Memory reduced non-Cartesian MRI encoding using the mixed-radix tensor product on CPU and GPU

Jyh-Miin Lin*, Grzegorz Kowalik, Jennifer A. Steeden and Vivek Muthurangu

Abstract—Multi-dimensional non-Cartesian MRI encoding using the precomputed interpolator can encounter the curse of dimensionality, in which the interpolator size exceeds the available memory on the parallel accelerators. Here we reformulate the multi-dimensional non-uniform fast Fourier transform (NUFFT) to a tensor form. The exponentially growing size of the fully precomputed interpolator can be reduced by tensor analysis. We propose a tree-like, mixed-radix tensor method which flexibly reduces the storage of the NUFFT. A parallel tensor product algorithm is proposed and tested with in vivo cardiac MRI data. Cross-architecture comparisons show that up to 88.1% and 62.4% memory savings are seen in 3D and 2D CINE MRI, respectively, subject only to a negligible loss of accuracy compared to the double-precision CPU version.

Index Terms—Mixed-radix, Tensor product, non-Cartesian MRI

I. INTRODUCTION

Magnetic resonance imaging (MRI) is a non-invasive imaging device involving signal formation and spatial encoding. Accelerated non-Cartesian MRI reconstructions have been actively studied in the past decade, while the increasing computing power of high-end graphics processing units (GPUs) allows real-time MRI reconstructions to be integrated into clinical scanning protocols [1]. Non-uniform fast Fourier transform (NUFFT) is the infrastructure of non-Cartesian imaging processing, and the need for 4D or higher dimensional NUFFT appears in the recent pursuit to acquire and reconstruct multi-dimensional MRI, exemplified by the tensor compressed sensing [2], [3], [4], MR multi-tasking [5], MR fingerprinting [6], and T2 shuffling [7]. These methods may be extended to incorporate multiparametric mapping into non-Cartesian compressed sensing.

However, the curse of dimensionality poses a challenge to multi-dimensional interpolation, which manifests as the exponential growth of memory with increased dimensions. Although this curse of dimensionality became less serious with the cheaper and larger system memory, the issue has recently resurfaced due to the limited size of high-speed memory on GPUs. This issue has been well known in multi-variate statistics and the previous NUFFTs, which reduced the computational complexity [8], [9] or the interpolator size [10] using tensor products on CPUs. Thus, the exponential growth of complexity or storage could be reduced to linear growth. The rationale is that the rank-1 tensor product is highly compressible [11], [12] and the run-time or the memory usage can be reduced.

The reason for moving to the tensor representation is that natural signals can be highly compressible when considering the dimensionality. For instance, a distributed memory considers a parallel Tucker decomposition of tensor [12]. The tensor form has recently been used in dictionary learning [13], tomography imaging [14], low-rank tensor completion [15]. The low-rank tensor is also used to enforce regional sparsity [16], thereby mitigating the incoherent noise in compressed sensing. Low-rank and sparsity tensors can reconstruct the spatiotemporally accelerated cardiovascular MRI [17]. The spatiotemporal tensor analysis for whole-brain fMRI classification is used to classify brain activities.

Analytic interpolation kernels have been the main approach used in accelerated NUFFTs. These analytic NUFFTs, based on Kaiser-Bessel or Gaussian kernels, have been implemented on GPU using the high-level shading language [18], and the recent parallel programming environments such as the vendor provided Compute Unified Device Architecture (CUDA) [19], [20], [21], [20], the open standard OpenCL [22], and OpenMP [20]. However, non-analytic adaptive NUFFT kernels [23], [24] have a longer tradition in MRI reconstructions. These adaptive NUFFT kernels can be numerically optimized to reduce the interpolation size, thereby saving computing resources. However, these adaptive kernels require two-stage computing. In the first stage, the kernel is generated on the CPU, and transferred to the GPU or CPU. In the second stage, the NUFFT can be executed on the GPU or CPU at high speed. However, this two-stage design can be obviated by the slow PCIe bus, or the limited memory size on the GPU, in which case the curse of dimensionality of multi-dimensional NUFFT arises. High-end GPUs with large memory are available at high costs, which are fairly rare in most clinical environments.

Here, we proposed a novel mixed-radix tensor product for multi-dimensional non-Cartesian MRI on the accelerators. Our implementation is running on the CPU and GPU using OpenCL and CUDA backends. The mixed-radix tensor product method allows flexible configurations, and the tensor products are recursively executed on the accelerator. This novel method is akin to the previous tenor product methods on the CPU and GPU [25], [19], [26], but here we aim to implement the multi-dimensional adaptive kernel on the GPU. We use to the adaptive min-max interpolator of [23], whereas other kernel functions may be precomputed and integrated into the mixed-radix tensor product method.

Our contribution lies in the novel cross-architecture, mixed-
radix tensor product implementation of NUFFT. The rest of this paper is organized as follows. In Section II we apply the Tucker product representations [17], [27] to NUFFT. We propose a novel mixed-radix tensor product form for multi-dimensional NUFFT. In Section III we describe the implementation, which is tested on the CPU and GPU. We apply the method to clinical cardiac imaging data and we test the consistency of the mix-radix tensor product method. Results are shown in Section IV. In Section V, we discuss the implications of this method in future studies.

II. METHODS

The problem setting of non-Cartesian MRI reconstruction has been well recognized as the following inverse problem:

\[ y = Ax + \epsilon \]

where \( y \) is the non-Cartesian samples, \( A \) is the Fourier encoding usually computed by NUFFT, \( x \) is the (flattened) image vector, and \( \epsilon \) is the noise vector.

In the following subsections, we formulate the inverse problem in the tensor product of Fourier integrals [28].

A. Tensor form of non-Cartesian Fourier encoding

Here, we generalize the Tucker product for Cartesian k-space [17], [4] to non-Cartesian k-space, using tensor notation in Kolda and Bader [27]. Throughout this paper, tensors are represented as mathematical fonts; matrices are written in bold fonts and upper case; vectors are expressed in bold fonts and lower case; scalars are written as regular fonts.

Multi-dimensional Fourier transform can be described as a \( D \)-mode matrix product of the tensor of interest, where each row vector of the factor matrix is the encoding at a k-space location. Each sampled point is the interaction between the encoding gradients and the image tensor.

First, the encoding matrix is expressed as the the tensor form of a Fourier integral [28]:

\[ Y_{1,1,1} = \mathcal{X}_{N_x,N_y,N_z} \times_1 E_{x1,N_x} \times_2 E_{y1,N_y} \times_3 E_{z1,N_z} + \mathcal{H} \]  

\( \mathcal{X} \) is a 3rd-order tensor of the distribution of magnetizations, \( E_x, E_y, E_z \) are the chosen row vectors of the discrete Fourier transform, \( \mathcal{H} \) is the noise tensor. The subscript indicates the dimension of the factor matrices (see Figure 1a for the illustration). The generic tensor form of Equation (2) is valid for Cartesian, non-Cartesian Fourier transform (e.g. the 3D MRI), and direct sampling (e.g. multi-slice 2D MRI or spatio-temporally accelerated 2D MRI).

Multiple \( M \) samples extend Equation (2) to the superdiagonal data tensor \( Y_{M,M,M} \):

\[ Y_{M,M,M} = \mathcal{X}_{N_x,N_y,N_z} \times_1 E_{xM,N_x} \times_2 E_{yM,N_y} \times_3 E_{zM,N_z} \]  

where the off-diagonal elements of \( Y \) are null (because they are not sampled in the imaging system). For simplicity, we omit the noise tensor \( \mathcal{H} \) hereafter. In Cartesian k-space, the row vectors of the phase encoding and frequency encoding matrices are periodical sinusoids, which match the order of the pulse sequence. In non-Cartesian k-space, the row-vectors of the encoding matrices are the instant spatial encodings of the spins. Non-Cartesian data can be gridded to Cartesian, as in Roohi et al [17], or they can be described as the tensor form of NUFFT in the next section.
B. The tensor form of non-uniform fast Fourier transform (NUFFT)

The 1D NUFFT can be represented as the following matrix form:

\[
E_{M,N} = V_{M,K} F_{K,N} W_{N,N}
\]  

\(N,K\) are the size of the image and k-space. \(M\) is the number of non-uniform samples. \(V_{M,K}\) is the interpolator (in the form of the sparse matrix); \(F_{K,N}\) is the oversampled FFT, \(W_{N,N}\) is the diagonal scale factor (also known as the roll-off) applied to the image.

In the full precomputation mode, the total memory for scaling factor is proportional to \(\prod_d N_d[d]\) \((N_d[d]\) is the matrix size of the \(d\)-th axis\), and the size of the interpolator is proportional to \(\prod_d J_d[d]\) \((J_d[d]\) is the interpolator size of the \(d\)-th axis\).

Combining Equations (3) and (4), we obtain the tensor form of 3D NUFFT:

\[
Y_{M,M,M} = X_{N_x,N_y,N_z} \times_1 W_{x,N_x,N_z} \times_2 W_{y,N_y,N_y} \times_3 W_{z,N_z,N_z} \times_1 F_{x,K_x,N_x} \times_2 F_{y,K_y,N_y} \times_3 F_{z,K_z,N_z} \times_1 V_{x,M,K_x} \times_2 V_{y,M,K_y} \times_3 V_{z,M,K_z}
\]  

(5)

The scaling factors \(W_{x,N_x,N_z}, W_{y,N_y,N_y}, \) and \(W_{z,N_z,N_z}\) are the diagonal matrix. The interpolators \(V_{x}, V_{y}, V_{z}\) are the matrix of the composite 1D vectors.

4D or higher dimensions can be derived from Equation (5). We omit the details.

C. Our contribution

We construct the mixed-radix NUFFT based on the tensor form in Equation (5).

1) Partial precomputation and tensor product method:

Given the rank-1 nature of the scaling factor and the interpolator in Equation (3) the curse of dimensionality can be alleviated by partial precomputation. The rationale of partial precomputation is to save the scaling factor and the interpolation matrix as composite 1D vectors (for scaling factors), or stacks of vectors (for interpolators). This strategy has previously been applied to CPU [8], [10], and we go on to apply the method to heterogeneous parallel accelerators (CPU and GPU). For completeness, the scaling factor and interpolator are described in detail below.

2) Scaling factor: The scaling factor (also known as the roll-off or deapodization) is a real-value multivariate function, which multiplies the image and compensates for the imperfections of the interpolators. Partial precomputation can be applied to the scaling factor because it is strictly rank-1. Thus, the marginal values of the scaling factors can be used to construct the scaling factor.

In the current implementation, the composite 1D scaling factors are saved in the continuous 1D array. Given the image tensor size \(Nd\), where the vector length of \(d\)-th dimension is \(Nd[d]\), the total length of the 1D array is \(\sum_d Nd[d]\) floating-point real numbers. The run-time computation multiplies the marginal values from the 1D vectors, and the final value is obtained from the joint density.

A feature of partial precomputation for the scaling factor is that the compression ratio increases with a large problem size and high image dimensions. For example, the compression ratio is 128 for a 2D \(256 \times 256\) matrix, which is higher than 64 for a \(128 \times 128\) matrix. The compression ratio is 21,845 for a \(256 \times 256 \times 256\) 3D NUFFT problem.

3) Interpolator: The tensor form of NUFFT in Equation (5) can be used to reduce the storage of multi-dimensional NUFFT (see Figure 2 for a diagrammatic example). Here, each composite interpolator includes two matrices: one data matrix (we call it \(udata\)) and one index matrix (we call it \(kindx\)) not shown in the figure. The matrix sizes of \(udata\) and \(kindx\) are \(Nd[d] \times Jd[d]\), where the address to the 1D array is easily obtainable. The multi-dimensional indices to the \(\prod Jd\) are stored in a meshindex matrix.

During the run-time, each thread reads the meshindex and then computes the k-space index and interpolator value. The interpolator values of the multi-dimensional interpolation matrix are multiplied by the k-space data. The resulting value of each interpolation point is saved in local memory, followed by a final reduction to compute the final value. The adjoint operation of the interpolation matrix reverses the process of computing the k-space index and the conjugate of the interpolator value, but the final values of the adjoint operation are written to the global memory using the atomic operations.

The pseudocodes are listed in Algorithms 1 and 2. The overall run-time complexity of partial precomputation is \(O((D+1)M \sum_d Jd[d])\). The total storage for the interpolators is \(M \sum_d Jd[d]\). Therefore, the memory sizes for the indexing \(kindx_{m,indptr}\) and interpolators \(udata_{m,indptr}\) are approximately proportional to \(M \sum_d Jd[d]\), which is smaller than \(M \prod_d Jd[d]\) in the fully precomputation mode. Table 1 compares the memory and complexity of a full computed interpolator and a partial precomputed interpolator.

![Fig. 2. Radix-1 3D tensor reconstruction of the multi-dimensional interpolator. a) Parallel tensor reconstruction of the interpolator b) Parallel tensor reconstruction of the regridding matrix. Each non-Cartesian interpolator is a rank-1 tensor, which allows for parallel tensor compressed interpolation on the GPU. Mixed-radix tensor compression is based on the identical kernel.](image-url)
Algorithm 1: Algorithm for radix-1 interpolation

Input: Jd, D, prodJd, sumJd, meshindex[prodJd * D], kindx[M * sum(Jd)], udata[M * sum(Jd)]

Output: y[M]

Initialisation:

1: for m = 0 to M do
2:   for j = 0 to prodJd do
3:     J = Jd[0]
4:     index_shift = m * sumJd
5:     index = index_shift + meshindex[dim * j]
6:     col = kindx[index]
7:     spdata = udata[index]
8:     index_shift += J
9:   end for
10: J = Jd[d]
11: index = index_shift + meshindex[dim * j + d]
12: col += kindx[index] + 1
13: spdata* = udata[index]
14: index_shift += J
15: end for
16: y[m] = vec[col] * spdata
17: end for
18: end for

Algorithm 2: Algorithm for radix-1 adjoint interpolation

Input: Jd, D, prodJd, sumJd, meshindex[prodJd * D], kindx[M * sum(Jd)], udata[M * sum(Jd)]

Output: y[M]

Initialisation:

1: for m = 0 to M do
2:   u = 0.0
3:   for j = 0 to prodJd do
4:     J = Jd[0]
5:     index_shift = m * sumJd
6:     index = index_shift + meshindex[dim * j]
7:     col = kindx[index]
8:     spdata = udata[index]
9:     index_shift += J
10: for d = 1 to D do
11:   J = Jd[d]
12:   index = index_shift + meshindex[dim * j + d]
13:   col += kindx[index] + 1
14:   spdata* = udata[index]
15:   index_shift += J
16: end for
17: ydata = y[m]
18: u = conj(spdata) * ydata
19: atomicAdd(k[col], u)
20: end for
21: end for

4) Mixed-radix partial precomputation: The mixed-radix precomputation is based on the associative property of the tensor forms in Equation (5). During the precomputation stage, the mixed-radix configuration can group several dimensions together; then the groups are computed on the parallel accelerators at the run-time stage.

The precomputation process can be recursively applied to a selected subset of the dimensions, and the intermediate states encompass a tree-like diagram. For example, a 3D hypercube undergoing the radix-2-1 tensor precomputation can be represented as (0, 1, 2) − > (01, 2). In another example, a 4D tesseract can be precomputed as (0, 1, 2, 3) − > (012, 3). The complete tree-like diagram can be seen in Figure 3 in which radix controls the order in which the partial precomputation takes place. We can represent the full precomputed NUFFT as radix = D precomputation because the D dimensions are collapsed during the precomputation, and there is only one subset in the radix = D configuration. The partial precomputation method in the above sections can, however, be attributed as radix = 1. The other configurations follow the same rule.

D. Parallel imaging with tensor compression of the coil axis

Parallel imaging is the standard scanning protocol of modern MRI. The multi-channel data are acquired simultaneously, and the structure of multi-channel data provides the extra information. In the sensitivity encoding, the coil sensitivity is
provided as the weighting function for the conjugate gradient method.

The parallel imaging encoding can be interpreted as a multi-channel image tensor weighted by the coil sensitivities (element-wise multiplication), followed by non-Cartesian Fourier encoding. We define the process using the following equation:

$$S \odot D\{\mathbf{x}\} = S \odot (X_{N_x \times N_y \times N_z,1} \times 4 e_{N_c,1}) \quad (6)$$

$S \odot$ is the element-wise multiplication of the coil sensitivities, $D \equiv \times 4 e_{N_c,1}$ is the populate operator, $X$ is the single-coil 3D image; $e \equiv [1, 1, ..., 1]^T$ is the $N_c \times 1$ matrix (a column vector).

We also define the aggregate operator $M$, which combines the multi-channel image tensor $X \odot \text{conj}(S)$:

$$M\{X \odot \text{conj}(S)\} = (X_{N_x \times N_y \times N_z,1} \odot \text{conj}(S)) \times 4 e^T/N_c \quad (7)$$

$e^T/N_c = [1/N_c, 1/N_c, ..., 1/N_c]$ is a length-$N_c$ row vector.

The parallel computing for the populate and aggregate operators on CPU and GPU is as follows. In the populate operation, each thread reads the value of the image voxel and populates the value to multi-channel image volumes. Once the copying is finished, the volumes are copied to an oversampled grid, followed by batched NUFFT. In the aggregate operation, the same voxels of multi-channel image volumes are saved in the local memory, and a final hierarchical reduction averages the values (see Figure 4 for a graphic illustration of the populate and aggregate operators on the GPU).

### III. Implementations

#### A. Mixed-radix multi-dimensional interpolator and scaling factor

The implementation uses the two-stage method, as the non-Cartesian interpolation matrix and the scaling factors are planned on the CPU then transferred to the GPU. At the planning stage, the interpolators and scaling factors are designed with the complex double-precision floating point numbers. The arrays are cast to single-precision and transferred to the global memory. The run-time computation uses the same kernel for different configured mixed-radix interpolators and scaling factors.

|                      | Precomputed (radix-D) interpolator | Radix-1 tensor compressed interpolator | Mixed-radix tensor compressed interpolator |
|----------------------|------------------------------------|----------------------------------------|--------------------------------------------|
| Scaling factor       | $\prod_{d=1}^D N_d[d](\text{float32})$ | $\sum_{d=1}^D N_d[d](\text{complex64})$ | $\text{total storage of subsets}$          |
| Interpolator size    | $M \prod_{d=1}^D J_d[d](\text{complex64})$ | $M \sum_{d=1}^D J_d[d](\text{complex64})$ | $\text{total storage of subsets}$          |
| Indices              | $M \prod_{d=1}^D J_d[d]$ | $M \sum_{d=1}^D J_d[d]$ | $\text{total storage of subsets}$          |

**TABLE I**

**COMPARISONS OF STORAGE AND RUN-TIME PERFORMANCE FOR PRECOMPUTED AND TENSOR COMPRESSED NUFFTS**

- **Scaling factor**
- **Interpolator size**
- **Indices**
- $M$: Number of non-uniform locations
- $J_d[d]$: Interpolation size for $d$-th dimension
- $N_d[d]$: Matrix size for $d$-th dimension
- $K_d[d]$: Oversampled k-space grid for $d$-th dimension

#### B. PyOpenCL and PyCUDA backends

The acceleration environment using PyOpenCL or PyCUDA [29] is freely available under the permissive MIT license. We update the NUFFT_hsa object of the PyNUFFT [30], and tensor related Hadamard products and mixed-radix multi-dimensional interpolation are also available in the re_subroutine.py at the PyNUFFT github repository.

To address the missing atomicAdd() in OpenCL-1.2, we implemented an atomicAdd using the atomic-compare-and-
swap (atomicCAS) function in OpenCL-1.2. CUDA provides its atomicAdd() for a single-precision floating points number.

C. Benchmarks of NUFFT

We tested the 2D and 3D acceleration of NUFFT based on acceleration packages of PyCUDA 2018.1.2, PyOpencl 2018 and Reikna 0.8. The testing system is equipped with an i7-6700HQ CPU with 32 GB memory, one NVIDIA GeForce GTX 1060 6GB and one external NVIDIA Quadro P6000 connected to the system with a PCIe-2 4× connector. The versions of the software on the testing computer are gcc-7.3.0, nvidia-driver 415.18, cuda version 9.2.88 and Intel OpenCL sdk 2014.

D. Spatio-temporal reconstruction of golden-angle radial CINE MRI

Radial k-space combined with fast gradient echo sequences is a rapid MRI acquisition mode widely used in cardiovascular MRI. The golden-angle radial spokes are continuously filling the k-space without repeating the previous angles. The radial sampling artifacts may be reduced by iterative reconstructions. Here, we test the mixed-radix tensor product with the soft-plus-hard reconstruction [31], which decomposes the reconstruction into the low-frequency part and the sparsity part. This concept is akin to the low-rank plus sparse method [32], but we develop the method based on the temporally constrained method [33] because of its simplicity and nearly deterministic results. Our method can be summarized as follows. While the low-frequency part is regularized by the hard-thresholding, the sparsity part is regularized by soft-thresholding.

The purpose of the test is to validate the results of single-precision version of mixed-radix tensor product are close to the double-precision CPU counterpart. We construct the encoding operator as a 4D tesseract, which is composed of 2 spatial dimensions, one time dimension, and one coil dimension. This 4D encoding process is formulated as follows:

\[ Y_{M,M,Nc} = X \times_1 E_x \times_2 E_y \times_3 E_t \times_4 E_c \]  

(8)

This 4D problem can be solved by the tensor form represented in Figure 6 a) the 4D tesseract represents the problem (x-y-t-coil) composed of repeated k-t sampling patterns for multiple coils; and b) the batched 3D problem (x-y-t) based on the single-coil k-t sampling patterns. This batched mode can be considered as a tensor compression of the coil dimension.

In the 4D tesseract configuration, the axis of the coil is considered the fourth-dimension, but this increases the size of the memory. In batched 3D mode, the axis of coil is compressed inside the kernel, which reduces the memory.

E. Consistency test

In the consistency test, the identical reconstruction algorithm is performed on different accelerators. Computations using single-precision floating numbers may affect the accuracy of the image reconstructions. However, these differences may not be perceivable to human eyes, so we compare the image quality using the structural similarity index (SSIM) (MATLAB, The MathWorks, Inc., Natick, MA, USA). Both the PyOpenCL and PyCUDA backends are used for tests on GPU, whereas the PyOpenCL is used for multi-core CPU. Image regularizations and iterations are performed on the system CPU. Thus, the acceleration is attributed to different accelerators and these reconstruction results are compared to the images reconstructed by GPU using image quality measures of the normalized Frobenius norm and SSIM. The testing computer is equipped with an i7-6700HQ CPU and 32 GB memory, and one discrete NVIDIA GeForce GTX 1060 6GB video card. Another testing platform connects to an external NVIDIA Quadro P6000 with error-correcting code (ECC) memory available.

IV. RESULTS

A. Mixed-radix tensor operations on the accelerators

Figure 5 compares the accelerations of mixed-radix 3D NUFFT on GPU (Figure 5a) and CPU (Figure 5b) versus the single-core, fully precomputed compressed sparse row format on the CPU. GPU provides a fast acceleration than CPU, and the accelerations of forward NUFFTs are consistently higher than adjoint NUFFTs on the GPU and CPU. The acceleration of single-coil forward NUFFT is about 7× and 10× faster than adjoint NUFFT on GPU and CPU, respectively. However, the performances of the adjoint NUFFT and the full selfadjoint NUFFT improve with multi-coil computing. The accelerations of 32 coils are 8× and 4× higher than single coil. This is because the CPU atomic operations used in adjoint NUFFT are slow, whereas the multi-coil atomic operations reduce the collisions because the values of two coils are written to different locations.

A higher degree of precomputation is faster than the lower degree of precomputation. The radix-3 with full precomputation is the fastest, followed by radix-2-1 and radix-1. The only exception is the adjoint NUFFT with 1-4 coils on CPU, where the acceleration of radix-1 outperforms radix-2-1 and radix-3.

Table II compares the global memory of 3D mixed-radix configurations. A highest degree of precomputation (radix-3) requires large memory, while the lowest degree of precomputation (radix-1-1-1) largely save the memory. 77.4% and 88.1% memory spaces are saved in radix-2-1 and radix-1-1-1, respectively.

B. Benchmarks of golden-angle radial MRI

The acceleration of the image reconstruction of the golden-angle radial MRI is shown in Figure 7. The comparison is performed using the OpenCL backend. Thus, any performance difference can be attributed to the hardware.

The batched 3D reconstruction is significantly accelerated by GPU. The Quadro P6000 and the GTX 1060 6GB GPU deliver over 18× and 13× faster reconstructions respectively than the single-threaded CPU, while the Intel CPU 6700HQ is nearly 2.5× faster than the single-threaded CPU. The 4D reconstruction on GPU is about 11× and 10× faster for Quadro P6000 and GTX 1060, respectively. However, the 4D CPU is only marginally slower than batched 3D, since the acceleration is nearly 2×.
The degree of precomputation marginally impacts on the performance. For instance, the radix-3 is slightly faster than the radix-2-1 or radix-1-1-1, but the radix-4, radix-3-1, radix-2-2, and radix-1-1-1-1 achieve very similar degrees of acceleration. The memory saving of mixed-radix configurations can be seen in Table III in which radix-1-1-1 saving 62.4% of the global memory.

C. Consistency tests for golden-angle radial cardiac MRI

Figure 6 compares the image reconstructed using OpenCL and CUDA backends and a differently configured radix. These images are compared to the referenced image reconstructed using the double precision floating number on the CPU.

The results of the soft-plus-hard reconstruction consistently differ between OpenCL and CUDA. A higher error in CUDA can be measured in the normalized Frobenius error norm, which is approximately $10 \times$ higher than the error norm of OpenCL. The reconstructed images of OpenCL on GPU and CPU lead to the error of $7.56 \times 10^5 - 2.56 \times 10^4$, while the error of CUDA on GPU is $1.9 \times 10^3 - 2.0 \times 10^3$.

The OpenCL backend leads to the reconstruction results which are nearly identical to those of the double-precision CPU computing. The SSIM values of OpenCL on CPU and GPU are constantly 1. However, the $10 \times$ higher error norm using CUDA backends consistently degrades the reconstructed image quality of CUDA. Thus, the SSIM between CPU and CUDA is consistently lower than 1, in the range of $0.9989 - 0.9989$.  

| Num of Samples | 144768 |
|----------------|--------|
| time frames    | 29     |
| oversampling ratio | 2 x    |
| oversampled grid | 384 x 384 |
| interpolator size | 6 x 6  |

The OpenCL backend leads to the reconstruction results which are nearly identical to those of the double-precision CPU computing. The SSIM values of OpenCL on CPU and GPU are constantly 1. However, the $10 \times$ higher error norm using CUDA backends consistently degrades the reconstructed image quality of CUDA. Thus, the SSIM between CPU and CUDA is consistently lower than 1, in the range of $0.9989 - 0.9989$.  

## Table III

|                  | radix-4 | radix-3-1 | radix-2-2 | radix-1-1-1 | radix-3(batch) | radix-2-1(batch) | radix-1-1-1(batch) |
|------------------|---------|-----------|-----------|-------------|----------------|------------------|--------------------|
| Scaling factor (MB) | 25.7    | 4.3       | 0.148     | 0.0017      | 4.3            | 0.147            | 0.00165            |
| Interpolator (MB)  | 250.2   | 257.1     | 257.1     | 97.3        | 41.7           | 42.9             | 15.1               |
| Oversampled grid (MB) | 205.3   | 205.3     | 205.3     | 205.3       | 205.3          | 205.3            | 205.3              |
| Indices (MB)       | 125.1   | 128.6     | 128.7     | 48.6        | 20.8           | 21.4             | 7.5                |
| Total storage (MB) | 606.2   | 595.2     | 591.1     | 351.2       | 272.1          | 269.7            | 227.8              |
| Memory saving (%)  | 0%      | 1.8%      | 2.5%      | 42.1%       | 55.1%          | 55.5%            | 62.4%              |
Fig. 6. Consistency tests for multi-coil cardiac image reconstruction methods using OpenCL and CUDA backends. An identical regularization method with 200 iterations was performed on the CPU. a) 4D reconstruction (x-y-t-coil) using different configurations. Multiple coils are encoded in the last dimension; b) batched 3D reconstruction (x-y-t) using different configurations. The coil dimension is compressed on the accelerator. The error of CUDA backend is approximately 10× higher than the error of OpenCL on the CPU and GPU.

0.9992. We also enabled the ECC functionality of CUDA but we still observed the consistently degraded image quality in the CUDA backend.

V. DISCUSSION AND CONCLUSION

We have developed a novel cross-architecture mix-radix tensor product algorithm with a reduced memory usage. The PyNUFFT was previously implemented such that multiple objects can be planned and dispatched to different types of accelerators, including CUDA/OpenCL and CPU/GPU. The dual OpenCL and CUDA implementations may allow an identical imaging reconstruction algorithm to be tested with different backends and hardware. However, the complexity of portable software is nontrivial [34], and the inconsistencies are often difficult to track. We observed that the results of cross-architecture using OpenCL are more consistent than with the CUDA backend. Although the inconsistencies between CPU and accelerators may be negligible to human eyes, the degraded SSIM values of CUDA are consistently detected throughout our tests. The result suggests that algorithms ported to different acceleration protocols may require consistency tests before clinical use.

The future study may integrate Python-based reconstructions with deep-learning frameworks. Iterative tensor-based NUFFT could be integrated with a multi-dimensional learning method to generate the multi-coil non-Cartesian k-space samples or the Toeplitz kernel. Python is one of the leading programming languages, and Python-based MRI reconstruction packages, such as PySAP (https://github.com/CEA-COSMIC/pysap) and MRIPY [35], have recently drawn much interest from the industry. This research approach can benefit from the general purpose Python programming language integrated with heterogeneous acceleration devices, as well as the plentiful software packages available in Python language.

ACKNOWLEDGMENT

We thank the NVIDIA Corporation for its donation of a Quadro P6000 GPU to test the algorithm. We thank Mr. Andreas Hauptmann, Dr. Felix Lucka and Professor Simon Arridge for their constructive feedback. The mixed-radix method is inspired by the email (Benoit Sarthou, personal communication, April 9, 2018).
REFERENCES

[1] M. Murphy, M. Alley, J. Demmel, K. Keutzer, S. Vaswaniawa, and M. Lustig, “Fast $\ell_1$-SPIRiT compressed sensing parallel imaging MRI: Scalable parallel implementation and clinically feasible runtime,” IEEE Transactions on Medical Imaging, vol. 31, no. 6, pp. 1250–1262, 2012.

[2] N. D. Sidiropoulos and A. Kyriillidis, “Multi-way compressed sensing for sparse low-rank tensors,” IEEE Signal Processing Letters, vol. 19, no. 11, pp. 757–760, 2012.

[3] Q. Li, D. Schonfeld, and S. Friedland, “Generalized tensor compressive sensing,” in 2013 IEEE International Conference on Multimedia and Expo (ICME). IEEE, 2013, pp. 1–6.

[4] C. F. Ciafa and A. Cichocki, “Multidimensional compressed sensing and their applications,” Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 3, no. 6, pp. 355–380, 2013.

[5] A. G. Christodoulou, J. L. Shaw, C. Nguyen, Q. Yang, Y. Xie, N. Wang, and D. Li, “Magnetic resonance multisensing for motion-resolved quantitative cardiovascular imaging,” Nature Biomedical Engineering, vol. 2, no. 4, p. 215, 2018.

[6] D. Ma, V. Gulani, N. Seiberlich, K. Liu, J. L. Sunshine, J. L. Duerk, and M. A. Griswold, “Magnetic resonance fingerprinting,” Nature, vol. 495, no. 7440, p. 187, 2013.

[7] J. I. Tumir, M. Uecker, W. Chen, P. Lai, M. T. Alley, S. S. Vaswaniawa, and M. Lustig, “2T shuffling: sharp, multicontrast, volumetric fast spin-echo imaging,” Magnetic Resonance in Medicine, vol. 77, no. 1, pp. 180–195, 2017.

[8] L. Greengard and J.-Y. Lee, “Accelerating the nonuniform fast fourier transform,” SIAM review, vol. 46, no. 3, pp. 443–454, 2004.

[9] F. Lekien and J. Marsden, “Tricubic interpolation in three dimensions,” International Journal for Numerical Methods in Engineering, vol. 63, no. 3, pp. 455–471, 2005.

[10] J. Keimer, S. Kunis, and D. Potts, “Using fft3—a software library for various nonequispaced fast fourier transforms,” ACM Transactions on Mathematical Software (TOMS), vol. 36, no. 4, p. 19, 2009.

[11] N. Verliefet, O. Debals, L. Sorber, and L. De Lathauwer, “Breaking the curse of dimensionality using decompositions of incomplete tensors: Tensor-based scientific computing in big data analysis,” IEEE Signal Processing Magazine, vol. 31, no. 5, pp. 71–79, 2014.

[12] W. Austin, G. Ballard, and T. G. Kolda, “Parallel tensor compression for large-scale scientific data,” in Parallel and Distributed Processing Symposium, 2016 IEEE International. IEEE, 2016, pp. 912–922.

[13] S. Soltani, M. E. Kilmer, and P. C. Hansen, “A tensor-based dictionary learning approach to tomographic image reconstruction,” BIT Numerical Mathematics, vol. 56, no. 4, pp. 1425–1454, 2016.

[14] S. Tan, Y. Zhang, G. Wang, X. Mou, G. Cao, Z. Wu, and H. Yu, “Tensor-based dictionary learning for dynamic tomographic reconstruction,” Physics in Medicine & Biology, vol. 60, no. 7, p. 2803, 2015.

[15] S. Gandy, B. Recht, and I. Yamada, “Tensor completion and low-n-rank tensor recovery via convex optimization,” Inverse Problems, vol. 27, no. 2, p. 025010, 2011.

[16] J. D. Trzasko, “Exploiting local low-rank structure in higher-dimensional MRI applications,” in Wavelets and Sparsity XV, vol. 8858. International Society for Optics and Photonics, 2013, p. 885821.

[17] S. F. Roohi, D. Zonoobi, A. A. Kassim, and J. L. Jaremko, “Multi-dimensional low rank plus sparse decomposition for reconstruction of under-sampled dynamic mri,” Pattern Recognition, vol. 63, pp. 667–679, 2017.

[18] T. S. Sørensen, T. Schaeffter, K. Ø. Noe, and M. S. Hansen, “Accelerating the nonequispaced fast Fourier transform on commodity graphics hardware,” IEEE Transactions on Medical Imaging, vol. 27, no. 4, pp. 538–547, 2008.

[19] D. S. Smith, S. Sengupta, S. A. Smith, and E. Brian Welch, “Trajectory optimized NUFFT: Faster non-Cartesian MRI reconstruction through prior knowledge and parallel architectures,” Magnetic Resonance in Medicine, 2018.

[20] A. H. Barnett, J. F. Magland, and L. a. Klintenberg, “A parallel non-uniform fast fourier transform library based on an exponential of semicircle kernel,” arXiv preprint arXiv:1808.06736, 2018.

[21] G. T. Kowelik, J. A. Steeden, B. Pandya, F. Odille, D. Atkinson, A. Taylor, and V. Muthurangan, “Real-time flow with fast GPU reconstruction for continuous assessment of cardiac output,” Journal of Magnetic Resonance Imaging, vol. 36, no. 6, pp. 1477–1482, 2012.

[22] U. I. Cheema, G. Nash, R. Ansari, and A. Khokhar, “Memory-optimized re-gridding architecture for non-uniform fast fourier transform,” IEEE Transactions on Circuits and Systems I: Regular Papers, 2017.

[23] J. A. Fessler and B. P. Sutton, “Nonuniform fast fourier transforms using min-max interpolation,” IEEE Transactions on Signal Processing, vol. 51, no. 2, pp. 560–574, 2003.

[24] Z. Yang and M. Jacob, “Mean square optimal NUFFT approximation for efficient non-Cartesian MRI reconstruction,” Journal of Magnetic Resonance Imaging, vol. 242, pp. 126–135, 2014.

[25] F. Knoll, A. Schwarzl, C. Diwoky, and D. K. Sodickson, “gpuNUFFT— an open source GPU library for 3D regridding with direct MATLAB interface,” in International Society for Magnetic Resonance in Medicine: Scientific Meeting & Exhibition. , 2014.

[26] J.-M. Lin, G. T. Kowellik, J. Montalt-Tordera, B. Sarthou, P. Cucci, J. A. Steeden, and V. Muthurangan, “(accepted) application of memory reduced NUFFT to multi-dimensional non-Cartesian MRI,” in Proceedings of the 27th Annual Meeting of International Society for Magnetic Resonance in Medicine, Montréal, QC, Canada, 2019.

[27] T. G. Kolda and B. W. Bader, “Tensor decompositions and applications,” SIAM review, vol. 51, no. 3, pp. 455–500, 2009.

[28] P. Osgood, Lecture Notes for EE 261 the Fourier Transform and Its Applications. CreateSpace Independent Publishing Platform, 2014. [Online]. Available: https://books.google.co.uk/books?id=om1srQ6EACAAJ

[29] A. Klöckner, N. Phito, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA and PyOpenCL: A scripting-based approach to GPU runtime code generation,” Parallel Computing, vol. 38, no. 3, pp. 157–174, 2012.

[30] J.-M. Lin, “Python non-uniform fast Fourier transform (PyNUFFT): An accelerated non-Cartesian MRI package on a heterogeneous platform (CPU/GPU),” Journal of Imaging, vol. 4, no. 3, p. 51, 2018.

[31] J.-M. Lin, S. Jennifer, and V. Muthurangan, “A soft plus hard temporally constrained reconstruction for accelerated non-Cartesian CINE MRI,” in Proceedings of the SCMR 22nd Scientific Sessions, Bellevue, WA, USA, 2018.

[32] R. Otazo, E. Candès, and D. K. Sodickson, “Low-rank plus sparse matrix decomposition for accelerated dynamic MRI with separation of background and dynamic components,” Magnetic Resonance in Medicine, vol. 73, no. 3, pp. 1125–1136, 2015.

[33] G. Adluru, S. P. Awate, T. Tadizzen, R. T. Whitaker, and E. V. DiBella, “Temporally constrained reconstruction of dynamic cardiac perfusion MRI,” Magnetic Resonance in Medicine, vol. 57, no. 6, pp. 1027–1036, 2007.

[34] M. Leeser, J. Ramachandran, T. Wahl, and D. Yablonski, “OpenCL floating point software on heterogeneous architectures—portable or not,” in Workshop on Numerical Software Verification (NSV), 2012.

[35] P. Cao, X. Zhu, J. Liu, Y. Wang, and P. Larson, “A Python-based MRI reconstruction toolbox, Mripy, for compressed sensing, parallel imaging and machine learning,” in Proceedings of the 26th Annual Meeting of International Society for Magnetic Resonance in Medicine, Paris, France, 2018.