J Supercomput (2015) 71:2381–2396
DOI 10.1007/s11227-015-1386-z

Accelerating MRI reconstruction via three-dimensional dual-dictionary learning using CUDA

Jiansen Li · Jianqi Sun · Ying Song · Jun Zhao

Published online: 25 February 2015

Abstract Using undersampled k-space data for reconstruction is an effective way to accelerate data acquisition of magnetic resonance imaging (MRI). With the development of compressed sensing (CS) theory, many solutions have been proposed for undersampled data reconstruction. Moreover, dictionary learning method has shown good results in improving reconstruction quality. However, CS reconstruction algorithms are time consuming, especially at dictionary training and sparse coding step. The computation overhead is even higher for three-dimensional reconstruction. With a large number of slices, data size can be massive and more time consuming. In this paper, we use three-dimensional dual-dictionary learning scheme for the reconstruction procedure. Three-dimensional dictionaries train the dictionary atoms in image blocks and utilize spatial correlation among MR slices. Dual-dictionary learning method cooperates low-resolution dictionary and high-resolution dictionary for sparse coding and image updating, respectively. Compute unified device architecture (CUDA) is utilized to design the parallel algorithms on graphics processing unit (GPU). We mainly optimize dictionary learning algorithm and image updating. We also develop parallel CPU codes using OpenMP (Open Multi-Processing) and another version of CUDA codes with algorithmic optimization. Experimental results show that more than 324 times of speedup is achieved compared with CPU-only codes with 24 MRI slices and more than 40 times of acceleration compared with OpenMP codes.

Keywords Dual-dictionary learning · Compressed sensing · Three-dimensional MRI reconstruction · CUDA · GPU

J. Li · J. Sun · Y. Song · J. Zhao (✉)
School of Biomedical Engineering, Shanghai Jiao Tong University, Shanghai, China
e-mail: junzhao@sjtu.edu.cn

1 Introduction

Magnetic resonance imaging (MRI) modality is safe and efficient. This method has no radiation damages to patients and can achieve tomography in any direction with high soft-tissue contrast. Therefore, MRI has been widely used clinically. However, conventional MRI has a long sampling time, which may lead to patient discomfort and motion artifacts in the reconstruction images. Hence, the use of MRI may be limited in various areas, such as cardiac imaging and functional MRI.

One solution is to perform reconstruction from undersampled k-space data. This procedure can improve data acquisition speed by sampling less data. Compressed sensing (CS) theory [6,7] suggests that a sparse signal can be reconstructed from its sparse representation under certain conditions. Therefore, reconstruction of MRI images can be achieved using undersampled k-space data. CS theory has been proved to have high quality reconstructions from undersampled measurements [9,14,24].

Dictionary learning method [8,18,21] is an extremely effective way to establish adaptive dictionaries with good sparsity. The dictionary can be used to train sparse representations of signals and reconstruct images. In our work, we utilize k-singular value decomposition (K-SVD) [1] and orthogonal matching pursuit (OMP) [5,22,23] algorithm to train dictionaries and obtain sparse representations.

Ying Song et al. [18] propose a new algorithm for MRI reconstruction using undersampled k-space data. They use three-dimensional dictionaries and perform reconstruction of multi-slice MRI images. In their method, the three-dimensional data are divided into blocks. Hence, spatial correlation among slices can be used as prior knowledge when training dictionaries and updating result images. Furthermore, they adopt a new dictionary learning scheme,dual-dictionary learning, with low-resolution dictionary D^{low} and high-resolution dictionary D^{high} for sparse coding and image updating, respectively. Their work indicates that dual-dictionary learning scheme is better than single dictionary learning scheme. In the image updating stage, replacing D^{low} with D^{high} results in significantly improved outputs. Consequently, the number of loops is reduced and procedure speeds up.

However, three-dimensional reconstruction copes with massive amount of data, which will increase with the increase in the number of slices. In addition, K-SVD and OMP are both iterative algorithms which are computationally expensive. The situation will become even worse when the amount of data increase. Therefore, accelerating the reconstruction procedure is needed. With the development of cheap hardware and parallel development tools, graphics processing unit (GPU)-based applications are widely used and are experiencing rapid development. References [2–4,16] present how hardware and software can work in concert on scalable multi-processor systems with a number of illustrative examples and applications. In our work, we first design parallel algorithm on GPU directly under the scheme of compute unified device architecture (CUDA) [12] (we call this version of CUDA code as "original CUDA"). This design utilizes GPU's strong computing power and high performance in parallel computing. Then, we carry out algorithmic optimization proposed in [15], which will serve as basis for the development of another version of CUDA code (called CUDA after Algorithmic

Optimization, i.e., CUDA-AO). In both versions of CUDA codes, we emphasize the optimization and parallelization of K-SVD and OMP algorithm. We also implement the parallel version codes on CPU using open multi-processing (OpenMP) under nearly the same parallelization mechanism for CUDA to further verify the efficiency of our CUDA codes.

GPU has experienced staggering development in recent years. GPU exhibits strong advantages, including having a stronger computing power than CPU, being cheaper than CPU but having the same computing load, and natural parallelism. These features have contributed to its rapid growth and widely use in scientific computing and engineering fields. With the constant development of technology, the demand for real-time and high-resolution 3D imaging techniques has been increasingly growing. However, the deployment of advanced technologies is limited by processing time. GPU-based MR image reconstruction acceleration has received considerable research attention [10,17,19,20]. In our work, we focus on a recent innovation in MRI reconstruction technique and design a new program that is applicable to GPU with CUDA architecture. With this approach, we are able to achieve large acceleration ratios.

CUDA is a programming model and general purpose parallel computing platform introduced by NVIDIA Corporation in November 2006. CUDA can utilize the parallel compute engines in NVIDIA GPUs to solve complex and tremendous computational tasks. This model allows programmers to easily develop programs on GPU without much knowledge of the GPU internal structure and parallelization mechanism of computing in threads. In CUDA, the smallest execution unit is called *thread*. Many threads are grouped into a *block*. Numerous blocks together form a *grid*. The threads in the same block can access the same *shared memory*, as well as be controlled to be synchronized. CUDA codes consist of the following two parts: *host* part is executed on CPU and *device* part is executed on GPU. The program should manage data transformation between CPU memory and GPU memory, which has relatively long time cost when processing small-scale data. The powerful computing capability of CUDA allows this model to be increasingly used in scientific computing areas. In our work, CUDA runtime API and CUDA Basic Linear Algebra Subroutines (CUBLAS) library [11], which is a GPU-accelerated version of the complete standard BLAS library, are used to develop CUDA programs.

The rest of this paper is organized as follows: Sect. 2 presents the reconstruction algorithm scheme for the three-dimensional MRI reconstruction. Section 3 introduces the parallel implementations on CUDA, including OMP, K-SVD, and reconstruction procedure. Section 4 shows the performance of the original CUDA method and CUDA-AO method. Section 5 draws the conclusions.

2 The reconstruction algorithm scheme

MRI datasets are divided into small blocks for three-dimensional reconstruction. Then, data are rearranged for further processing. The formulation of the multi-slice MRI reconstruction [18] is shown in (1).

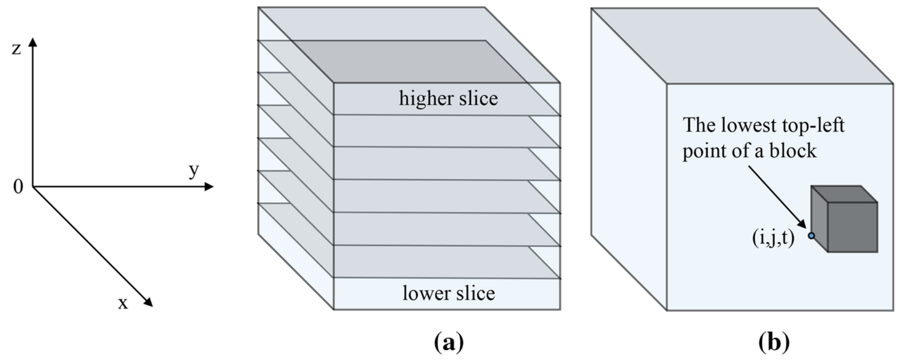


Fig. 1 **a** Three-dimensional MRI series volume. Multi-slice MR images are concerned. **b** Multi-slice MR images are divided into blocks. The *lowest top-left* point of a block is shown

$$\begin{aligned}&\min_{\mathbf{x},\boldsymbol{\alpha}_{i,j,t}} \sum_{i,j,t} \|\mathbf{R}_{i,j,t}\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}_{i,j,t}\|_2^2 + \nu \|\mathbf{F}_u\mathbf{x} - \mathbf{y}\|_2^2 \\ &\text{s.t.} \ \ \|\boldsymbol{\alpha}_{i,j,t}\|_0 \leqslant \rho \quad \forall i, j, t\end{aligned} \tag{1}$$

where $\mathbf{x}$ is the unknown image to be reconstructed, $\mathbf{R}_{i,j,t}$ is the extraction operator, indexed by the location of the lowest top-left point, (i, j, t), in the image block, as shown in Fig. 1. $\boldsymbol{\alpha}_{i,j,t}$ is the sparse representation of $\mathbf{x}$ under dictionary $\mathbf{D}$, $\|\cdot\|_0$ is the l_0-norm which counts the number of nonzero elements. $\mathbf{F}_u$ is the undersampling Fourier matrix, $\mathbf{y}$ is the undersampled k-space measurements, ρ is the sparsity level, and ν is defined by $\nu = \lambda/\sigma$, where λ is a positive constant and σ is the standard deviation of the noise.

The first term in (1) restrains the quality of sparse approximations of the image blocks with respect to dictionary $\mathbf{D}$. The second term controls the data fidelity of the reconstruction. The reconstruction scheme consists of the following three steps:

1. *Dictionary learning step* K-SVD algorithm is used to train dual dictionaries D^{low} and D^{high}.
2. *Sparse coding step* The image $\mathbf{x}$ is assumed to be fixed and dictionary D^{low} is used to obtain the sparse representation $\boldsymbol{\alpha}$.
3. *Image updating step* D^{low} is replaced by D^{high}. $\boldsymbol{\alpha}$, D^{high} are used to reconstruct and update the final images.

At the *dictionary learning step*, we use OMP algorithm in K-SVD [1] to compute the representation vectors for each column of signal $\mathbf{x}$. At the *sparse coding step*, OMP is also utilized to obtain the sparse representations with respect to the low-resolution dictionary [18]. In addition, the time used by OMP accounts for the largest proportion of total time consumed by the reconstruction procedure. K-SVD algorithm is efficient in training adaptive dictionaries [1]. We use this algorithm to train both high-resolution and low-resolution dictionaries in our reconstruction scheme. However, K-SVD is an extremely computationally expensive iterative method. Later in this paper (please see Table 2), we can see that the execution time of K-SVD is typically more than 91 %

of the total time used for reconstruction in original CPU version codes. OMP and K-SVD are the most time-consuming parts. Hence, we focus on these parts, as well as develop parallel version algorithms and codes on CUDA, to accelerate the whole reconstruction process.

3 Parallel implementations on CUDA

3.1 CUDA implementation of OMP

OMP is an attractive sparse signal recovery algorithm that can achieve good performance and is easier to implement. OMP is a kind of greedy algorithm. This algorithm chooses an atom from the dictionary at every step. The atom chosen should be the closest to the residual signal and has the largest inner product. Then, the signal is orthogonally projected to the span of selected atoms for approximation. Numerous matrix/vector operations are found in OMP algorithm, which are convenient and easy to implement in a parallel manner on GPU. Signal $\mathbf{x}$ is a matrix with columns that correspond to the blocks shown in Fig. 1. Data in one block are rearranged to form a column vector and then stored in one column of $\mathbf{x}$. Each column in $\mathbf{x}$ needs a loop cycle.

OMP algorithm uses iterative mechanism. In one circle of iteration, one column of the signal $\mathbf{x}$ is manipulated to obtain the corresponding sparse representation. In the original CUDA method, we simply parallelize the codes inside the iteration in OMP algorithm. We then develop parallel implementations of matrix/vector operations to accelerate the codes. Detailed CUDA version OMP algorithm is shown in Algorithm 1. Notably, "$\mathbf{0}$" represents empty set. For a index set $\mathbf{I} = (i_1, i_2, \ldots, i_n)$, $\boldsymbol{\alpha}_{\mathbf{I}}$ indicates the sub-matrix of $\boldsymbol{\alpha}$, which consists of the columns indexed by $\mathbf{I}$, in the order of $i_1, i_2, \ldots, i_n$.

One important part of OMP is to calculate the pseudo-inverse of matrices. This procedure is time consuming when matrices are large. Rubinstein et al. [15] introduce a method that use Cholesky factorization to avoid computation of pseudo-inverse of matrices in OMP algorithm. We use this method to first optimize the codes algorithmically and then transfer the codes to CUDA. The modified OMP algorithm is more suitable for parallel implementation on CUDA.

Algorithm 1 Orthogonal Matching Pursuit (OMP) implemented on CUDA

```
Initialization and Input:
    $\mathbf{D}$: dictionary. $\rho$: sparse level. $\boldsymbol{\alpha}$: sparse representation. $\mathbf{x}$: signal.
    $\mathbf{r}$: residual signal. $\boldsymbol{\Lambda}$: selected atoms. $d_j$: the $j^{th}$ column of $\mathbf{D}$.
    $\mathbf{I}$: a set that contains the index of all the columns selected from $\mathbf{D}$.
    Set $\boldsymbol{\alpha} = \mathbf{0}, \boldsymbol{\Lambda} = \mathbf{0}, \mathbf{I} = \mathbf{0}$.
Main Procedure:
 1: procedure OMP_KERNEL
 2:     int tid = blockDim.x * blockIdx.x + threadIdx.x
 3:     if tid is less than the number of columns of signal $\mathbf{x}$ then
 4:         Compute the increasing factor for each intermediate variable.
 5:         $\mathbf{x}_c \leftarrow$ the tid$^{th}$column of $\mathbf{x}$
 6:         $\mathbf{r} \leftarrow \mathbf{x}_c$
 7:         while sparse level $\rho$ is not reached do
 8:             $j \leftarrow \arg\max_i |d_i^T \mathbf{r}|$
 9:             $\mathbf{I} \leftarrow (\mathbf{I}, j)$
10:             $\boldsymbol{\Lambda} \leftarrow (\boldsymbol{\Lambda}, d_j)$
11:             $\boldsymbol{\alpha}_{\mathbf{I}} \leftarrow \arg\min_{\boldsymbol{\alpha}} \|\mathbf{x} - \boldsymbol{\Lambda}\boldsymbol{\alpha}\|_2^2$
12:             $\mathbf{r} \leftarrow \mathbf{x}_c - \boldsymbol{\Lambda}\boldsymbol{\alpha}_{\mathbf{I}}$
13:         end while
14:     end if
15: end procedure
Output: $\boldsymbol{\alpha}$
```

Data need to be divided into many small blocks for three-dimensional MRI reconstruction. For instance, for a $4 \times 4 \times 4$ block, all blocks are rearranged together by adding one column to $\mathbf{x}$ for each block. Thus, the number of columns of $\mathbf{x}$ may be extremely large when the slice number of the MRI data becomes larger. In the original CPU version OMP algorithm, each column of $\mathbf{x}$ requires one circle of iteration for execution. Moreover, large number of loops are necessary and consume much time. In CUDA, we can assign one thread to execute the operations inside one loop. Numerous threads can work concurrently to simultaneously compute the results. The number of threads allocated for computation is the same with the column number of $\mathbf{x}$. We arrange tens of thousands of threads to compute the OMP_kernel procedure in Algorithm 1. The iteration can be eliminated. In addition, all operations for each column of $\mathbf{x}$ can run simultaneously. The iteration inside the manipulation of each column of $\mathbf{x}$ for reaching the sparse level still exists, which can ensure the convergence for each signal component.

In the CUDA version OMP algorithm, numerous intermediate variables should be allocated space and initialized on GPU memory at the beginning. Numerous threads run simultaneously and every thread needs to visit all intermediate variables to avoid data conflict among threads. Hence, we must enlarge the size of all intermediate variables according to the thread number running at one time. For instance, if a single thread requires an intermediate variable with size 1,000 in C++ *float* type and 2,000 threads run simultaneously, then this intermediate variable should be allocated a memory space of 1,000 $\times$ 2,000 at the initialization stage. Each thread can visit the corresponding memory indexed by their *tid* value by adding the increasing factor of the address pointer for each intermediate variable, as shown in Algorithm 1, line 2. Furthermore, intermediate variables that remain unchanged during the execution of OMP can be stored in the shared memory to further speedup the procedure. All functions called inside the OMP_kernel procedure should be modified properly and defined by the __*device*__ declaration specifier, which, like __*global*__, is part of the C extensions of CUDA [12].

3.2 CUDA implementation of K-SVD

K-SVD algorithm is utilized to train both low-resolution dictionary and high-resolution dictionary. K-SVD updates only one column of the dictionary at each circle of iteration, as well as involves only the signals that use the current atom. The detail of K-SVD is described in Reference [1]. K-SVD consists of two steps, namely, sparse coding stage that uses OMP algorithm to compute the sparse representation vectors of each column of $\mathbf{x}$ with a fixed dictionary $\mathbf{D}$ and codebook update stage that analyzes data in the sparse representation $\boldsymbol{\alpha}$ achieved in the sparse coding stage, then computes the representation error matrix and applies a singular value decomposition (SVD) to update the dictionary atom and corresponding row in $\boldsymbol{\alpha}$.

CUDA implementation of K-SVD is shown in Fig. 2. We adopt the CUDA version OMP method in Algorithm 1 to parallelize the sparse coding stage. The codebook update stage requires one circle of loop for each column of $\mathbf{D}$. In each loop, the current dictionary atom and corresponding sparse representation vector are updated. The updated $\mathbf{D}$ and $\boldsymbol{\alpha}$ are used in the next loop. Hence, one circle of iteration depends on the results of the previous circle. Therefore, the iteration cannot be eliminated by simply assigning many CUDA threads to run simultaneously. However, we can still accelerate K-SVD procedure by parallelizing the codes inside the loops.

First, we analyze the data in $\boldsymbol{\alpha}$ by obtaining the indices of items that use the current dictionary atom. We denote the kth column of $\mathbf{D}$ as $\mathbf{d}_k$, and set $\mathbf{d}_k$ the current dictionary atom being trained. Then, we should obtain the item indices in the kth row of $\boldsymbol{\alpha}$ that use $\mathbf{d}_k$, i.e., whose values are nonzero. In CUDA, we can deploy as many threads as the number of items in $\boldsymbol{\alpha}$ to traverse the data and simultaneously obtain the results. This procedure can be implemented before the iteration if we keep track of the row indices of $\boldsymbol{\alpha}$, as shown in Fig. 2. Second, computation of the representation error matrix merely consists of matrix/vector operations. Moreover, this computation is straightforward manner of parallelizing this procedure on GPU. In addition, we use CULA [13] to perform singular value decomposition of the error matrix, which is an implementation

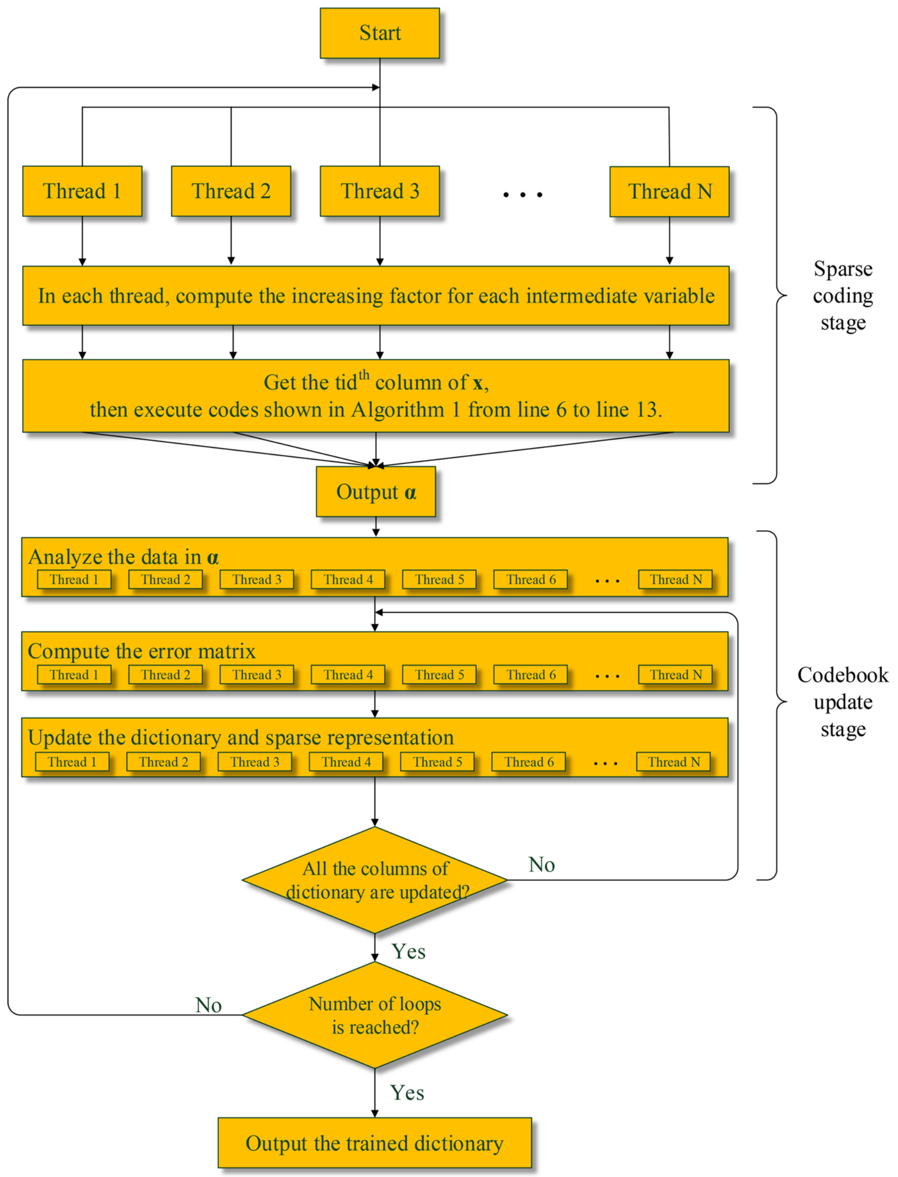

Start
Thread 1
Thread 2
Thread 3
. . .
Thread N
In each thread, compute the increasing factor for each intermediate variable
Get the tidth column of x,
then execute codes shown in Algorithm 1 from line 6 to line 13.
Output α
Sparse coding stage
Analyze the data in α
Thread 1 Thread 2 Thread 3 Thread 4 Thread 5 Thread 6 . . . Thread N
Compute the error matrix
Thread 1 Thread 2 Thread 3 Thread 4 Thread 5 Thread 6 . . . Thread N
Update the dictionary and sparse representation
Thread 1 Thread 2 Thread 3 Thread 4 Thread 5 Thread 6 . . . Thread N
Codebook update stage
All the columns of dictionary are updated?
No
Yes
Number of loops is reached?
No
Yes
Output the trained dictionary

Fig. 2 K-SVD algorithm implemented on CUDA

of the linear algebra package interface for CUDA-enabled GPUs. We also employ CUBLAS library to accelerate the execution of several linear algebra operations.

SVD operation in the codebook update stage demands considerable amount of computation. This operation is regarded as time consuming. Rubinstein et al. [15] develop an approximate K-SVD implementation that avoids SVD operation. We utilize their method to optimize K-SVD algorithmically and then develop CUDA version codes of K-SVD, which is part of CUDA-AO method.

When implementing K-SVD algorithm on CUDA, numerous intermediate results can be stored on the GPU memory and then used for other GPU computations. For instance, we can use the data for error matrix computation after obtaining the item indices in $\boldsymbol{\alpha}$ that use the current dictionary atom. Both operations executed on GPU and data are stored on GPU memory. Hence, we need not to transfer data between GPU memory and CPU memory, which requires a relatively long time and degrades the performance of our codes.

3.3 Image reconstruction using CUDA

Both low-resolution dictionary and high-resolution dictionary are trained by K-SVD algorithm implemented on CUDA, as we have mentioned in Sect. 1. To update the final results, we employ the method proposed in [18] and rewrite the codes using CUDA. First, we use the low-resolution dictionary D^{low} to obtain the sparse representation $\boldsymbol{\alpha}$ of MRI datasets to be rebuilt. Then, we reconstruct and update the results using D^{high} and $\boldsymbol{\alpha}$. The detailed reconstruction procedure is shown in Fig. 3.

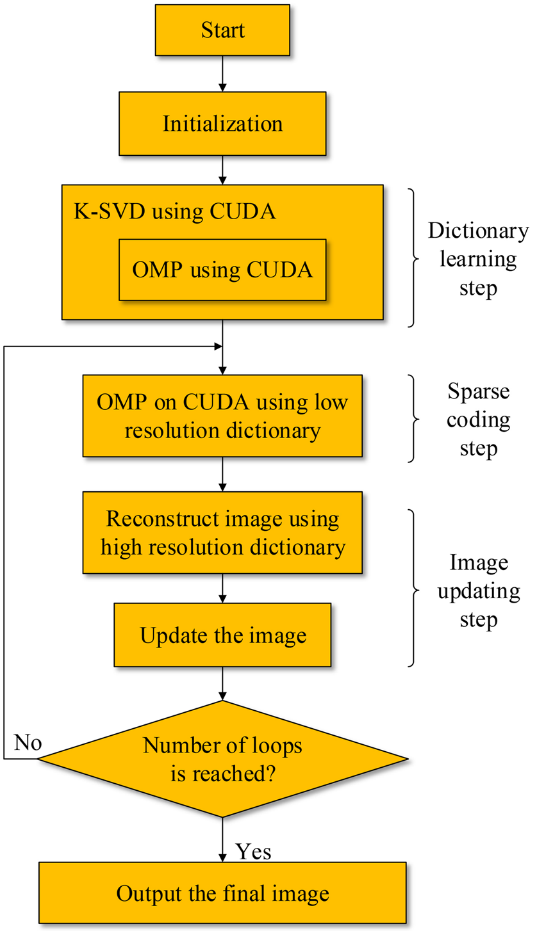
The reconstruction process requires a few loops to acquire better results, and the result of one loop iteration is further updated by the following loops. So again, the iteration cannot be eliminated using CUDA threads parallelization technique. But luckily, we only need very few iterations [18], just less than 10.

At the sparse coding step, the CUDA version OMP algorithm described in Sect. 3.1 is utilized. We can divide the datasets into many parts to obtain better parallelization effect. Every part is processed simultaneously with every CUDA thread running an OMP_kernel procedure. The results of each thread are integrated together to form the final results. If the GPU memory is insufficient to meet the requirement, a compromise solution is to execute the process in a few loops. An example would be if 150 numbers needed be processed, but the GPU memory is capable of dealing with no more than 100 numbers. Clearly, not all data can be processed at the same time. Thus, we divide the 150 numbers into two parts, dealing with only 75 numbers once. The task will be finished within two circles of loops. For implementation, the GPU's memory consists of 3,072 megabytes (MB). At the sparse coding step, dictionary size is set to 600. For a $256 \times 256 \times 28$ MR dataset, after it is rearranged by $4 \times 4 \times 4$ block, a new matrix with a size of $64 \times 1{,}835{,}008$ is obtained, and the size of the sparse representation coefficient matrix is $600 \times 1{,}835{,}008$, which requires more than 4,400 MB of memory for float type data. The memory requirement will be even larger if other intermediate variables are considered. Obviously, the GPU memory cannot meet this demand. This problem can be overcome by executing the process in a few loops. If the number of loops is set to 4, only $1{,}835{,}008/4 = 458{,}752$ columns (requiring approximately 1,100 MB of memory) need to be processed in each loop, and the memory requirement is reduced to a quarter of the original. In this case, the GPU memory can handle the requirement.

4 Results

We use CUDA 5.5 to develop our programs on GPU. All computations are performed on a 64-bit computer with Intel Xeon E5640 2.67 GHz CPU and NVIDIA GeForce

Fig. 3 Image reconstruction using CUDA



GTX 780 TI GPU under Windows Server 2008 R2 operating system. The experimental computer has two packaged-together Xeon E5640 2.67 GHz CPUs, thus it has 8 cores and 16 threads. All codes are written with C++ and CUDA C language with Microsoft Visual Studio 2010 integrated development environment (IDE). Our host PC has 48 gigabytes (GB) of memory, whereas the memory size of GPU is 3,072 MB. We test the bandwidth of GPU using the sample utility “bandwidthTest” in the NVIDIA GPU Computing Toolkit. The result shows host-to-device bandwidth is 4.25 GB/s, the device-to-host bandwidth is 5.59 GB/s, and the device-to-device bandwidth is 224.34 GB/s, as shown in Fig. 4. All codes are compiled in 64-bit version within Microsoft Visual Studio 2010 under release mode. Complier options are set to maximize speed, enabling intrinsic functions and enabling function-level linking. We mainly use the NVIDIA Visual Profiler to record the running time of the codes. Timing operations on the host are also used.

We test the acceleration results with different numbers of slices on MRI datasets. Table 1 shows the time consumed by OMP algorithm at sparse coding step and its

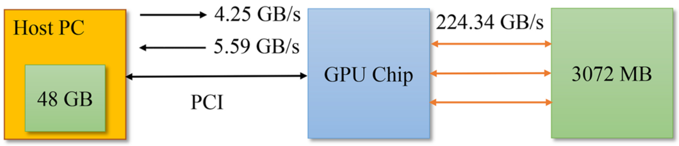


Fig. 4 Bandwidth of NVIDIA GeForce GTX 780 TI

Table 1 Execution time and speedup of OMP algorithm at sparse coding step

Number of slices	OMP executioin time (s)				Speedup			
	Original CPU	OpenMP	Original CUDA	CUDA-AO	Original CUDA		CUDV-AO	
					Original CPU	OpenMP	Original CPU	OpenMP
4	38.97	36.45	35.26	7.63	1.11	1.03	5.11	4.78
8	57.65	50.76	62.91	14.99	0.92	0.81	3.85	3.39
12	120.94	134.53	89.66	21.43	1.35	1.50	5.64	6.28
16	565.53	144.07	153.64	32.09	3.68	0.94	17.62	4.49
20	629.65	193.07	160.13	32.67	3.93	1.21	19.28	5.91
24	738.13	274.23	176.44	38.02	4.18	1.55	19.42	7.21
28	1,299.99	318.51	221.69	47.07	5.86	1.44	27.62	6.77

speedup (Fig. 5a shows a more intuitional view). When implemented on CUDA, numerous threads work simultaneously and the time consumed is reduced. The amount of data and the time consumed by CPU codes increase rapidly if the number of MRI slices become larger. The time consumed by OMP increases from 38.97s to 1,299.99s when the number of slices changes from 4 to 28. However, Original CUDA method needs 221.69s and CUDA-AO method needs only 47.07s with 28 MRI slices. This indicates that CUDA can greatly accelerate the procedure. CUDA-AO method avoids calculating the pseudo-inverse of matrices and is more suitable for parallelization. Hence, CUDA-AO method achieves better acceleration effect. However, the speedup obtained by OMP is not so significant when compared with K-SVD (Table 2), because the iteration for reaching the sparse level still exists to ensure the convergence for each signal component and this will increase the time overhead.

Table 2 shows the execution time of K-SVD algorithm in original CUDA method and CUDA-AO method. The results indicate great acceleration. After algorithmic optimization, SVD is avoided and the method is more suitable for parallelization using CUDA. Hence, CUDA-AO method achieves much better acceleration effect than original CUDA method. We can see that when the slice number is 24, original CUDA method can obtain approximately 305 times of speedup, while approximately 1,912 times of speedup can be achieved if algorithmic optimization is implemented. As shown in Fig. 5b, the speedup increases significantly with the increase in the number of MRI slices. The speedup is mostly obtained in the codebook update stage shown in

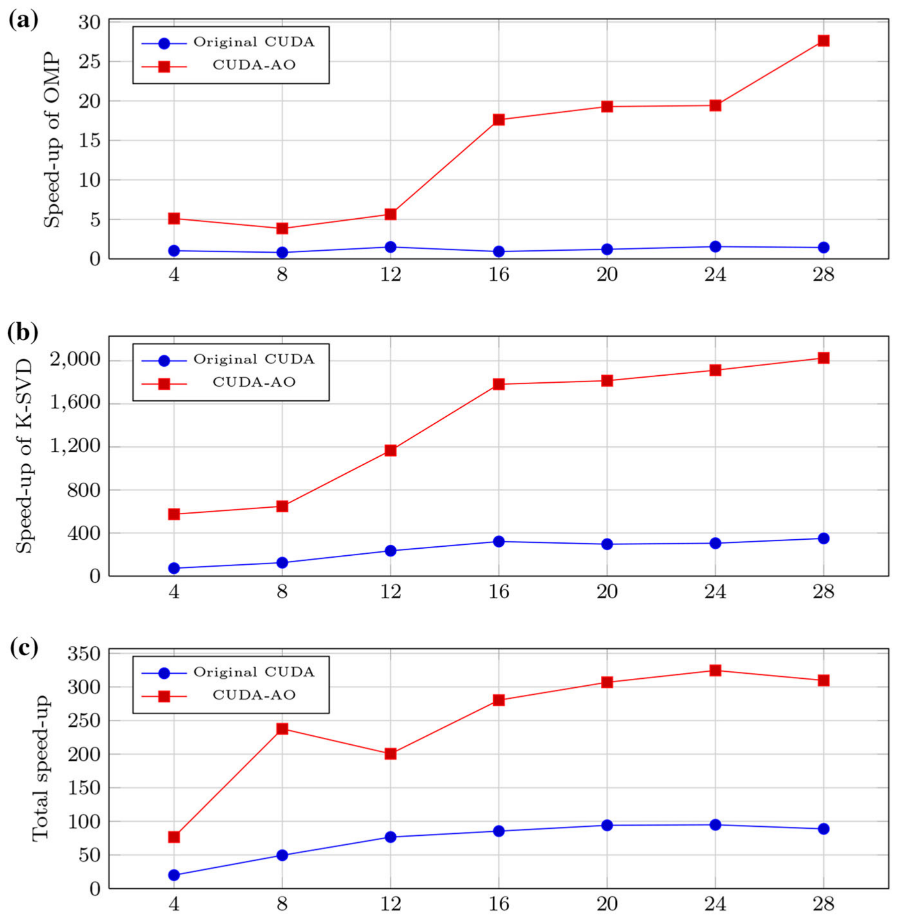


Fig. 5 Speedup with different numbers of MRI slices versus original CPU codes: **a** OMP algorithm at sparse coding step; **b** K-SVD algorithm and **c** total reconstruction procedure

Fig. 2. We deploy many CUDA threads to traverse the data and simultaneously obtain the results. Many intermediate variables can be stored in GPU memory for further use and need not to be transferred to CPU memory. This design dramatically reduces the time consumed.

Table 3 shows the execution time of original CPU codes, original CUDA method, and CUDA-AO method. We can see that with the increase in the number of MRI slices, the speedup becomes larger, as shown in Fig. 5c. The reconstruction procedure of MRI can obtain more than 20 times of speedup using original CUDA method. The acceleration effect is even better when using CUDA-AO method, which holds approximately 324 times of speedup when the number of slices is 24. The results indicate that both original CUDA method and CUDA-AO method can achieve good effect of speedup. Moreover, CUDA-AO method is better. Hence, optimizing the method algorithmically makes the technique more suitable for parallelization on GPU.

Table 2 Execution time and speedup of K-SVD algorithm

Number of slices	K-SVD execution time (s)				Speedup			
	Original CPU	OpenMP	Original CUDA	CUDA-AO	Original CUDA		CUDA-AO	
					Original CPU	OpenMP	Original CPU	OpenMP
4	1,408.11	181.67	19.26	2.45	73.12	9.43	575	74.15
8	7,160.53	918.166	57.47	11.05	124.60	15.98	648	83.09
12	14,180.30	1,134.51	60.27	12.15	235.30	18.82	1,167	93.38
16	25,219.81	2,779.86	78.47	14.15	321.38	35.43	1,782	196.46
20	26,675.44	3,722.44	89.88	14.70	296.78	41.43	1,815	253.23
24	30,640.20	4,421.37	100.37	16.02	305.28	44.05	1,913	275.99
28	37,329.18	5,768.48	106.67	18.42	349.96	54.08	2,026	313.16

Table 3 Execution time and speedup obtained with original CUDA method and CUDA-AO method

Number of slices	Total execution time (s)				Speedup			
	Original CPU	OpenMP	Original CUDA	CUDA- AO	Original CUDA		CUDA-AO	
					Original CPU	OpenMP	Original CPU	OpenMP
4	1,466.73	232.987	73.275	19.14	20.02	3.18	76.63	12.17
8	7,251.59	1,030.96	146.44	45.13	49.52	7.04	237.64	22.84
12	14,361.8	1,320.17	187.35	71.62	76.66	7.05	200.53	18.43
16	25,952.9	3,016.09	303.65	92.59	85.47	9.93	280.31	32.57
20	29,214.5	4,001.44	310.51	95.19	94.09	12.89	306.91	42.04
24	35,789.8	4,833.13	377.33	110.29	94.85	12.81	324.51	43.82
28	38,833.6	6,049.65	437.37	125.38	88.79	13.83	309.73	48.25

CUDA-AO method exhibits superior qualities over OpenMP codes, particularly when the MR data volume is large, as shown in Fig. 6. Although OpenMP codes utilize nearly the same parallelization mechanism as CUDA-AO, the performance of the former is limited by the computing capability of CPU and is therefore much less effective than CUDA-AO method. When the MR slice number is 28, we can see that more than 300 times of speedup is acquired by CUDA-AO method versus OpenMP for the K-SVD algorithm. Meanwhile, the total reconstruction time of CUDA-AO is approximately 1/48 of that of OpenMP codes.

For CUDA-AO method, when the number of MRI slices is small, for instance, 4 or 8, the speedup line is low and the acceleration effect is relatively not obvious, as shown in Fig. 5. With 4 or 8 MRI slices, the amount of data is not overly large and CPU has strong computing power. Therefore, the performance gap between CPU and

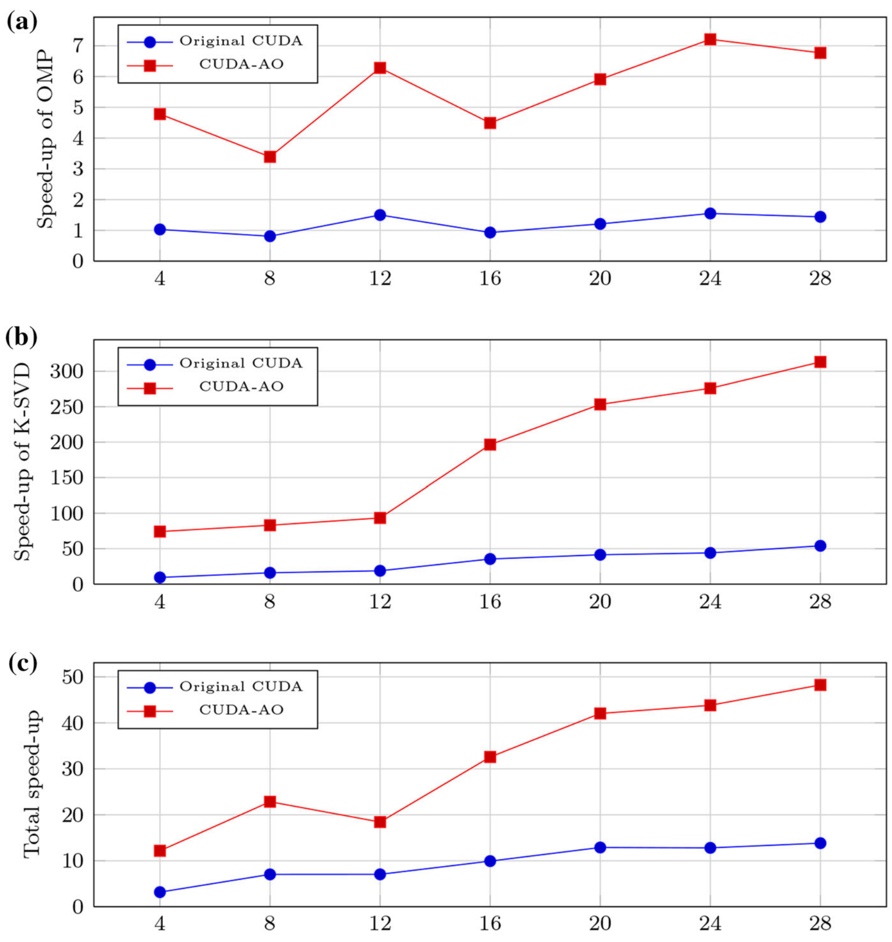


Fig. 6 Speedup with different numbers of MRI slices versus OpenMP codes: **a** OMP algorithm at sparse coding step; **b** K-SVD algorithm and **c** total reconstruction procedure

GPU is not substantial. However, when the amount of data is increased to a certain degree, for instance, 16 MRI slices, the speed-up line becomes rather high and the acceleration effect becomes extremely obvious. These findings indicate that CUDA is more capable of dealing with massive data and huge computation. If the number of slices is further increased, the acceleration effect would be limited by the performance of the graphics card and the increase in the speed-up line would be insignificant. The data transfer overheads between GPU and CPU can be easily recorded using NVIDIA Visual Profiler. The GPU–CPU time for CUDA-AO method is shown in Table 4, which indicates that the time consumed by GPU–CPU data transfer is relatively short, and that time gradually increases as MRI data volumes increase. Note that with less data, the GPU–CPU data transfer time accounts for a relatively large ratio of the total reconstruction time. This ratio decreases as the data volume becomes larger.

Table 4 Data transfer time (seconds) between GPU and CPU for CUDA-AO method

Number of slices	4	8	12	16	20	24	28
Device-to-host	0.303	0.623	0.878	1.181	1.484	1.765	2.052
Host-to-device	1.140	1.397	1.654	1.944	2.187	2.459	2.748
GPU–CPU time	1.443	2.020	2.532	3.125	3.671	4.224	4.800
Total time	19.141	45.131	71.619	92.586	95.191	110.292	125.377
Ratio (%)	7.539	4.476	3.535	3.375	3.856	3.83	3.828

5 Conclusion

In this work, we accelerate the reconstruction of MRI by three-dimensional dual-dictionary learning using CUDA. The parallel algorithms on GPU and acceleration performance are investigated. We develop the following two versions of CUDA codes: (1) original CUDA method that directly transfer original CPU codes to CUDA; (2) CUDA-AO method that first improves the original CPU codes with algorithmic optimization, then implements the codes on CUDA. Parallel codes on CPU are also developed using OpenMP. Experiments show that approximately 94 times of speedup is achieved using original CUDA method when the number of slices is 24, while roughly 324 times of speedup is obtained with CUDA-AO method. When compared with OpenMP, CUDA-AO method can acquire more than 40 times of speedup.

Our methods have achieved great success for the K-SVD algorithm. This indicates that for some iterative algorithm, when the iteration can't be eliminated, we can still acquire acceleration by parallelizing the operations inside the loops. Plenty of CUDA threads run simultaneously for the OMP algorithm. Moreover, each thread implements a single OMP procedure individually. Our CUDA implementations extraordinarily reduce the time consumed by the total reconstruction procedure.

This work shows that CUDA and algorithmic optimization offer great advantages in accelerating MRI reconstruction. This is an effective way to put long time-consuming algorithms to practical and clinical use.

Acknowledgments This work is supported by National Natural Science Foundation of China (No. 813716234), National Basic Research Program of China (2010CB834302), and Shanghai Jiao Tong University Medical Engineering Cross Research Funds (YG2013MS30 and YG2011MS51).

References

1. Aharon M, Elad M, Bruckstein A (2006) K-svd: an algorithm for designing overcomplete dictionaries for sparse representation. Signal Process IEEE Trans 54(11):4311–4322
2. Arabnia HR (1996) Distributed stereo-correlation algorithm. Comput Commun 19(8):707–711
3. Arabnia HR, Oliver MA (1987) A transputer network for the arbitrary rotation of digitised images. Comput J 30(5):425–432
4. Arabnia HR, Oliver MA (1989) A transputer network for fast operations on digitised images. In: Computer graphics forum, vol. 8. Blackwell Publishing Ltd, pp 3–11
5. Davis G, Mallat S, Avellaneda M (1997) Adaptive greedy approximations. Constr Approx 13(1):57–98
6. Donoho DL (2006) Compressed sensing. Inf Theory IEEE Trans 52(4):1289–1306

7. Van de Gronde J, Vuçini E (2008) Compressed sensing overview. http://www.cg.tuwien.ac.at/research/publications/2008/Gronde_2008/Gronde_2008-report.pdf
8. Kreutz-Delgado K, Murray JF, Rao BD, Engan K, Lee TW, Sejnowski TJ (2003) Dictionary learning algorithms for sparse representation. Neural Comput 15(2):349–396
9. Lustig M, Donoho D, Pauly JM (2007) Sparse mri: The application of compressed sensing for rapid mr imaging. Magn Resonance Med 58(6):1182–1195
10. Murphy M, Keutzer K, Vasanawala S, Lustig M (2010) Clinically feasible reconstruction time for l1-spirit parallel imaging and compressed sensing mri. In: Proceedings of the ISMRM Scientific Meeting & Exhibition, p 4854
11. Nvidia: Cublas library (2012). http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf
12. Nvidia: Cuda c programming guide (2012). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
13. Photonics, E.: Cula programmer's guide release r16a (2012). http://www.culatools.com/files/docs/R16a/CULAReferenceManual_R16a.pdf
14. Qiu C, Lu W, Vaswani N (2009) Real-time dynamic mr image reconstruction using kalman filtered compressed sensing. In: Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on, IEEE pp 393–396
15. Rubinstein R, Zibulevsky M, Elad M (2008) Efficient implementation of the k-svd algorithm using batch orthogonal matching pursuit. CS Technion p 40
16. Shahin M, Tollner E, Evans M, Arabnia H (1999) Watercore features for sorting red delicious apples: a statistical approach. Trans ASAE 42(6):1889–1896
17. Smith DS, Gore JC, Yankeelov TE, Welch EB (2012) Real-time compressive sensing MRI reconstruction using GPU computing and split Bregman methods. Int J Biomed Imaging 2012:1–6
18. Song Y, Zhu Z, Lu Y, Liu Q, Zhao J (2014) Reconstruction of magnetic resonance imaging by three-dimensional dual-dictionary learning. Magn Resonance Med 71(2):1285–1298
19. Sorensen TS, Prieto C, Atkinson D, Hansen MS, Schaeffter T (2010) Gpu accelerated iterative sense reconstruction of radial phase encoded whole-heart mri. In: Proceedings of the 18th scientific meeting, international society for magnetic resonance in medicine. Stockholm, vol. 2869
20. Stone SS, Haldar JP, Tsao SC, Hwu WM, Sutton BP, Liang ZP et al (2008) Accelerating advanced mri reconstructions on gpus. J Parallel Distrib Comput 68(10):1307–1318
21. Tosic I, Frossard P (2011) Dictionary learning. Signal Process Mag IEEE 28(2):27–38
22. Tropp JA (2004) Greed is good: algorithmic results for sparse approximation. Inf Theory IEEE Trans 50(10):2231–2242
23. Tropp JA, Gilbert AC (2007) Signal recovery from random measurements via orthogonal matching pursuit. Inf Theory IEEE Trans 53(12):4655–4666
24. Trzasko J, Manduca A (2009) Highly undersampled magnetic resonance image reconstruction via homotopic-minimization. Med Imaging IEEE Trans 28(1):106–121