CufftShift: High performance CUDA-accelerated FFT-shift library

Conference Paper · April 2014

CITATIONS
7

READS
320

1 author:

Marwan Abdellah
École Polytechnique Fédérale de Lausanne
30 PUBLICATIONS **314** CITATIONS

SEE PROFILE

CUFFTSHIFT
High Performance CUDA-accelerated FFT-Shift Library

Marwan Abdellah
École Polytechnique Fédérale de Lausanne (EPFL)
Switzerland
marwan.abdellah@epfl.ch

Keywords: FFT-Shift, CUDA, GPU, CUFFT, X-ray Volume Rendering.

Abstract
For embarrassingly parallel algorithms, a Graphics Processing Unit (GPU) outperforms a traditional CPU on price-per-flop and price-per-watt by at least one order of magnitude. This had led to the mapping of signal and image processing algorithms, and consequently their applications, to run entirely on GPUs.

This paper presents CUFFTSHIFT, a ready-to-use GPU-accelerated library, that implements a high performance parallel version of the FFT-shift operation on CUDA-enabled GPUs. Compared to Octave, CUFFTSHIFT can achieve up to 250×, 115×, and 155× speedups for one-, two- and three dimensional single precision data arrays of size 33554432, 8192^2 and 512^3 elements, respectively. The library is designed to be compatible with the CUFFT library, which lacks a native support for GPU-accelerated FFT-shift operations.

1. INTRODUCTION
Frequency domain analysis and Fourier methods are in extensive use in a wide range of applications in various engineering and scientific fields, such as signal and image processing, electronics, financial economics for a long time [1, 2, 3]. According to the nature of the signal, many transform algorithms have been presented to switch signals between the frequency and the temporal/spatial domains. **Fourier transform** (FT) is considered one of the most important tools that has been adopted primarily by the scientific community to handle this transformation. Numerically, it was firstly calculated relying on the Discrete Fourier Transform (DFT) algorithm, but due to its $O(N^2)$ computational complexity, Fast Fourier Transform (FFT) is used as a less time-consuming alternative for computing FTs in a divide-and-conquer fashion with an $O(NLogN)$ complexity [3, 4, 5].

Due to their embarrassingly parallel nature, signal processing algorithms that involve FT operations have been implemented on parallel computing architectures to exhibit dramatic speedups compared to their serial implementations [6, 7]. During the past couple of decades, such algorithms were accelerated on multiple computing units embedded in computer clusters; which consist of several powerful communicating nodes linked together via high speed network adapters and fiber optics connections. However, this solution was relatively expensive and time-consuming to maintain.

With the advent of commodity programmable GPUs, it became much cheaper and power efficient to exploit their powerful underlying architecture to implement pure data-parallel algorithms. In 2007, NVIDIA released CUDA (Compute Unified Device Architecture), a parallel computing platform and programming model that provides flexible access to the parallel computational elements of the GPU [8]. As an integrated part of the CUDA library, NVIDIA ships a CUDA-based FFT library (CUFFT), which provides a GPU-accelerated implementation of the FFT algorithm on the GPU [9]. Compared to other FFT libraries, CUFFT has achieved up to 10× speedup with respect to other serial and multi-threaded implementations of MKL [10] and FFTW [11] libraries.

FFT-shift is a mandatory subsequent procedure to any FFT operation to set the zero-frequency of the resulting spectrum at the origin. Unfortunately, the CUFFT library does not integrate any FFT-shift functionality into it. This issue notably limits the ultimate acceleration of FFT-based applications on the GPU, and this is why some workarounds or custom implementations have to be found [12, 13]. Other solutions use CPU-based implementation, however, this requires a roundtrip and copy from GPU to CPU and back, introducing a significant overhead and consequently a negative influence on the overall performance of the application.

For these reasons, there was a strong motivation to have a parallel implementation of one-, two-, and three-dimensional FFT-shift modules to complement the missing functionality of the CUFFT library. This work extends [14] and presents CUFFTSHIFT - a GPU-based library[1] that provides a simple interface and fast implementation for FFT-shift operations, allowing the complete execution of FFT-based algorithms on the CUDA-enabled GPUs.

[1]The library is available online under the LGPL license, `https://github.com/marwan-abdellah/cufftShift`

1.1. FFT-Shift

For N-dimensional signals the FFT-shift operation wraps around the resulting spectrum from the FFT operation to have its DC (zero-frequency) component centered at the origin. The shift problem itself originates in the definition of the DFT function since most of the implementations of the FFT algorithm use the causal version shown in Equation (1), where $F_N[m]$ is a complex number that encodes the amplitude and phase of a sinusoidal component of the function $f_N[n]$, m and n are the samples in the Fourier and spatial domains respectively, and N is the discrete signal sequence length.

$$F_N[m] = \sum_{n=0}^{N-1} f_N[n] \exp(-i\frac{2\pi m}{N}N) \quad (1)$$

According to this formula, shrink-wrapped FFT implementations generate the origin at the first element of the resulting arrays from the FFT operation. As a result, displaying this spectral array will set the origin at the edge of the array rather than being at the center. However, it is much more intuitive to have the origin set at the center of the array, which requires the FFT operation to be followed by a spectrum-fixing FFT-shift operation.

Most wide-spread standalone FFT libraries such as FFTW [15] and MKL [16] do not provide ready to use FFT-shift functions in their implementation. However, other scientific mathematical packages such as MATLAB (MATHWORKS) and Octave [17] provide the functions `fftshift` and `ifftshift` to perform one-, two- and three-dimensional forward and inverse FFT-shift operation, respectively [18, 19]. Lehmann [20] has designed a multi-threaded shifting filter to operate only on two-dimensional images. This FFTSHIFTIMAGEFILTER has been integrated in the Insight Segmentation and Registration Toolkit (ITK).

1.2. Use Case in Medical Visualization: X-Ray Volume Rendering

Amongst several applications that use FFT-shift operations in their pipeline, X-Ray Volume Rendering[2] has been selected to motivate the presented work. This rendering technique is based on the Projection-Slice theory to generate X-ray like images of medical data sets [21].

Figure 1 depicts an axial projection image of the Foot data set generated by this rendering algorithm, and Figure 2 shows the rendering pipeline. This technique reduces the computational complexity of image generation from $O(N^3)$ associated with spatial domain rendering algorithms to $O(N^2 logN)$ by switching part of its pipeline to operate in the frequency domain. Three FFT-shift operations are performed in this pipeline, once on the real-valued spatial volume, once on the spectral volume and once on the final image resulting from the inverse FFT operation.

[2]This technique is also refereed by Fourier Volume Rendering in other contexts.

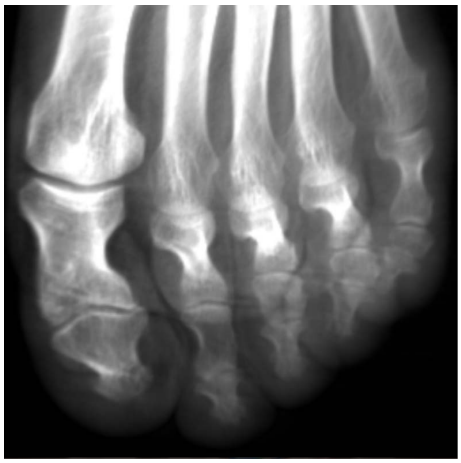

Figure 1. Axial projection image of the Foot data set created using X-ray Volume Rendering.

Traditionally, this pipeline uses a hybrid CPU-GPU implementation, where the different FFT-shift operations are entirely executed on the CPU [21, 22]. For this particular use case, the presence of an accelerated implementation of FFT-shift modules that operate on two- and three-dimensional real and complex data is crucial for leveraging the performance of the entire pipeline on the GPU.

2. ALGORITHMS & IMPLEMENTATION

In this section, the algorithms reflecting the 1D, 2D and 3D FFT-shift operations are presented. The FFT-shift operation is an embarrassingly parallel problem. It rearranges the output array from a preceding FFT operation by moving – or *shifting* – the DC component of the resulting spectrum from the edge to the center of the array. This operation entails swapping the corresponding elements of the array interchangeably depending on structure of the array. For instance, and for a one-dimensional *array* of length N, the FFT-shift operation will swap the $array[0]$ element with the $array[N/2]$ element.

This operation requires decomposing the array into subspaces and formulating the *shifting equations* that govern swapping the corresponding elements in the array and then calculating the right indices to swap these elements correctly. These equations are easily derived for flat arrays containing one-dimensional data. This algorithm is implemented by writing the appropriate *CUDA kernel*, using the one-dimensional data layout used in the CUFFT library for executing one-, two-, and three-dimensional FFTs [9].

The main challenge in this problem is to find the correct CUDA thread index to map multi-dimensional signals onto one-dimensional arrays. Additionally, the limited threading hierarchy of early CUDA-enabled GPU architectures com-

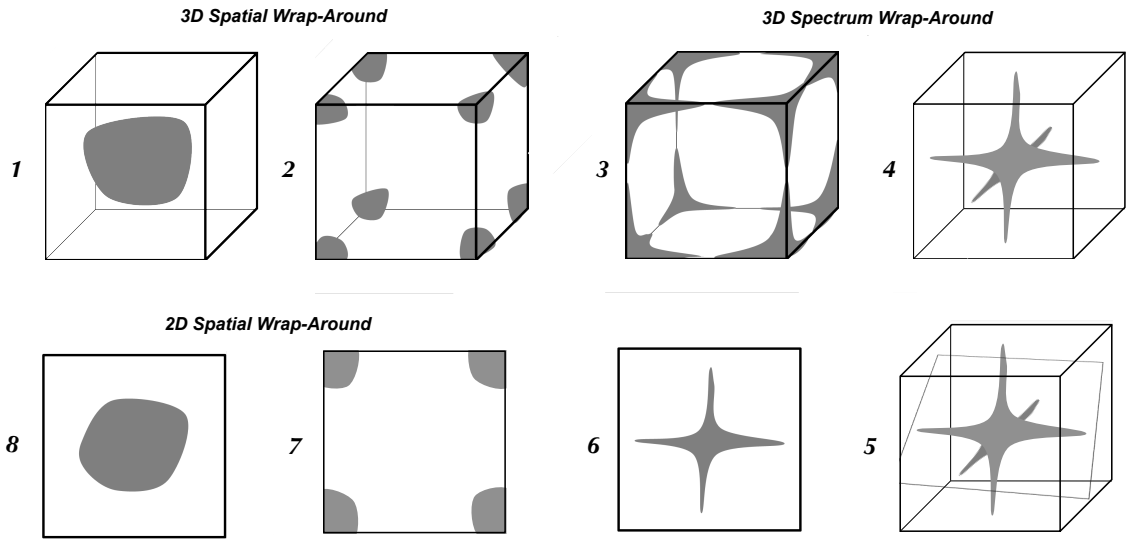

Figure 2. Graphical illustration for the Fourier Volume Rendering algorithm. It requires three FFT-shift operations to generate a correct projection image. Stages 1, 2, 3 and 4 are reprocessing stages preparing the spectral volume for a continuous rendering loop that is executed in stages 5, 6, 7 and 8. Image courtesy of [21].

plicates these calculations further for the three-dimensional case. All the multi-dimensional arrays mentioned in this paper are stored in one-dimensional – or *flat* – arrays containing multi-dimensional data.

The remainder of this section is divided into two parts. Section 2.1. presents the parallel algorithms for the one-, two- and three-dimensional FFT-shift operations, and Section 2.2. highlights the most notable implementation details.

2.1. Algorithms

2.1.1. 1D Data Arrays

For vectors or one-dimensional data arrays, the FFT-shift operation swaps the two halves of the arrays H_1 and H_2 interchangeably, as shown in Figure 3. The vector is assumed to have an even number of entries, otherwise it will be zero-padded before executing the swapping operation. The indexing in this case is trivial, since there is no mapping between the dimensionality of the array and its contents. Algorithm 1 shows the `cufftShift_1D_kernel` that underlies the `cufftShift_1D` function invocation. Note that the *index* calculation here is just required to execute the right swapping across the elements of the vector, but it is not related to any mapping operation. The BLOCKIDX, BLOCKWIDTH and THREADIDX parameters are built-in variables calculated by the CUDA runtime system during kernel execution.

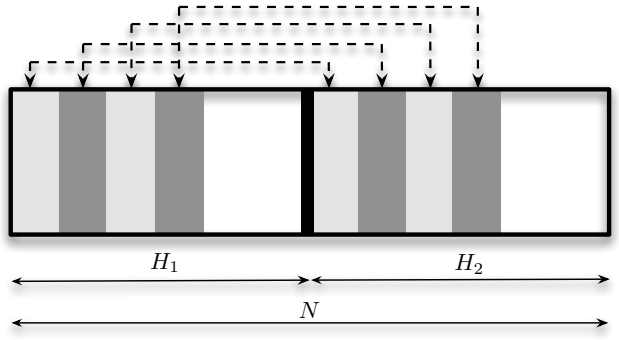

Figure 3. FFT-shift operation for a one-dimensional array. The array is decomposed to 2 halves H_1 and H_2 to have them swapped interchangeably.

Algorithm 1 CUFFTSHIFT_1D_KERNEL($data, N$)

Require: N is even number of data elements
Require: $data$ size = N
Require: $data$ is 1D array
  $index \leftarrow$ BLOCKIDX.x * BLOCKWIDTH + THREADIDX.x
  if $index < N/2$ **then**
    $temp = data[index]$
    $data[index] \leftarrow data[index + (N/2)]$
    $data[index + (N/2)] \leftarrow temp$
  end if

2.1.2. 2D Data Arrays

In general, mapping a two-dimensional image by a flat array, as shown in Figure 4, is achieved using a simple formula that maps the row and column indexes into a one-dimensional substitute. Shifting two-dimensional signals in flat arrays is complicated by the calculation of the shift equations required

to swap the two-dimensional data across the flat array and by using the right thread indices to map the corresponding elements that have to be swapped.

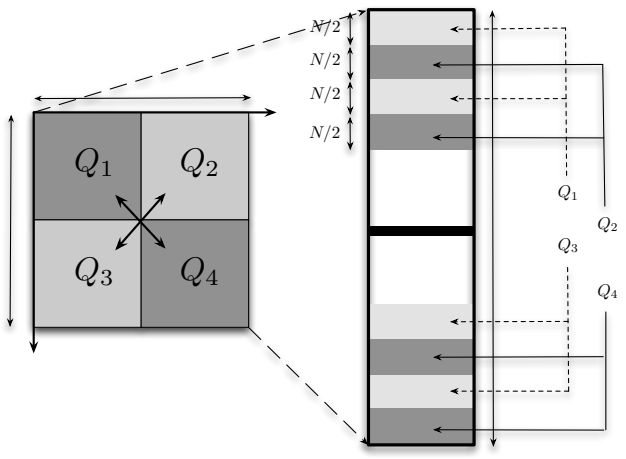

Figure 4. FFT-shift operation for a two-dimensional array stored in a 1D vector. On the left, the array is logically represented by a two-dimensional image. The mapping of the contents of this image into 1D array is shown on the right.

An input image with power-of-two and unified dimensions ($N_x = N_y$) can be decomposed into four quadrants Q_1, Q_2, Q_3 and Q_4. The FFT-shift operation swaps each two opposite quadrants by replacing the contents of Q_1 with Q_4, Q_4 with Q_1, Q_2 with Q_3 and Q_3 with Q_2. The library provides the function `cufftShift_2D` to shift two-dimensional signal packed in flat array. The skeleton of the underlying CUDA kernel `cufftShift_2D_kernel` is shown in Algorithm 2. The algorithm highlights the shift equations eq_1 and eq_2 and the conditions that determine the correct indices of the elements that will be swapped.

Algorithm 2 CUFFTSHIFT_2D_KERNEL($data, N$)

Require: N is a power of two integer
Require: $data$ size = N^2
Require: $data$ is 2D **flat** array
  $eq1 \leftarrow ((N*N)+N)/2$
  $eq2 \leftarrow ((N*N)-N)/2$
  $x \leftarrow$ BLOCKIDX.x * BLOCKWIDTH + THREADIDX.x
  $y \leftarrow$ BLOCKIDX.y * BLOCKHEIGHT + THREADIDX.y
  $index \leftarrow (y*N)+x$
  if $x < N/2$ **then**
    if $y < N/2$ **then**
      $temp = data[index]$
      $data[index] \leftarrow data[index+eq1]$
      $data[index+eq1] \leftarrow temp$
    end if
  else
    if $y > N/2$ **then**
      $temp = data[index]$
      $data[index] \leftarrow data[index+eq2]$
      $data[index+eq2] \leftarrow temp$
    end if
  end if

2.1.3. 3D Data Arrays

Compared to the one- and two-dimensional cases, shifting three-dimensional signals is much more complex. In addition to the mapping of the three-dimensional contents of the data into flat arrays and calculating the shift equations, the implementation is restricted to using two-dimensional grids to support older CUDA devices that have compute capability of 1.x, adding one layer of indirection to the mapping.

Using input images of unified dimensionality ($N_x = N_y = N_z$) and power-of-two size, the three-dimensional array is decomposed into eight equivalent octants $O_1, O_2, ..., O_8$. The `cufftShift_3D` function swaps each two opposite octants: $O_1 \leftrightarrow O_8, O_2 \leftrightarrow O_7, O_3 \leftrightarrow O_6$ and $O_4 \leftrightarrow O_5$. The three-dimensional volume is handled as a stack of two-dimensional slices for compatibility with older CUDA devices. Each slice is processed in a single kernel invocation as shown in Algorithm 3. This kernel has been extended from the previous two-dimensional one to add the slice index check before swapping the slice elements with their corresponding elements in the opposite slice. Note that the *index* here is calculated in terms of x, y, and z, which exactly mimics having a three-dimensional grid in the threading hierarchy. This algorithm is graphically illustrated in Figure 5.

2.2. Implementation

The CUFFTSHIFT library provides a programming interface compatible with the CUFFT library. The interface has been designed to provide template functions with the same data types supported by CUFFT: `cufftReal`, `cufftDoubleReal`, `cufftComplex`, and `cufftDoubleComplex`.

The implementation is fully backward compatible with CUDA-enabled devices of compute capability of 1.1. These GPUs only support two-dimensional kernel grids, complicating the three-dimensional shift implementation as described previously. The functions `cufftShift_2D` and `cufftShift_3D` directly operate on the resulting arrays from the CUFFT plans (data structures that contain all the data required by CUFFT to compute the FFT) without changing the layout of the multi-dimensional data in their respective flat arrays.

The implementation is scalable and the maximum size of the arrays to be shifted is only limited by the available GPU memory. The shift operation is fully performed in place, fully

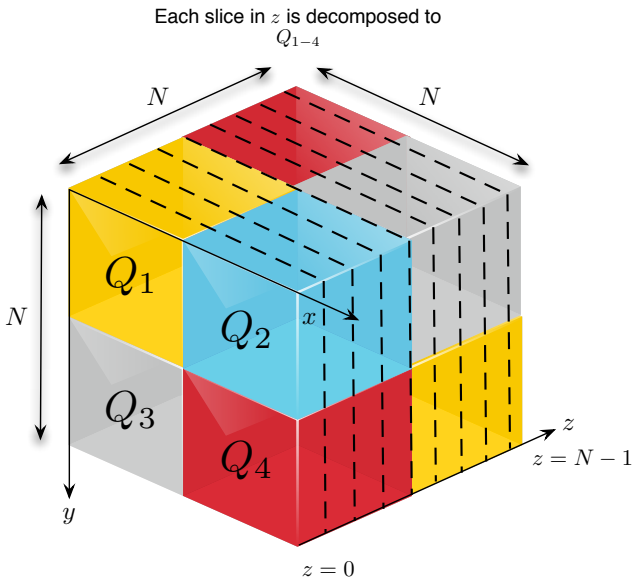

Figure 5. FFT-shift operation for three-dimensional data. The sub-cubes in the same colors are swapped. The volume is decomposed in a stack of N slices of size N^2 in the Z-direction, processing each slice in a single kernel invocation.

optimizing GPU memory usage. The implementation will automatically optimize the kernel block and grid configuration depending on the GPU architecture, input array size and dimensionality.

The implementation is limited to operate only on 1D arrays of even sizes and 2D/3D arrays with power-of-two sizes and unified dimensionality. This constraint can be easily resolved by zero-padding the input arrays, which is a common approach in signal processing communities.

3. RESULTS & DISCUSSION

The benchmarking was done on a workstation with two Intel Xeon X5690 CPU running at 3.47 GHz, 24 GB of DDR3 memory and an NVIDIA GeForce GTX 580 GPU. The GPU has CUDA device capability 2.0 and contains 3 GB of DDR5 memory with 384-bit interface and 192.4 GB/s bandwidth. It also contains 512 CUDA cores running at 1.54 GHz.

Figures 6(a), 6(c), and 6(e) illustrate the profiling benchmarks for executing the `cufftShift_1D`, `cufftShift_2D`, and `cufftShift_3D` functions on one-, two- and three-dimensional flat arrays of different sizes and the data types supported by the CUFFT library. The benchmarks represent the average number of data transfer operations per second in (MOps/sec). The automatic kernel configuration done upon each function invocation is included in the measurements. The upper limits of the data sizes benchmarked were limited by the memory constraints of our GPU. However, the library

Algorithm 3 CUFFTSHIFT_3D_KERNEL($data, N$)

Require: N is a power of two integer
Require: $data$ size = N^3
Require: $data$ is 3D **flat** array
  for $z \leftarrow 0$ to $(N-1)$ **do**
    $eq1 \leftarrow ((N*N*N)+(N*N)+N)/2$
    $eq2 \leftarrow ((N*N*N)+(N*N)-N)/2$
    $eq3 \leftarrow ((N*N*N)-(N*N)+N)/2$
    $eq4 \leftarrow ((N*N*N)-(N*N)-N)/2$
    $x \leftarrow$ BLOCKIDX.$x*$BLOCKWIDTH + THREADIDX.x
    $y \leftarrow$ BLOCKIDX.$y*$BLOCKHEIGHT + THREADIDX.y
    $index \leftarrow (z*(N*N))+(y*N)+x$
    if $z < N/2$ **then**
      if $x < N/2$ **then**
        if $y < N/2$ **then**
          $temp \leftarrow data[index]$
          $data[index] \leftarrow input[index+eq1]$
          $data[index+eq1] \leftarrow temp$
        else
          $temp \leftarrow data[index]$
          $data[index] \leftarrow input[index+eq3]$
          $data[index+eq3] \leftarrow temp$
        end if
      else
        if $y < N/2$ **then**
          $temp \leftarrow data[index]$
          $data[index] \leftarrow input[index+eq2]$
          $data[index+eq2] \leftarrow temp$
        else
          $temp \leftarrow data[index]$
          $data[index] \leftarrow input[index+eq4]$
          $data[index+eq4] \leftarrow temp$
        end if
      end if
    end if
  end for

is scalable and can work on larger arrays as long as they can fit into the GPU global memory.

As shown in the benchmarks, the performance of the shifting operations for the different data arrays with `cufftDoubleReal` and `cufftComplex` data types is almost the same since they have the same memory size, just with different layouts. The variation of the performance for the different data types becomes more apparent for larger arrays (containing more than 2^{20} data elements). Also, and as expected, the performance of the three-dimensional case is clear to be lower than the two-dimensional case due to the serial execution of multiple kernels for processing the volume stack.

Figures 6(b), 6(d), and 6(f) compare our implementation on single precision data against the Octave `fftshift` functions for one-, two, and three-dimensional data arrays respectively. For the three cases, the performance of our shifting functions for small arrays is already multiple times faster, and for large data arrays we outperform Octave by more than two orders of magnitude.

4. REMARKS AND CONCLUSION

This work presents CUFFTSHIFT, a CUDA-based GPU-accelerated library that implements one-, two- and three-dimensional FFT-shift operations. The library affords a flexible programming interface that is also compatible with the commonly used CUFFT library. During the run-time, it generates optimized kernel configurations based on the running GPU, input array size and the dimensionality of the FFT-shift operation. These features make the library an easy to use, easily integrated alternative for accelerating FFT-based applications on CUDA-enabled GPUs. As a reference, the presented implementation is compared to the `fftshift` module in Octave, outperforming it by up to 250×, 115× and 155× for one-, two-, and three-dimensional cases, respectively. Most importantly, it enables fully GPU-accelerated FFT processing pipelines by simplifying their implementation while providing better performance.

5. FUTURE WORK

The current implementation is planned to be further optimized by improving the kernel structure to reduce the warp divergence resulting from the branching required to index the correct elements. Shared memory will be considered as a faster alternative to the global memory. Future versions of this library will extend the implementation to support shifting non-power-of-two arrays, which is a constraint of the current implementation.

ACKNOWLEDGMENT

The author would like to thank Dr. Ahmet Bilgili and Stefan Eilemann (Blue Brain Project, EPFL) for their valuable comments and effort to put this work in this shape.

REFERENCES

[1] Vijay K. Madisetti and Douglas B. Williams. *Digital Signal Processing Handbook on CD-ROM*. CRC Press, 1999.

[2] David Brandwood. *Fourier Transforms in Radar and Signal Processing (Artech House Radar Library)*. Artech House Publishers, 2003.

[3] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] John L. Semmlow. *Biosignal and Biomedical Image Processing*, chapter 3. Marcel Dekker, 270 Madison Avenue, New York, NY 10016, U.S.A., 2004.

[5] Steven T. Karris. *Signals and Systems*, chapter 10. Orchard, Fremont, California, 2003.

[6] Zhiyi Yang, Yating Zhu, and Yong Pu. Parallel Image Processing Based on CUDA. In *International Conference on Computer Science and Software Engineering, 2008*, volume 3, pages 198–201, December 2008.

[7] Thilaka Sumanaweera and Donald Liu. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[8] NVIDIA. NVIDIA CUDA C Programming Guide, April 2012.

[9] NVIDIA. CUFFT Library, CUDA Toolkit 5, October 2012.

[10] NVIDIA. CUFFT. `https://developer.nvidia.com/cufft`, 2013. Figure (FFTS UP TO 10X FASTER THAN MKL).

[11] Hugh Merz. CUFFT vs FFTW comparison. `http://www.sharcnet.ca/~merz/CUDA_benchFFT/`.

[12] A. Eklund, M. Warntjes, M. Andersson, and H. Knutsson. Phase Based Volume Registration on the GPU with Application to Quantitative MRI. In *Proceedings of the SSBA Symposium on Image Analysis*, Uppsala, Sweden, March 2010.

[13] Anders Eklund, Mats Andersson, and Hans Knutsson. True 4D Image Denoising on the GPU. *International Journal of Biomedical Imaging*, 11, 2011.

[14] M. Abdellah, S. Saleh, A. Eldeib, and A. Shaarawi. High performance multi-dimensional (2d/3d) fft-shift implementation on graphics processing units (gpus). In *Biomedical Engineering Conference (CIBEC), 2012 Cairo International*, pages 171–174, Dec 2012.

[15] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

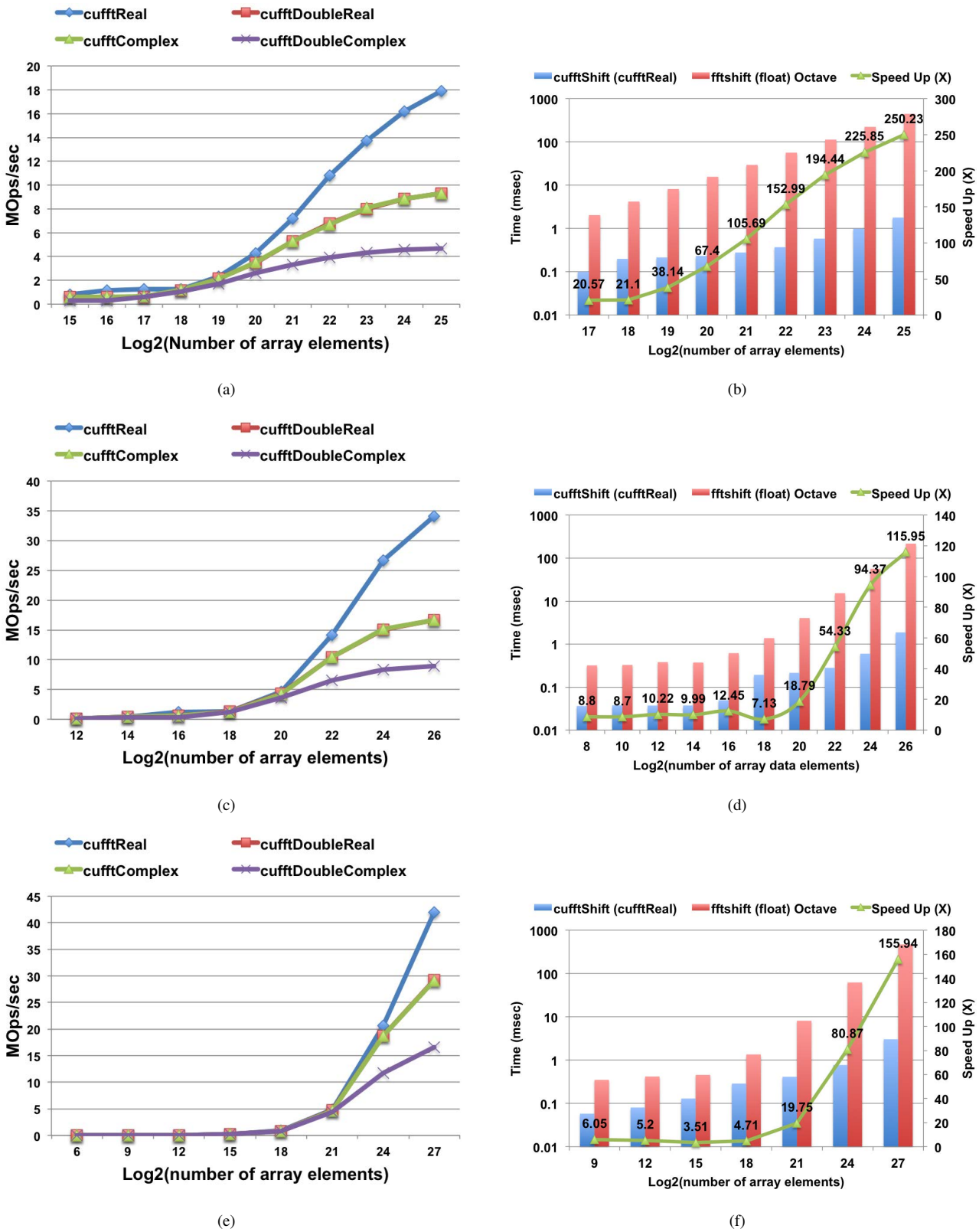

Figure 6. Benchmarking results. The processing speed in operations per second is shown for the 1D, 2D, and 3D shift operations in (a), (c), and (e) respectively. On the right-hand side, we compare the performance between our implementation and the Octave for 1D (b), 2D (d), and 3D (f) data.

[16] Intel. Intel Math Kernel Library (Intel MKL) 11.0. http://software.intel.com/en-us/intel-mkl.

[17] John W Eaton, David Bateman, and Soren Hauberg. *GNU Octave Manual Version 3*. Network Theory Ltd., 2008.

[18] MATHWORKS. ffftshift R2012a Documentation Page. http://www.mathworks.ch/help/techdoc/ref/fftshift.html, March 2012.

[19] Julius O. Smith. *Spectral Audio Signal Processing*. W3K Publishing, 2011.

[20] Gaetan Lehmann. FFT Shift. *The Insight Journal (July - December)*, 2006.

[21] Marwan Abdellah, Ayman Eldeib, and Amr Shaarawi. Constructing a Functional Fourier Volume Rendering Pipeline on Heterogeneous Platforms. In *Proceedings of the 6th Cairo International Biomedical Engineering Conference*, 2012.

[22] Ivan Viola, Armin Kanitsar, and Meister Eduard Gröller. GPU-based Frequency Domain Volume Rendering. In *Proc. of the 20th spring conf. on Computer graphics*, pages 55–64, 2004.