TorchKbNufft: A High-Level, Hardware-Agnostic Non-Uniform Fast Fourier Transform

Matthew J. Muckley[1], Ruben Stern[1], Tullie Murrell[2], and Florian Knoll[1]

1. Radiology, NYU School of Medicine, New York, NY, 2. Facebook AI Research, Menlo Park, CA

Introduction: The non-uniform Fast Fourier Transform[1,2] (NUFFT) is a critical operation for reconstruction from MRI scanner data when using non-Cartesian k-space sampling trajectories. Due to its complexity, the NUFFT is usually the bottleneck in non-Cartesian reconstruction compute time. Implementations have prioritized speed over robustness and ease-of-use, typically consisting of compiled C[3] or CUDA[4] code. However, recent advances in software frameworks such as PyTorch allow indirect access to low-level application-programmer interfaces (APIs) across a variety of hardware platforms in a unified framework. Specifically, PyTorch provides tools that can be seamlessly implemented on either CPUs or GPUs while maintaining a single high-level code base, facilitating easy deployment and maintainability. This project implements `torchkbnufft`, a pure PyTorch Kaiser-Bessel NUFFT. Its structure in the PyTorch framework allows it to take advantage of the latest advances in GPU architectures in addition to seamless integration for training neural network reconstruction models.

Description: Primary interfaces into the package are implemented PyTorch module objects. These objects can be initialized with a minimal set of parameters – for these cases, simply putting in a tuple with image dimensions. Once initialized, the objects can be used to perform NUFFT operations. Subroutines are are wrapped in automatic differentiation functions, which facilitates training of neural networks with NUFFT operators with minimal memory overhead. The code along with example Jupyter notebooks are stored at https://github.com/mmuckley/torchkbnufft. The package can be installed on a system with Python via a one-line pip command: `pip install torchkbnufft`.

```
from torchkbnufft import AdjKbNufft
adjnufft_ob = AdjKbNufft(im_size=(256, 256))
# run on CPU
image = adjnufft_ob(kdata, ktraj)
# run on GPU
device = torch.device('cuda')
adjnufft_ob = adjnufft_ob.to(device)
image = adjnufft_ob(kdata.to(device), ktraj.to(device))
```

Figure 1: An example of module initialization and NUFFT adjoint operations on the CPU or GPU.

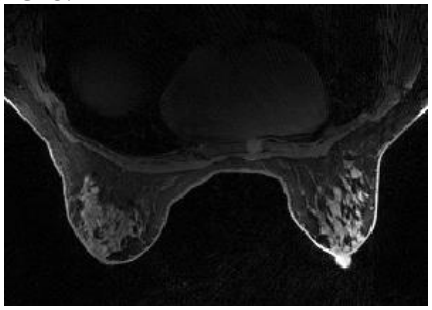

Figure 2: An example of breast perfusion data reconstructed with the package (16 coils, 400 radial spokes).

Experiments: To evaluate the effectiveness of the implementation, we compared computation speed on 15-coil, 405-spoke radial multicoil problem with reference implementations for CPU and the GPU. The length of each spoke was 512, and the image used was a 256 x 256 Shepp-Logan phantom. Experiments were performed on a workstation with an Intel Xeon E5-1620 4-core CPU and an Nvidia GeForce 1080 GPU running CUDA v10.1 and PyTorch v1.3. Two implementation frameworks were considered for `torchkbnufft`: a standard implementation that applies table interpolation, and an alternative implementation that precomputes the sparse interpolation matrix. We conducted speed tests and compared to reference implementations with binaries compiled in C and CUDA – results in Table 1.

Table 1: Comparison of computation speed between `torchkbnufft` and reference CPU[3] and GPU[4] implementations. All times are shown in seconds.

Operation	CPU Ref[3]	CPU Ours	CPU Ours (precomputation)	GPU Ref[4]	GPU Ours	GPU Ours (precomputation)
Forward NUFFT	1.1600	3.4947	3.4440	0.0645	0.1040	0.1037
Adjoint NUFFT	1.4000	10.9971	1.0880	0.0574	0.6107	0.1759

Table 1 shows that in order to approach the performance of compiled binaries, some interaction with the PyTorch sparse matrix API is important. Although speed was not the first priority for development, when using sparse matrices, `torchkbnufft` can actually surpass the references for some operations. We also found that performance depended heavily on problem size – for example, the computation time of a forward/backward operations of the GPU `torchkbnufft` and the GPU reference became roughly equal at problem sizes that considered 64 coils on a 256 x 256 image grid (0.5850 seconds for the reference and 0.6270 seconds for `torchkbnufft`). Sparse matrix multiplications remain an area of active development in PyTorch and CUDA, and further advances may give speed improvements for the software framework.

Summary: We presented `torchkbnufft`, a high-level NUFFT interface which can be used a subroutine in training neural networks. The implementation is easy to deploy and platform-agnostic, allowing rapid prototyping of new non-Cartesian image reconstruction methods.

References: 1. J.A. Fessler and B.P. Sutton, *IEEE-TMI* 2003. 2. P.J. Beatty et al., *IEEE-TMI* 2005. 3. J.A. Fessler et al., "Michigan Image Reconstruction Toolbox," available at web.eecs.umich.edu/~fessler/code/. 4. A. Schwarzl and F. Knoll, "gpuNUFFT," available at github.com/andyschwarzl/gpuNUFFT/. **Support:** NIH R01 EB024532 and P41 EB017183.