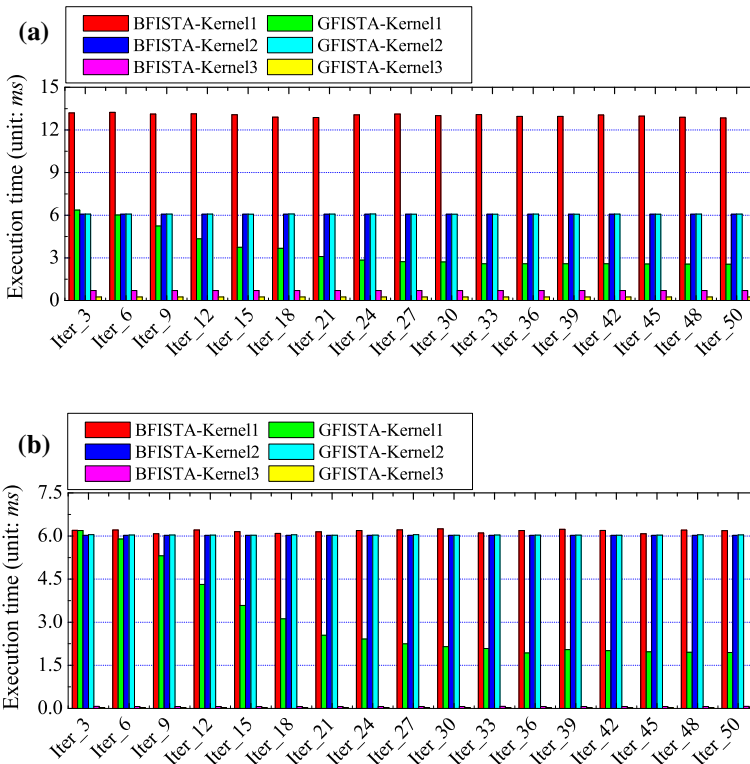


Fig. 17 Execution time of Kernel 1, Kernel 2, and Kernel 3 in the selected iteration steps for BFISTA and GFISTA. a Mat07, b Mat12

The execution time of Kernel 1 of BFISTA is nearly invariable for all selected iteration steps. However, the execution time of Kernel 1 of GFISTA decreases as the iteration step increases because of the sparsity of solutions, and is greatly advantageous over that of Kernel 1 of BFISTA in each iteration step. These results further verify the efficiency of our proposed FISTA and ALM solvers.

Table 3 Execution time of MFISTASOL

Prob	GFISTA	MFISTASOL	Speedup
1	0.589×128	24.033	3.137
2	0.582×128	27.805	2.679
3	0.533×128	21.282	3.205
4	0.525×128	26.240	2.561
5	0.501×128	19.708	3.252
6	0.485×128	25.479	2.441
7	0.470×128	19.219	3.132
8	0.492×128	25.454	2.476
9	0.460×128	19.276	3.058
10	0.483×128	21.458	2.881
11	0.457×128	19.051	3.075
12	0.471×128	20.720	2.912

5.4 Performance of the Concurrent Multiple l_1 -min Solver

We test the performance of our proposed concurrent multiple l_1 -min solver, MFISTA-SOL. The test setup is as same as in Sect. 5.3. For each test case, 128 l_1 -min problems are concurrently calculated. All experimental results on the GTX980 are shown in Table 3. The time unit is second (denoted by s). We observe that compared to the sequential execution of the FISTA solver, our proposed concurrent multiple l_1 -min solver, MFISTASOL, can obtain the average speedup of around 3.0.

6 Conclusion

This paper proposes two robust implementations of the matrix-vector multiplication on the GPU. Moreover, based on the two proposed matrix-vector multiplication implementations on the GPU, we presents two highly parallel l_1 -min solvers, the FISTA solver and the ALM solver, utilizing the technique of merging kernels and the sparsity of the solution of l_1 -min problems. To accommodate the real requirements of solving multiple l_1 -min problems concurrently, we design a concurrent multiple l_1 -min solver. Experimental results show that our proposed methods are efficient, and have high performance.

Next, we will further do research in this field, and apply the proposed solvers to the real problems.

Acknowledgements We gratefully acknowledge the comments from anonymous reviewers, which greatly helped us to improve the contents of the paper.

References

1. Bruckstein, A., Donoho, D., Elad, M.: From sparse solutions of systems of equations to sparse modeling of signals and images. *SIAM Review* **51**(1), 34–81 (2009)

2. Mallat, S.: *A Wavelet Tour of Signal Processing—The Sparse Way*, 3rd edn. Academic, Cambridge (2009)
3. Donoho, D.L., Elad, M.: Optimally sparse representation in general (nonorthogonal) dictionaries via l_1 minimization. *Proc. Natl. Acad. Sci.* **100**(5), 2197–2202 (2003)
4. Donoho, D.L., Elad, M.: On the stability of the basis pursuit in the presence of noise. *Signal Process.* **86**(3), 511–532 (2006)
5. Tropp, J.A.: Greed is good: algorithmic results for sparse approximation. *IEEE Trans. Inf. Theory* **50**(10), 2231–2242 (2004)
6. Tropp, J.A.: Just relax: convex programming methods for subset selection and sparse approximation. *IEEE Trans. Inf. Theory* **52**(3), 1030–1051 (2006)
7. Chen, S.S., Donoho, D.L., Saunders, M.A.: Atomic decomposition by basis pursuit. *SIAM J. Sci. Comput.* **20**(1), 33–61 (1998)
8. Candès, E., Romberg, J., Tao, T.: Stable signal recovery from incomplete and inaccurate measurements. *Commun. Pure Appl. Math.* **59**(8), 1207–1223 (2006)
9. Wright, J., Yang, A., Ganesh, A., Sastry, S., Ma, Y.: Robust face recognition via sparse representation. *IEEE Trans. Pattern Anal.* **31**(2), 210–227 (2009)
10. Elhamifar, E., Vidal, R.: Sparse subspace clustering: algorithm, theory, and applications. *IEEE Trans. Pattern Anal.* **35**(11), 2765–2781 (2013)
11. Elhamifar, E., Vidal, R.: Sparse subspace clustering: computer vision and pattern recognition. In: *IEEE Conference on CVPR 2009*, pp. 2790–2797 (2009)
12. Wright, J., Ma, Y., Mairal, J., et al.: Sparse representation for computer vision and pattern recognition. *Proc. IEEE* **98**(6), 1031–1044 (2010)
13. Zibulevsky, M., Elad, M.: L_1 – L_2 optimization in signal and image processing. *IEEE Signal Proc. Mag.* **27**(3), 76–88 (2010)
14. Figueiredo, M.A.T., Nowak, R.D., Wright, S.J.: Gradient projection for sparse reconstruction: application to compressed sensing and other inverse problems. *IEEE J. STSP* **1**(4), 586–597 (2007)
15. Kim, S.J., Koh, K., Lustig, M., Boyd, S., Gorinevsky, D.: An interior-point method for large-scale l_1 -regularized least squares. *IEEE J. STSP* **1**(4), 606–617 (2007)
16. Donoho, D.L., Tsai, Y.: Fast solution of L_1 -norm minimization problems when the solution may be sparse. Stanford University, Technical Report (2006)
17. Nesterov, Y.: Gradient methods for minimizing composite objective function. *Gen. Inf.* **38**(3), 768–785 (2007)
18. Beck, A., Teboulle, M.: A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Imaging Sci.* **2**(1), 183–202 (2009)
19. Bertsekas, D.: *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, Belmont (1982)
20. Yang, A.Y., Zhou, Z., Balasubramanian, A.G., Sastry, S.S., Ma, Y.: Fast l_1 -minimization algorithms for robust face recognition. *IEEE Trans. Image Process.* **22**(8), 3234–3246 (2013)
21. Stephen, B., et al.: Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.* **3**(1), 1–122 (2011)
22. Yang, A.Y., Sastry, S.S., Ganesh, A., Ma, Y.: Fast l_1 -minimization algorithms and an application in robust face recognition: a review. In: *17th IEEE International Conference on Image Processing (ICIP)*, pp. 1849–1852 (2010)
23. NVIDIA: CUDA C Programming Guide 6.5. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/> (2014)
24. NVIDIA: CUBLAS Library 6.5. <http://docs.nvidia.com/cuda/cublas-library/> (2014)
25. Nagesh, P., Gowda, R., Li, B.: Fast GPU implementation of large scale dictionary and sparse representation based vision problems. In: *2010 IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*, pp. 1570–1573 (2010)
26. Shia, V., Yang, A.Y., Sastry, S.S.: Fast l_1 -minimization and parallelization for face recognition. In: *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pp. 1199–1203 (2011)
27. Tibshirani, R.: Regression shrinkage and selection via the lasso. *J. R. Stat. Soc. Series B* **58**, 267–288 (1996)
28. Bertsekas, D.P.: *Nonlinear Programming*. Athena Scientific, Belmont (2003)
29. Gao, J., Liang, R., Wang, J.: Research on the conjugate gradient algorithm with a modified incomplete Cholesky preconditioner on GPU. *J. Parallel Distrib. Comput.* **74**(2), 2088–2098 (2014)

30. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC09). Portland, Oregon, USA: ACM, November, pp.14–19 (2009)