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 200444, China; mr_tab@shu.edu.cn

² Shanghai Institute for Advanced Communication and Data Science, Shanghai 200444, China

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

Received: 4 January 2019; Accepted: 11 February 2019; Published: 15 February 2019

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 at detecting the objects within higher dimensional spaces. The recently-proposed tensor sensing approach based on the transform tensor model achieves a lower error rate and faster speed than the previous tensor-based compress sensing approach. However, the running time of the tensor sensing approach increases exponentially with the dimension of tensors, thus not being 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. Experimental results show that our GPU tensor sensing is as accurate as the CPU counterpart with an average of $44.79\times$ and up to $84.70\times$ speedups for varying-sized synthetic tensor data. For IKEA Model 3D model data of a smaller size, our GPU algorithm achieved $15.374\times$ speedup over the CPU tensor sensing. We further encapsulate the GPU algorithm into an open-source library, called cuTensorSensing (CUDA Tensor Sensing), which can be used for efficient RF tomographic imaging.

Keywords: radio frequency; tomographic imaging; tensor; GPU

1. Introduction

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 Figure 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 [1].

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 the measured power of wireless signals. Classic radio tomography and vector-based methods have been proposed. The work in [2] presented a linear model for using received signal strength (RSS) measurements to obtain images of moving objects. The work in [3] addressed substantial issues for the practical implementation. A tensor-based compressed sensing method [1] 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. [4] proposed to use the transform-based tensor model to formulate the RF tomographic

imaging as a tensor sensing problem, then used 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 a lower error rate and faster computation speed.

Figure 1. An illustration of the RF tomographic imaging network.

However, the iterative algorithm Alt-Min proposed by Deng et al. [4] iteratively estimates a pair of tensors, which is a computationally-intensive process, thus not being 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 [5–7], graph processing [8–10], big data analytics [11,12], image processing [13], and fluid dynamics [14]. 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. [4] 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 $45.39\times$ speedup on average and up to $84.70\times$ speedup on bigger tensor sizes 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 [15].

Our contributions are summarized as follows. First, we analyze the steps of the 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 methodology. 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 [16,17] 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 [1,4] to infer three-dimensional spaces. The tensor-based compressed sensing [1] uses the tensor nuclear norm (TNN) [18] to extend RF tomographic problems to the three-dimensional case. This approach has a high computational complexity and error rate. Deng et al. [4] exploited the transform-based tensor model [19] 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 research works [20–24] 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 focused on accelerating specific tensor operations including tensor contraction [25,26], factorization [27], transpose [28,29], and tensor-matrix multiplication [30]. These works propose parallel tensor algorithms specifically optimized for the GPU architectures. GPOTENSOR [31] 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, GPOTENSOR updates its previously-factorized components instead of recomputing them from the raw data. Sparse tensor-times-dense matrix multiplication is a critical bottleneck in data analysis and mining applications, and [32] proposed an efficient primitive on CPU and GPU platforms. Different from 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. [4] 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 follows:

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

This method uses the Alt-Min algorithm (Algorithm 1) to estimate two low rank matrices iteratively 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** $\ell = 1-L$ **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

Algorithm 2 shows the implementation of tensor sensing on CPU.

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

First, $\mathbf{U}$ of the previous iteration (in the first iteration, $\mathbf{U}$ is initialized randomly) is used to form a block diagonal matrix $\mathbf{U}_b$ (Algorithm 2, Line 2):

$$\mathbf{U}_b = \begin{bmatrix} \mathbf{U} & & & \\ & \mathbf{U} & & \\ & & \ddots & \\ & & & \mathbf{U} \end{bmatrix} \quad (2)$$

Next, the least squares minimization problem is solved to estimate $\mathbf{V}$ of the next iteration (Algorithm 2, Line 3 and Line 4). The least minimization problem is formulated as follows:

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

As the estimated $\mathbf{V}$ is in vector form $\text{vec}(\mathbf{V})$, $\text{vec}(\mathbf{V})$ is transformed back to matrix form $\mathbf{V}$ (Algorithm 2, Line 5). To estimate $\mathbf{U}$, the least squares minimization problem in Equation (3) should be transposed:

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

The process of estimating $\mathbf{U}$ is similar to estimating $\mathbf{V}$, which is implemented with the corresponding transposed matrix (Algorithm 2 Lines 6–12). Next, the above process is repeated until the number of iterations reaches the set value. Lastly, the two estimated matrices are multiplied to get the final result matrix (Algorithm 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 Structure

In Algorithm 2, after least squares minimization (Lines 4 and 10 in Algorithm 2), we get vectorized matrices. For a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the corresponding vectorized $\mathbf{A}$ is $\mathbf{A}_v \in \mathbb{R}^{mn \times 1}$. The vectorized matrices are converted back to the original matrices (Lines 5 and 11 in Algorithm 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 Figure 2.

Figure 2. Vectorizing a matrix $\mathbf{A}$ into $\text{vec}(\mathbf{A})$ in memory.

4.1.2. Multiplication of Block Diagonal Matrices

Using the operational properties of the block matrix, we get:

$$[\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_{N_2}] \begin{bmatrix} \mathbf{U}^c & & & \\ & \mathbf{U}^c & & \\ & & \ddots & \\ & & & \mathbf{U}^c \end{bmatrix} = [\mathbf{A}_1\mathbf{U}^c, \mathbf{A}_2\mathbf{U}^c, \dots, \mathbf{A}_{N_2}\mathbf{U}^c] \quad (5)$$

This 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 $\mathbf{A}$, the batch of $\mathbf{A}_i$ is 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 the NVIDIA cuBLAS Library to compute a batch of matrix multiplications simultaneously to achieve better performance. Figure 3 shows how this process performs on GPU and Table 1 shows the parameters setting of this routine.

Figure 3. Multiplication of block diagonal matrices on GPU.

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

Parameters	Meaning	Value
transA	operation $\text{op}(\mathbf{A})$ that is non- or transpose	non-transpose
transU	operation $\text{op}(\mathbf{U})$ that is non- or transpose	transpose
$\mathbf{A}$	pointer to the $\mathbf{A}$ matrix corresponding to the first instance of the batch	d_A
$\mathbf{U}$	pointer to the $\mathbf{U}$ matrix	d_y
$\mathbf{W}$	pointer to the $\mathbf{W}$ 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 <i>gemm</i> to perform in the batch	N_2

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 Algorithm 2). However, the transpose operation takes much computing time and resource. As the operation after transpose of the matrix is the multiplication of diagonal matrices, we eliminate the 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 Algorithm 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 a data reusing strategy to reduce data transfer overhead and resource consumption.

In the entire tensor sensing flow, we invoke data transfer only twice at the beginning and at the end. At the beginning, the input data are transferred from the CPU to the GPU. At 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 $\mathbf{y}$ is overrode by the result vectors ($\text{vec}(\mathbf{U})$ and $\text{vec}(\mathbf{V})$). Therefore, we need to reassign the vector $\mathbf{y}$ at the beginning of each least squares minimization iteration. However, it is an expensive operation to load the original data of vector $\mathbf{y}$ 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 $\mathbf{y}$. 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 of 380 GB/s is much higher than the bandwidth between the CPU and GPU of 12 GB/s, this strategy significantly reduces data transfer overhead. Algorithm 3 and Figure 4 describe the computation flow and data organization of the tensor sensing on the GPU.

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 $\mathbf{A} \in$

$\mathbf{R}^{M \times N_1 N_2 N_3}$ converted from M sensing tensors $\mathcal{A}_m$

Output: $\mathbf{X} \in \mathbf{R}^{N_1 N_3 \times N_2}$

- 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: $\mathcal{A}, \mathbf{y}, \mathbf{U}^0 \xrightarrow{\text{cudaMemcpyHostToDevice}} dA(\mathcal{A}), dy(\mathbf{U}^0), dyL(\mathbf{y})$
 - 3: **for** $i = 0 \rightarrow \text{IterNum}$ **do**
 - 4: $dW(\text{uninitialized}), dA(\mathcal{A}), dy(\mathbf{U}^i) \xrightarrow{\text{cublas}<t>\text{gemmStridedBatchd}} dW(\mathcal{A} \text{diag}(\mathbf{U}^i)), dA(\mathcal{A}), dy(\mathbf{U}^i)$
 - 5: $dy(\mathbf{U}^i), dyL(\mathbf{y}) \xrightarrow{\text{cudaMemcpyDeviceToDevice}} dy(\mathbf{y}), dyL(\mathbf{y})$
 - 6: $dW(\mathcal{A} \text{diag}(\mathbf{U}^i)), dy(\mathbf{y}) \xrightarrow{\text{cusolver}<t>\text{qr}} dy(\mathbf{V}), dW(\text{uninitialized})$
 - 7: $dW(\text{uninitialized}, dA(\mathcal{A}), dy(\mathbf{V}^i)) \xrightarrow{\text{cublas}<t>\text{gemmStridedBatchd}} dW(\mathcal{A} \text{diag}(\mathbf{V}^i)), dA(\mathcal{A}), dy(\mathbf{V}^i)$
 - 8: $dy(\mathbf{V}^i), dyL(\mathbf{y}) \xrightarrow{\text{cudaMemcpyDeviceToDevice}} dy(\mathbf{y}), dyL(\mathbf{y})$
 - 9: $dW(\mathcal{A} \text{diag}(\mathbf{V}^i)), dy(\mathbf{y}) \xrightarrow{\text{cusolver}<t>\text{qr}} dy(\mathbf{U}^i), dW(\text{uninitialized})$
 - 10: **end for**
 - 11: **return** $\mathbf{X} = \mathbf{UV}$
-

Figure 4. Memory organization in the calculation process.

5. Experiment Methodology

In this section, we describe the experimental 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 hyper-threading technology. The server is running Ubuntu 18.04 LTS with Kernel Version 4.15.0. The CPU and GPU tensor sensing is running with MATLAB R2017b and CUDA 10.0, respectively.

5.2. Testing Data

In the experiment, we used both IKEA model and synthetic data in the evaluation. For IKEA model data, we used the IKEA 3D data [1] that generated a ground truth tensors of size $60 \times 60 \times 15$ with rank 6. Each 3D model was placed in the middle of the “tensor” and occupied a part of the space. In this task, we mainly focused on the location and outline information, while the texture and color information were ignored. The synthetic data were generated according to the compressed sensing model [1]. The synthetic data did not correspond to a specific physical model. We used them to demonstrate the performance of our approach for different data sizes.

5.3. Testing Process

The synthetic and real data were processed by the CPU tensor sensing implementation in MATLAB and GPU tensor sensing implementation in CUDA, respectively. We evaluated and compared two versions of GPU tensor sensing: the unoptimized one and the optimized one. The unoptimized GPU tensor sensing adopted none of the optimization techniques in Section 4.2. We repeated each experiment five times and report the average results.

5.4. Comparison Metrics

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

- Running time: Varying the tensor size and fixing other parameters, we measured the execution time of the CPU tensor sensing, unoptimized GPU tensor sensing, and optimized GPU tensor sensing. Finally, we calculated speedups as the running time of the CPU tensor sensing divided by the running time of GPU tensor sensing.
- Error rate: We adopted the metric relative square error, defined as $RSE = \|\hat{\mathcal{X}} - \mathcal{X}\|_F / \|\mathcal{X}\|_F$.

6. Results and Analysis

6.1. Running Time of Synthetic Data

Figure 5a 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 one, where n varies from 40–120 at a step of 20. The sampling rate was set to 20%, and both CPU and GPU tensor sensing performed five iterations for completion. The detailed time value is listed in Table 2. While $M = N_1 \times N_2 \times N_3 \times \text{sampling rate}$ and $\mathcal{A} \in \mathbb{R}^{M \times N_1 N_2 N_3}$, the scale of the main operation matrix $\mathcal{A}$ increased at a rate of four times as the increase rate of n .

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

Figure 5. (a) The running time of the CPU tensor sensing and GPU tensor sensing and (b) the speedups of the unoptimized and optimized GPU tensor sensing.

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

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

6.2. Error Rate and Running Time of IKEA Model Data

This experiment evaluated 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 six. The sampling rate was set to 50%. Figure 6b shows

the recovery results with 10 iterations. The running time of the CPU tensor sensing at five iterations was 14.91 s on average, while the running time of the GPU tensor sensing was 0.97 s on average; thus, the speedup is $15.37\times$. As shown in Figure 7, under increased iterations from 1–30, the RSEs dropped significantly. More importantly, the CPU tensor sensing and GPU implementation achieved almost the same RSEs at all iterations (the two curves in Figure 7 overlap with each other), which means that they achieved similar error rate performance in tensor sensing.

Figure 6. (a) The 3D visualization of the IKEA models and (b) the corresponding recovery results.

Figure 7. 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 achieved a similar error rate, but much faster speed. For synthetic data, the GPU tensor sensing achieved an average of $44.79\times$ and up to $84.70\times$ speedups versus the CPU tensor sensing for bigger tensors. For IKEA Model 3D objects' data of a smaller tensor size, the GPU tensor sensing achieved a $15.37\times$ 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, T.Z.; algorithms, experiments, and results, D.X.

Funding: This research is partially supported by the Natural Science Foundation of Shanghai under Grant No. 17ZR1409800 and the Science and Technology Committee of Shanghai Municipality under Grant No. 16010500400.

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

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

References

1. Matsuda, T.; Yokota, K.; Takemoto, K.; Hara, S.; Ono, F.; Takizawa, K.; Miura, R. Multi-dimensional wireless tomography using tensor-based compressed sensing. *Wirel. Pers. Commun.* **2017**, *96*, 3361–3384. [CrossRef]
2. Wilson, J.; Patwari, N. Radio tomographic imaging with wireless networks. *IEEE Trans. Mob. Comput.* **2010**, *9*, 621–632. [CrossRef]
3. Beck, B.; Ma, X.; Baxley, R. Ultrawideband Tomographic Imaging in Uncalibrated Networks. *IEEE Trans. Wirel. Commun.* **2016**, *15*, 6474–6486. [CrossRef]
4. Deng, T.; Qian, F.; Liu, X.Y.; Zhang, M.; Walid, A. Tensor Sensing for Rf Tomographic Imaging. In Proceedings of the 2018 IEEE International Conference on Multimedia and Expo (ICME), Miami, FL, USA, 10–12 April 2018; pp. 1–6.
5. Cui, H.; Zhang, H.; Ganger, G.R.; Gibbons, P.B.; Xing, E.P. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In Proceedings of the Eleventh European Conference on Computer Systems, London, UK, 18–21 April 2016; pp. 1–16.
6. Brito, R.; Fong, S.; Song, W.; Cho, K.; Bhatt, C.; Korzun, D. Detecting Unusual Human Activities Using GPU-Enabled Neural Network and Kinect Sensors. In *Internet of Things and Big Data Technologies for Next Generation Healthcare*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 359–388.
7. Campos, V.; Sastre, F.; Yagües, M.; Torres, J.; Giró-i Nieto, X. Scaling a Convolutional Neural Network for Classification of Adjective Noun Pairs with TensorFlow on GPU Clusters. In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Madrid, Spain, 14–17 May 2017; pp. 677–682.
8. Shi, X.; Luo, X.; Liang, J.; Zhao, P.; Di, S.; He, B.; Jin, H. Frog: Asynchronous graph processing on GPU with hybrid coloring model. *IEEE Trans. Knowl. Data Eng.* **2018**, *30*, 29–42. [CrossRef]
9. Zhong, W.; Sun, J.; Chen, H.; Xiao, J.; Chen, Z.; Cheng, C.; Shi, X. Optimizing Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 1149–1162. [CrossRef]
10. Pan, Y.; Wang, Y.; Wu, Y.; Yang, C.; Owens, J.D. Multi-GPU graph analytics. In Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, USA, 29 May–2 June 2017; pp. 479–490.
11. Gutiérrez, P.D.; Lastra, M.; Benítez, J.M.; Herrera, F. SMOTE-GPU: Big Data preprocessing on commodity hardware for imbalanced classification. *Prog. Artif. Intell.* **2017**, *6*, 1–8. [CrossRef]
12. Rathore, M.M.; Son, H.; Ahmad, A.; Paul, A.; Jeon, G. Real-time big data stream processing using GPU with spark over hadoop ecosystem. *Int. J. Parallel Program.* **2017**, *46*, 1–17. [CrossRef]
13. Devadithya, S.; Pedross-Engel, A.; Watts, C.M.; Landy, N.I.; Driscoll, T.; Reynolds, M.S. GPU-Accelerated Enhanced Resolution 3-D SAR Imaging With Dynamic Metamaterial Antennas. *IEEE Trans. Microw. Theory Tech.* **2017**, *65*, 5096–5103. [CrossRef]
14. Verma, K.; Szeu, K.; Wille, R. Advanced load balancing for SPH simulations on multi-GPU architectures. In Proceedings of the High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 12–14 September 2017; pp. 1–7.
15. Intelligent Information Processing (IIP) Lab. Available online: <http://www.findai.com> (accessed on 15 February 2019).
16. Kanso, M.A.; Rabbat, M.G. Compressed RF tomography for wireless sensor networks: Centralized and decentralized approaches. In *International Conference on Distributed Computing in Sensor Systems*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 173–186.
17. Mostofi, Y. Compressive cooperative sensing and mapping in mobile networks. *IEEE Trans. Mob. Comput.* **2011**, *10*, 1769–1784. [CrossRef]

18. Li, Q.; Schonfeld, D.; Friedland, S. Generalized tensor compressive sensing. In Proceedings of the 2013 IEEE International Conference on Multimedia and Expo (ICME), San Jose, CA, USA, 15–19 July 2013; pp. 1–6.
19. Liu, X.Y.; Wang, X. Fourth-order tensors with multidimensional discrete transforms. *arXiv* **2017**. arXiv:1705.01576
20. Jing, N.; Jiang, L.; Zhang, T.; Li, C.; Fan, F.; Liang, X. Energy-efficient eDRAM-based on-chip storage architecture for GPGPUs. *IEEE Trans. Comput.* **2016**, *65*, 122–135. [[CrossRef](#)]
21. 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 Trans. Archit. Code Optim. (TACO)* **2015**, *12*, 1–23. [[CrossRef](#)]
22. Zhang, T.; Zhang, J.; Shu, W.; Wu, M.Y.; Liang, X. Efficient graph computation on hybrid CPU and GPU systems. *J. Supercomput.* **2015**, *71*, 1563–1586. [[CrossRef](#)]
23. Zhang, T.; Shu, W.; Wu, M.Y. CUIRRE: An open-source library for load balancing and characterizing irregular applications on GPUs. *J. Parallel Distrib. Comput.* **2014**, *74*, 2951–2966. [[CrossRef](#)]
24. 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: Berlin/Heidelberg, Germany, 2016; pp. 12–21.
25. Nelson, T.; Rivera, A.; Balaprakash, P.; Hall, M.; Hovland, P.D.; Jessup, E.; Norris, B. Generating efficient tensor contractions for gpus. In Proceedings of the 44th International Conference on Parallel Processing (ICPP), Beijing, China, 1–4 September 2015; pp. 969–978.
26. Shi, Y.; Niranjana, U.; Anandkumar, A.; Cecka, C. Tensor contractions with extended BLAS kernels on CPU and GPU. In Proceedings of the IEEE 23rd International Conference on High Performance Computing (HiPC), Kochi, India, 16–19 December 2016; pp. 193–202.
27. Antikainen, J.; Havel, J.; Josth, R.; Herout, A.; Zemcik, P.; Hautakasari, M. Nonnegative tensor factorization accelerated using GPGPU. *IEEE Trans. Parallel Distrib. Syst.* **2011**, *22*, 1135–1141. [[CrossRef](#)]
28. Lyakh, D.I. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Comput. Phys. Commun.* **2015**, *189*, 84–91. [[CrossRef](#)]
29. Hynninen, A.P.; Lyakh, D.I. cuTT: A high-performance tensor transpose library for CUDA compatible GPUs. *arXiv* **2017**. arXiv:1705.01598
30. Rogers, D.M. Efficient primitives for standard tensor linear algebra. In Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, ACM, Miami, FL, USA, 17–21 July 2016; p. 14.
31. Zou, B.; Li, C.; Tan, L.; Chen, H. GPUSensor: Efficient tensor factorization for context-aware recommendations. *Inf. Sci.* **2015**, *299*, 159–177. [[CrossRef](#)]
32. Li, J.; Ma, Y.; Yan, C.; Vuduc, R. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In Proceedings of the IEEE Workshop on Irregular Applications: Architecture and Algorithms (IA3), Salt Lake City, UT, USA, 13–18 November 2016; pp. 26–33.