Article

Efficient Tensor Sensing For RF Tomographic Imaging on GPUs

Da Xu [1] and Tao Zhang [1,2*]

[1] Department of Computer Engineering and Science, Shanghai University, Shanghai, China 200444; mr_tab@shu.edu.cn

[2] Shanghai Institute for Advanced Communication and Data Science, Shanghai, China;

* Correspondence: taozhang@shu.edu.cn; Tel.: +86-21-66135300

Version January 22, 2019 submitted to Future Internet

Abstract: Radio-frequency (RF) tomographic imaging is a promising technique for inferring multi-dimensional physical space by processing RF signals traversed across a region of interest. Tensor based approaches for tomographic imaging are superior in detecting the objects within higher dimensional spaces. The recently proposed tensor sensing approach based on the transform tensor model achieves lower error rate and faster speed than previous tensor-based compress sensing approach. However, the running time of the tensor sensing approach increases exponentially with the dimension of tensors, thus not very practical for big tensors. In this paper, we address this problem by exploiting massively parallel GPUs. We design, implement and optimize the tensor sensing approach on an NVIDIA Tesla GPU and evaluate the performance in terms of the running time and recovery error rate. Experiment results show that our GPU tensor sensing is as accurate as the CPU counterpart with an average of 44.79× and up to 84.70× speedups for varying sized synthetic tensor data. For IKEA Model 3D model data of smaller size, our GPU algorithm achieves 15.37x speedup over the CPU tensor sensing. We further encapsulate the GPU algorithm into an open-source library, called *cuTensorSensing*, which can be used for efficient RF tomographic imaging.

Keywords: Radio frequency; tomographic imaging; tensor; GPU

1. Introduction

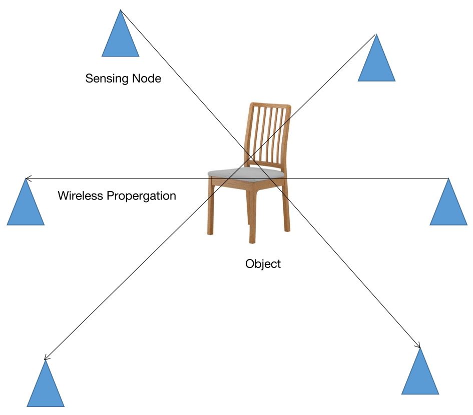

Figure 1. An illustration of RF tomographic imaging network

Radio frequency (RF) tomographic imaging, also called wireless tomography, is a technique for detecting objects within a specific region by analyzing radio frequency signals transmitted between wireless nodes, as shown in Fig. 1. This technique uses radio reflections that bounce off the object body and does not require the object to carry any wireless device. This advantage has made it ideal for security checking, rescue operations, space applications, and smart buildings [3].

In general, wireless signal propagating on a link loses power due to shadowing, distance and multipath fading. The task of RF tomographic imaging is to estimate a spatial distribution of the shadowing loss in a detecting region from measured power of wireless signals. A tensor-based compressed sensing method [3] was proposed for RF tomographic imaging in a three and higher dimensionally structured region, which estimates the distribution by utilizing its low-rank property. However, the method requires computing tensor singular value decomposition (t-SVD) in each algorithmic iteration, which leads to high computational complexity. To address this problem, Deng et al. [?] proposed to use the transform-based tensor model to formulate the RF tomographic imaging as a tensor sensing problem, then use a fast iterative algorithm Alt-Min to solve the tensor sensing problem. Their method fully utilizes the geometric structure of the three-dimensional loss field tensor. Compared to the tensor-based compressed sensing method, Deng's method achieved lower error rate and faster computation speed.

However, the iterative algorithm Alt-Min proposed by Deng et al.[?] iteratively estimates a pair of tensors, which is a computationally-intensive process, thus not very practical for RF tomographic imaging for big objects or regions, or real-time applications. In this paper, we address this problem by exploiting Graphics Processing Units (GPUs). Because of massive hardware parallelism and high memory bandwidth, GPUs have been widely used in diverse applications including machine learning [?] [?] [?], graph processing [?] [?] [?], big data analytic [?] [?], image processing [?], and fluid dynamics [?]. In order to reap the power of GPUs, the algorithmic steps need to be mapped delicately onto the architecture of GPUs, especially the thread and memory hierarchy. In this work, we design, implement and optimize the transform model based tensor sensing method of Deng et al. [?] on an NVIDIA Tesla V100 GPU with CUDA (Compute Unified Device Architecture) and evaluate it in terms of the running time and recovery error. Experiment results show that the GPU algorithm achieves similar relative error as the CPU counterpart. Moreover, the GPU tensor sensing outperforms the CPU tensor sensing on all tensor sizes with 45.39x speedup on average and up to 84.70x speedup on bigger tensor size such as $120 \times 120 \times 6$. We encapsulate this GPU tensor sensing algorithm into an open-source library called "cuTensorSensing" (CUDA Tensor Sensing) which is available at: http://www.findai.net.

Our contributions are summarized as follows. First, we analyze the steps of transform model based tensor sensing using the Alt-Min algorithm and discuss how to map them onto the GPU architecture. Second, we design, implement and optimize the tensor sensing on an NVIDIA Tesla V100 GPU. Third, we evaluate and compare the GPU tensor sensing and the CPU tensor sensing in terms of running time and relative error with both synthetic data and IKEA Model data. Fourth, we encapsulate the GPU tensor sensing implementation into an open-source library such that it can be used in diverse applications.

The remainder of the paper is organized as follows. In Section 2, we discuss the related works. Section 3 presents the notations and briefly summarizes the RF tomographic imaging task as a tensor sensing problem. Section 4 describes the design, implementation and optimizations of the tensor sensing on the GPU. In Section 5, we describe the experiment methology. Section 6 evaluates the GPU tensor sensing with both synthetic data and IKEA Model data. The conclusions are given in Section 7.

2. Related Works

Existing works on RF tomographic imaging can be classified into vector-based and tensor-based approaches. The vector-based approaches [1] [2] aim at estimating a spatial distribution of the shadowing loss in 2D regions of interest. They are not able to infer three-dimensional regions due to the fact that spatial structures of the signal data are ignored. Therefore, researchers have proposed the tensor-based approaches [3] [?] to infer higher dimensional spaces of 3D and above. The tensor-based compressed sensing [3] used tensor nuclear norm (TNN) [4] to extend RF tomographic problems to three-dimensional case. This approach has high computational complexity and error rate. Deng et al. [?] exploited the transform-based tensor model [5] to explore three-dimensional spatial structures for higher accuracy and speed. Our work aims at accelerate Deng's work on GPUs to make it practical for real-time scenarios and bigger tensors.

GPUs have massive parallelism and high memory bandwidth, and many existing researches [6] [7] [8] [9] [10] have demonstrated the benefit of utilizing GPUs to accelerate general purpose computing. Due to the high dimensions of tensors, tensor computations are often computation-intensive and time consuming. Recently, GPUs have been increasingly adopted to accelerate diverse tensor computations. Some works focus on accelerating specific tensor operations including tensor contraction [?] [14], factorization [?], transpose [?] [?], and tensor-matrix multiplication [?]. These works propose parallel tensor algorithms specifically optimized for the GPU architectures. GPUTENSOR [?] is a parallel tensor factorization algorithm that splits a tensor into smaller blocks and exploits the inherent parallelism and high-memory bandwidth of GPUs. To handle dynamic tensor data, GPUTENSOR updates its previously factorized components instead of recomputing them from the raw data. Sparse tensor-times-dense matrix multiply is a critical bottleneck in data analysis and mining applications and [15] proposed an efficient primitive on CPU and GPU platforms. Different with these works, our work considers the tensor sensing based on the transform tensor model for RF tomographic imaging.

3. Description of the Tensor Sensing Problem

Deng et al. [?] proposed to use the transform-based tensor model to formulate the RF tomographic imaging as a tensor sensing problem. Then they utilized a fast iterative algorithm Alt-Min to solve the tensor sensing problem. Here we briefly summarize their approach including the Alt-Min algorithm in order to map it onto the GPU architecture.

3.1. Alt-Min Algorithm

The goal of tensor sensing is to recover the loss field tensor $\mathcal{X}$ from the linear map matrix $\mathbf{A}$ and the measurement vector $\mathbf{y}$, which is formulated as following:

$$\widehat{\mathbf{X}} = \arg\min_{\mathbf{X}} \|\mathbf{y} - \mathbf{A}\mathbf{X}\|_F^2, \text{s.t.rank}(\mathbf{X}) \leq r. \tag{1}$$

This method uses Alt-Min algorithm(Alg.1) to iteratively estimate two low rank matrices whose matrix product is the squeezed matrix of the object tensor $\mathcal{X}$.

Algorithm 1 Alt-Min Algorithm of the Tensor Sensing

```
Input: linear map matrix $\mathbf{A}$, measurement vector $\mathbf{y}$, iteration number $L$.
Output: squeezed $\mathcal{X}$: $\mathbf{X}$
1: Initialize $\mathcal{U}^0$ randomly;
2: for $\ell = 1$ to $L$ do
3:     $\mathbf{V}^\ell \leftarrow$ least squares minimization$(\mathbf{A}, \mathbf{U}^{\ell-1}, \mathbf{y})$
4:     $\mathbf{U}^\ell \leftarrow$ least squares minimization$(\mathbf{A}, \mathbf{V}^{\ell}, \mathbf{y})$
5: end for
Output: : Pair of tensors $(\mathbf{U}^L, \mathbf{V}^L)$.
```

3.2. Implementation of the Tensor Sensing on CPU

Alg.2 shows the implementation of tensor sensing on CPU. First, $\mathbf{U}$ of previous iteration(for the

Algorithm 2 Implementation of the Tensor Sensing on CPU

```
Input: Randomly initialized matrix $\mathbf{U}$, measurement vector $\mathbf{y}$, linear map matrix $\mathbf{A}$
Output: $\mathbf{X}$
1: for $i = 1 \to IterNum$ do
2:     use $\mathbf{U}$ to form a block diagonal matrix $\mathbf{U}_b$
3:     $\mathbf{W} \leftarrow \mathbf{A} * \mathbf{U}_b$
4:     vec($\mathbf{V}$) $\leftarrow$ perform least square minimization on $\mathbf{W}$ and $\mathbf{y}$
5:     $\mathbf{V} \leftarrow$ transform vec($\mathbf{V}$) back to matrix form
6:     $\mathbf{V}_t \leftarrow$ transpose$\mathbf{V}$
7:     use $\mathbf{V}$ to form a block diagonal matrix $\mathbf{V}_{t_b}$
8:     $\mathbf{A}_t \leftarrow$ transpose$\mathbf{A}$
9:     $\mathbf{W} = \mathbf{A}_t * \mathbf{V}_{t_b}$
10:    vec($\mathbf{U}_t$) $\leftarrow$ perform least square minimization on $\mathbf{W}$ and $\mathbf{y}$
11:    $\mathbf{U}_t \leftarrow$ transform vec($\mathbf{U}_t$) back to matrix form
12:    $\mathbf{U} \leftarrow$ transpose$\mathbf{U}_t$
13: end for
14: return $\mathbf{X} = \mathbf{UV}$
```

first iteration, $\mathbf{U}$ is initialized randomly) is used to form a block diagonal matrix $\mathbf{U}_b$(Alg.2 line 2):

$$\mathbf{U}_b = \begin{bmatrix} \mathbf{U} & & & \\ & \mathbf{U} & & \\ & & \ddots & \\ & & & \mathbf{U} \end{bmatrix} \tag{2}$$

Next, the least square minimization problem is solved to estimate $\mathbf{V}$ of the next iteration(Alg.2 line 3 and line 4). The least minimization problem is formulated as following:

$$\text{vec}(\mathbf{V}) = \arg\min_{\mathbf{V}} \|\mathbf{y} - \mathbf{A}\mathbf{U}_b\mathbf{V}\|_F^2 \tag{3}$$

As the estimated $\mathbf{V}$ is in vector form vec($\mathbf{V}$), vec($\mathbf{V}$) is transformed back to matrix form $\mathbf{V}$(Alg.2 line 5). To estimate $\mathbf{U}$, the least square minimization problem in equation.3 should be transposed:

$$\text{vec}(\mathbf{U}_t) = \arg\min_{\mathbf{U}_t} \|\mathbf{y} - \mathbf{A}_t\mathbf{V}_t\mathbf{U}_t\|_F^2 \tag{4}$$

The process of estimating $\mathbf{U}$ similar to estimating $\mathbf{V}$ is implemented with the corresponding transposed matrix(Alg.2 line 6 to line 12). Next, the above process is repeated until the number of iteration reaches the set value. Last, the two estimated matrices are multiplied to get the final result matrix(Alg.2 line 14).

4. The Implementation and Optimization of Efficient GPU Tensor Sensing

To achieve high performance on GPUs, we need to consider the data representation, the mappings from computations to GPU threads, and memory accesses. We first design a basic GPU tensor sensing implementation, then optimize the implementation to further improve performance.

4.1. Design and Implementation of the GPU Tensor Sensing

4.1.1. Data Struct

In Alg. 2, after least squares minimization (lines 4 and line 10 in Alg. 2), we get vectorized matrices. For a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, the corresponding vectorized $\mathbf{A}$ is $\mathbf{A}_v \in \mathbb{R}^{mn\times 1}$. And the vectorized matrices are converted back to the original matrices (lines 5 and 11 in Alg. 2). In some scientific computing programming languages, such as Matlab, this conversion must be done with the appropriate conversion function. We adopt the column-first storage format to store matrices and vectors, which not only ensures read and write continuity but also avoids explicit vector-to-matrix conversions since vector $\mathbf{A}_v$ and matrix $\mathbf{A}$ in memory are the same in this format, as shown in Fig. 2.

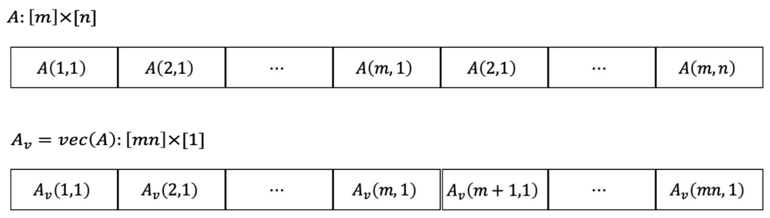

Figure 2. Vectorize a matrix **A** into vec(**A**) in memory

4.1.2. Multiply of Block Diagonal Matrices

Using the operational properties of the block matrix we get:

$$[\mathbf{A}_1, \mathbf{A}_2, \cdots, \mathbf{A}_{N_2}] \begin{bmatrix} \mathbf{U}^c & & & \\ & \mathbf{U}^c & & \\ & & \ddots & \\ & & & \mathbf{U}^c \end{bmatrix} = [\mathbf{A}_1\mathbf{U}^c, \mathbf{A}_2\mathbf{U}^c, \cdots, \mathbf{A}_{N_2}\mathbf{U}^c] \tag{5}$$

It shows that multiplication of block diagonal matrices can be transformed into a batch of small matrix multiplications. As we use column-first format to store **A**, the batch of $\mathbf{A}_i$ are stored in constant stride. Let p indicate the location of the first element of $\mathbf{A}_0$, then the location of the first element of $\mathbf{A}_i$ is $p + i \times N_1 N_3$. We utilize the gemmStridedBatchd() routine in NVIDIA cuBLAS Library to compute a batch of matrix multiplications simultaneously to achieve better performance.

Table 1. The parameters of the gemmStridedBatchd() routine in the cuBLAS library.

Parameters	Meaning	Value
transA	operation op(**A**) that is non- or transpose	non-transpose
transU	operation op(**U**) that is non- or transpose	transpose
A	pointer to the **A** matrix corresponding to the first instance of the batch	d_A
U	pointer to the **U** matrix	d_y
W	pointer to the **W** matrix	d_W
strideA	the address offset between $\mathbf{A}_i$ and $\mathbf{A}_{i+1}$	$M \times N_1 N_3$
strideU	the address offset between $\mathbf{U}_i$ and $\mathbf{U}_{i+1}$	0
strideW	the address offset between $\mathbf{W}_i$ and $\mathbf{W}_{i+1}$	$M \times N_3$
batchNum	number of *gemm* to perform in the batch	N_2

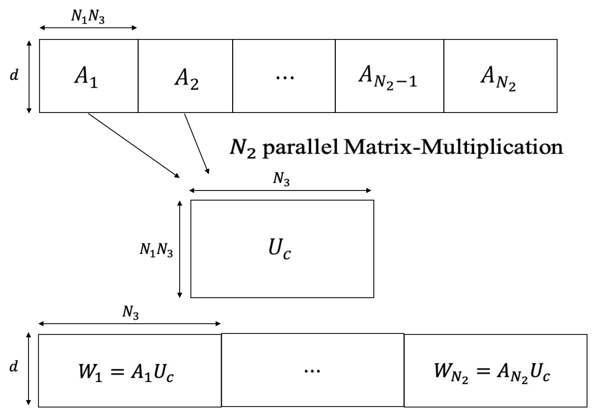

Figure 3. Multiplication of block diagonal matrices on GPU

4.1.3. Eliminating Explicit Transpose Operations

After each least squares method, the transpose of the target matrix is obtained. The transpose operation of the matrix needs to be performed (lines 6 and 12 in Alg. 2). However, the transpose operation takes a lot of computing time and resource. As the operation after transpose of the matrix is multiplication of diagonal matrices, we eliminate explicit matrix transpose by enabling the transpose option in the gemmStridedBatched() routine. In this way, the gemmStridedBatched() routine will perform matrix transpose implicitly and efficiently before the matrix multiplications.

4.1.4. Least Squares Minimization

As shown in Alg. 2, least squares minimization (LSM) is the major step of the tensor sensing approach, which is the most time-consuming part of the entire approach. There are many approaches to perform least squares minimization. QR factorization is one of the most efficient approaches, which is well supported by CUDA.

4.2. Optimizations of the GPU Tensor Sensing

During the computation flow of the GPU tensor sensing, frequent data transfer between the CPU and GPU will significantly degrade system performance. Therefore we design data reusing strategy to reduce data transfer overhead and resource consumption.

In the entire tensor sensing flow, we envoke data transfer only twice at the beginning and in the end. At the beginning, the input data is transferred from the CPU to the GPU. In the end, the final result matrix is sent back to the CPU. We optimize the computations in the tensor sensing flow such that they all perform in-place calculations. For instance, in the QR decomposition to solve the least squares problem, the input vector **y** is overrode by the result vectors (vec(**U**) and vec(**V**)). Therefore, we need to reassign the vector **y** at the beginning of each least squares minimization iteration. However, it is an expensive operation to load the original data of vector **y** from the CPU memory every time. Instead, we pre-allocate a memory space named dyL on the GPU that stores the original data of the vector **y**. Every time we need to reassign the dy, we use the GPU device to device transfer routine cudaMemcpyDeviceToDevice() to copy the data from dyL to dy. Since the device to device bandwidth $380GB/s$ is much higher than the bandwidth between the CPU and GPU $12GB/s$, this strategy significantly reduces data transfer overhead. Alg.3 and Fig.4 describes the computation flow and data organization of the tensor sensing on the GPU.

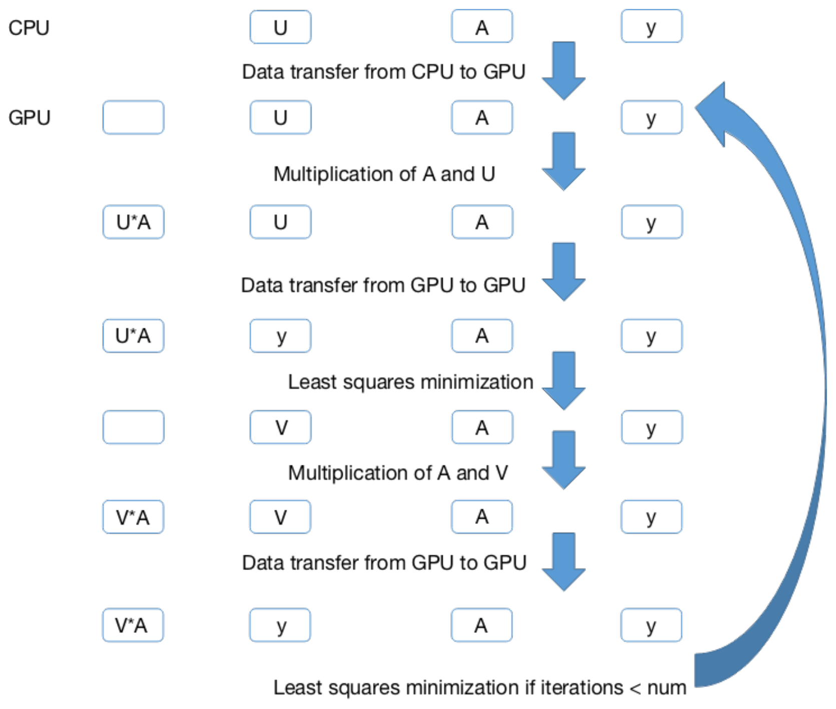
CPU
U
A
y
Data transfer from CPU to GPU
GPU
U
A
y
Multiplication of A and U
U*A
U
A
y
Data transfer from GPU to GPU
U*A
y
A
y
Least squares minimization
V
A
y
Multiplication of A and V
V*A
V
A
y
Data transfer from GPU to GPU
V*A
y
A
y
Least squares minimization if iterations < num

Figure 4. Memory organization in the calculation process

5. Experiment Methodology

In this section, we describe the experiment methodology including the hardware and software platform, testing data, testing process, and the comparison metrics.

5.1. Hardware and Software Platform

We use an NVIDIA Tesla V100 GPU to evaluate the performance of GPU tensor sensing. The GPU incorporates 5120 CUDA cores @ 1.53 GHz and 32 GB DDR5 memory. It is installed on a server with 128 GB DDR memory and an Intel i7-7820x CPU with 8 cores @ 3.60 GHz supporting 16 hardware threads with hyperthreading technology. The server is running Ubuntu 18.04 LTS with the kernel version 4.15.0. The CPU and GPU tensor sensing is running with Matlab R2017b and CUDA 10.0, repectively.

5.2. Testing Data

In the experiment, we use both IKEA model and synthetic data in the evaluation. For IKEA model data, we use the IKEA 3D data (**??**) that generates a ground truth tensor of size $60 \times 60 \times 15$ with rank 6. Each 3D model is placed in the middle of the "tensor" and occupies a part of the space. In this task, we mainly focus on the location and outline information, while the texture and color information are ignored. The synthetic data is generated according to the compressed sensing model [3]. The synthetic

Algorithm 3 Computation Flow of the Tensor Sensing on the GPU

```
Input: Data on CPU memory: randomly initialized $\mathbf{U}^0$, measurement vector $\mathbf{y} \in \mathbf{R}^M$, matrix $\mathbf{A} \in \mathbb{R}^{M \times N_1 N_2 N_3}$ converted from $M$ sensing tensors $\mathcal{A}_m$
Output: $\mathbf{X} \in \mathbb{R}^{N_1 N_3 \times N_2}$
1: apply for memory on GPU device: $dy, dA, dW, dyL$ ($dA(\mathcal{A})$ means that the content in $dA$ is $\mathcal{A}$, the same below)
2: data transfer: $\mathcal{A}, \mathbf{y}, \mathbf{U}^0 \xrightarrow{cudaMemcpyHostToDevice} dA(\mathcal{A}), dy(\mathbf{U}^0), dyL(\mathbf{y})$
3: for $i = 0 \to IterNum$ do
4:     $dW(\text{uninitialized}), dA(\mathcal{A}), dy(\mathbf{U}^i) \xrightarrow{cublas<t>gemmStridedBatchd} dW(\mathcal{A}\,\text{diag}(\mathbf{U}^i)), dA(\mathcal{A}), dy(\mathbf{U}^i)$
5:     $dy(\mathbf{U}^i), dyL(\mathbf{y}) \xrightarrow{cudaMemcpyDeviceToDevice} dy(\mathbf{y}), dyL(\mathbf{y})$
6:     $dW(\mathcal{A}\,\text{diag}(\mathbf{U}^i)), dy(\mathbf{y}) \xrightarrow{cusolver<t>qr} dy(\mathbf{V}), dW(\text{uninitialized})$
7:     $dW(\text{uninitialized}, dA(\mathcal{A}), dy(\mathbf{V}^i))e \xrightarrow{cublas<t>gemmStridedBatchd} dW(\mathcal{A}\,\text{diag}(\mathbf{V}^i)), dA(\mathcal{A}), dy(\mathbf{V}^i)$
8:     $dy(\mathbf{V}^i), dyL(\mathbf{y}) \xrightarrow{cudaMemcpyDeviceToDevice} dy(\mathbf{y}), dyL(\mathbf{y})$
9:     $dW(\mathcal{A}\,\text{diag}(\mathbf{V}^i)), dy(\mathbf{y}) \xrightarrow{cusolver<t>qr} dy(\mathbf{U}^i), dW(\text{uninitialized})$
10: end for
11: return $\mathbf{X} = \mathbf{U}\mathbf{V}$
```

data does not correspond to a specific physical model. We use it to demonstrate the performance of our approach at different data sizes.

5.3. Testing Process

The synthetic and real data are processed by the CPU tensor sensing implementation in Matlab and GPU tensor sensing implementation in CUDA, respectively. We evaluate and compare two versions of GPU tensor sensing: the unoptimized one and the optimized one. The unoptimized GPU tensor sensing adopts none of the optimization techniques in Sec. 4.2. We repeat each experiment five times and report the average results.

5.4. Comparison Metrics

We adopt two metrics for comparison: running time and relative error rate.

- *Running time:* varying the tensor size and fixing other parameter, we measure the execution time of the CPU tensor sensing, unoptimized GPU tensor sensing and optimized GPU tensor sensing. Finally we calculate speedups as the running time of the CPU tensor sensing divided by the running time of GPU tensor sensing.
- *error rate:* we adopt the metric relative square error, defined as $RSE = \|\widehat{\mathcal{X}} - \mathcal{X}\|_F / \|\mathcal{X}\|_F$.

6. Results and Analysis

6.1. Running Time of Synthetic Data

()

Figure 7: (a) is the running time of the CPU tensor sensing and GPU tensor sensing, (b) is the speedups of the unoptimized and optimized GPU tensor sensing.

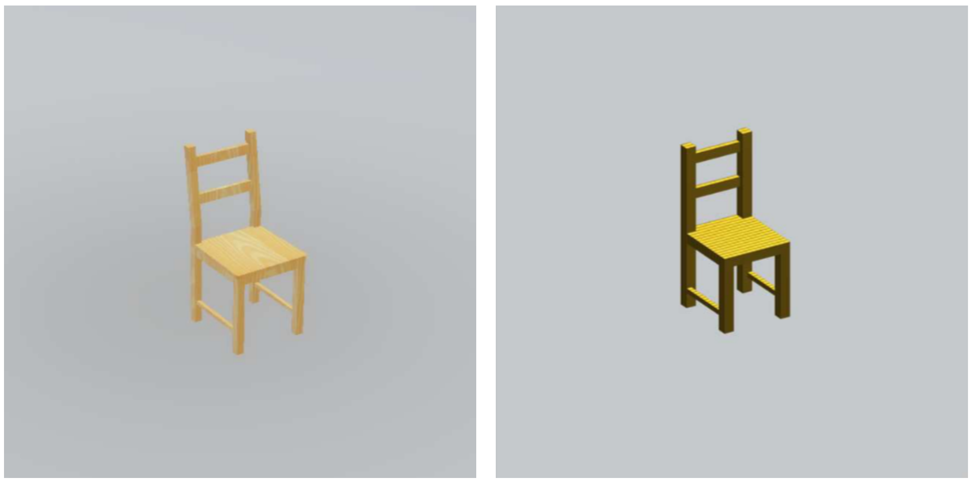

Figure 5. (a) is the 3D visualization of an IKEA models, (b) is the corresponding recovery results.

Table 2. Running time under varying tensor size ($n \times n \times 6$).

n	40	60	80	100	120
CPU tensor sensing time (S)	3.07	13.65	50.43	118.84	251.54
Unoptimized GPU tensor sensing time (S)	9.50	11.40	12.15	20.70	25.74
Optimized GPU tensor sensing time (S)	0.44	0.63	0.98	2.01	2.97
Speedups-unoptimized	0.32	1.20	4.15	5.74	9.77
Speedups-optimized	6.98	21.67	51.46	59.12	84.70

Fig. 7 shows that the running time of the CPU tensor sensing and two GPU tensor sensing implementations (unoptimized and optimized ones) for $\mathcal{X}$ of size $n \times n \times 6$ of rank 1, where n varies from 40 to 120 at a step of 20. The sampling rate is set to 20%, and both CPU and GPU tensor sensing iterate 5 iterations for completion. The detailed time value is listed in Table 2. While $M = N_1 \times N_2 \times N_3 \times$ sampling rate and $\mathcal{A} \in \mathbb{R}^{M \times N_1 N_2 N_3}$, the scale of the main operation matrix $\mathcal{A}$ increases at a rate of four times as the increase rate of n.

We can see that the running time of the CPU tensor sensing is polynomial with the increase of n, while the running time of the GPU tensor sensing is linearly growing. The unoptimzed and optimized GPU tensor sensing achieved an average of 4.24× and 44.79× and up to 9.77× and 84.70× speedups, respectively. This illustrates the effectiveness of the optimization methods proposed in Sec. 4.2. When the tensor size is small, the data transfer occupies a major portion of the entire execution time of the unoptimized GPU tensor sensing. As a result, its performance is even lower than the CPU tensor sensing.

6.2. Error Rate and Running Time of IKEA Model Data

This experiment evaluate the error rate of the CPU and GPU tensor sensing under different iterations for $\mathcal{X}$ of size $60 \times 60 \times 15$ with rank 3. The sampling rate is set to 50%. Fig.**??** shows the recovery results at 10 iterations. The running time of the CPU tensor sensing at 5 iterations is 14.91 seconds on average, while the running time of the GPU tensor sensing is 0.97 second on average, thus the speedup is 15.37×. As shown in Fig. 8, under increased iterations from 1 to 30, the RSEs drop significantly. More importantly, the CPU tensor sensing and GPU implementation achieve almost the same RSEs at all iterations (the two curves in Fig. 8 overlap with each other), which means that they achieve similar error rate performance in tensor sensing.

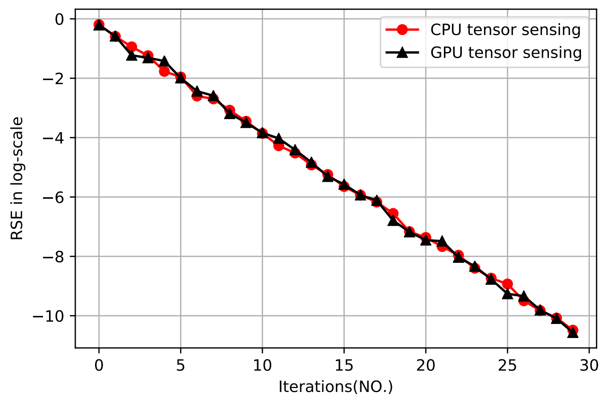

Figure 8. RSE of the CPU tensor sensing and GPU tensor sensing.

7. Conclusions

In this work, we present an open-source library named *cuTensorSensing* for efficient RF tomographic imaging on GPUs. The experiment evaluations show that the proposed GPU tensor sensing works effectively and accurately. Compared with the counterpart on CPU, the GPU tensor sensing achieves similar error rate but much faster speed. For synthetic data, the GPU tensor sensing achieves an average of 44.79× and up to 84.70× speedups versus the CPU tensor sensing for bigger tensors. For IKEA Model 3D objects data of smaller tensor size, the GPU tensor sensing achieves a 15.37× speedup over the CPU tensor sensing. The cuTensorSensing library is useful for efficient RF tomographic imaging.

Author Contributions: abstract, introduction, related works, conclusions and the revising of the entire manuscript, Tao Zhang; algorithms, experiments, and results, Da Xu.

Funding: This research is partially supported by Natural Science Foundation of Shanghai under grant No. 17ZR1409800, Science and technology committee of Shanghai Municipality under grant No. 16010500400.

Acknowledgments: The authors would like to thank anonymous reviewers for their fruitful feedback and comments that have helped them improve the quality of this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kanso, M.A.; Rabbat, M.G. Compressed RF tomography for wireless sensor networks: Centralized and decentralized approaches. International Conference on Distributed Computing in Sensor Systems. Springer, 2009, pp. 173–186.
2. Mostofi, Y. Compressive cooperative sensing and mapping in mobile networks. *IEEE Transactions on Mobile Computing* **2011**, *10*, 1769–1784.
3. Matsuda, T.; Yokota, K.; Takemoto, K.; Hara, S.; Ono, F.; Takizawa, K.; Miura, R. Multi-dimensional wireless tomography using tensor-based compressed sensing. *Wireless Personal Communications* **2017**, *96*, 3361–3384.
4. Li, Q.; Schonfeld, D.; Friedland, S. Generalized tensor compressive sensing. Multimedia and Expo (ICME), 2013 IEEE International Conference on. IEEE, 2013, pp. 1–6.
5. Liu, X.Y.; Wang, X. Fourth-order tensors with multidimensional discrete transforms. *arXiv preprint arXiv:1705.01576* **2017**.
6. Jing, N.; Jiang, L.; Zhang, T.; Li, C.; Fan, F.; Liang, X. Energy-efficient eDRAM-based on-chip storage architecture for GPGPUs. *IEEE Transactions on Computers* **2016**, *65*, 122–135.
7. Zhang, T.; Jing, N.; Jiang, K.; Shu, W.; Wu, M.Y.; Liang, X. Buddy SM: Sharing Pipeline Front-End for Improved Energy Efficiency in GPGPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* **2015**, *12*, 1–23.

8. Zhang, T.; Zhang, J.; Shu, W.; Wu, M.Y.; Liang, X. Efficient graph computation on hybrid CPU and GPU systems. *The Journal of Supercomputing* **2015**, *71*, 1563–1586.
9. Zhang, T.; Shu, W.; Wu, M.Y. CUIRRE: An open-source library for load balancing and characterizing irregular applications on GPUs. *Journal of parallel and distributed computing* **2014**, *74*, 2951–2966.
10. Zhang, T.; Tong, W.; Shen, W.; Peng, J.; Niu, Z. Efficient Graph Mining on Heterogeneous Platforms in the Cloud. In *Cloud Computing, Security, Privacy in New Computing Environments*; Springer, 2016; pp. 12–21.
11. Chetlur, S.; Woolley, C.; Vandermersch, P.; Cohen, J.; Tran, J.; Catanzaro, B.; Shelhamer, E. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* **2014**, pp. 1–9.
12. Tsung, P.K.; Tsai, S.F.; Pai, A.; Lai, S.J.; Lu, C. High performance deep neural network on low cost mobile GPU. Consumer Electronics (ICCE), 2016 IEEE International Conference on. IEEE, 2016, pp. 69–70.
13. Banasiak, D. Statistical Methods of Natural Language Processing on GPU. In *Man–Machine Interactions 4*; Springer, 2016; pp. 595–604.
14. Shi, Y.; Niranjan, U.; Anandkumar, A.; Cecka, C. Tensor contractions with extended BLAS kernels on CPU and GPU. High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on. IEEE, 2016, pp. 193–202.
15. Li, J.; Ma, Y.; Yan, C.; Vuduc, R. Optimizing sparse tensor times matrix on multi-core and many-core architectures. Irregular Applications: Architecture and Algorithms (IA3), Workshop on. IEEE, 2016, pp. 26–33.
16. Dongarra, J.; Hammarling, S.; Higham, N.J.; Relton, S.D.; Zounon, M. Optimized Batched Linear Algebra for Modern Architectures. European Conference on Parallel Processing. Springer, 2017, pp. 511–522.