





SDP Memo: Fast Fourier Transforms

Document number.....SKA-TEL-SDP-0000058
Document Type.....REP
Revision.....02C
Author..... Stefano Salvini
Release Date.....2015-05-29
Document Classification..... Unrestricted
Status..... Draft

Lead Author	Designation	Affiliation
Stefano Salvini		The University of Oxford
Signature & Date:	 Stefano Salvini (Mar 31, 2016) stef.salvini@oerc.ox.ac.uk	
Released by	Designation	Affiliation
Bojan Nikolic	SDP Project Engineer	University of Cambridge
Signature & Date:	 Bojan Nikolic (Mar 31, 2016) b.nikolic@mrao.cam.ac.uk	

Version	Date of Issue	Prepared by	Comments
1	2015-02-09	Stefano Salvini	PDR submission
1A	2015-05-29	Stefano Salvini	Updated with PDR panel comments
02C	2016-03-31	Ian Cooper / Ferdl Graser	Copy-edited for dPDR submission. No content changes

ORGANISATION DETAILS

Name	Science Data Processor Consortium
------	-----------------------------------

Table of Contents

List of Tables	4
1. Introduction	5
2. Computational Costs and Complexity	6
2.1 Number of Operations.....	6
2.2 Data Sizes	7
2.3 Operations Density and Data Movement.....	7
2.4 Observed Efficiency (Performance Relative to Peak).....	7
2.5 Partial FFT	8
3. System and Software Used.....	9
3.1 CPU Multithreading and FFTW	11
4. Model Problems	12
5. Numerical Performance	13
6. Computational Performance	17
7. Conclusions	20

List of Tables

Table 1	2D FFT approximate performance (% of peak).....	8
Table 2	Hardware used in the benchmarks	9
Table 3	Errors for random matrix (case 1).....	13
Table 4	Errors for zero matrix with N sources of unit strength (Case 2).....	14
Table 5	Errors for zero matrix with 1 source of strength 100,000 (Case 3).....	15
Table 6	Execution times in seconds for model problem 1.....	17
Table 7	Performance figures in Gflops/second for model problem 1.	18

1. Introduction

The Fast Fourier Transform (FFT) is one of the basic numerical components for SKA SDP computational pipelines.

The following issues have been addressed here:

1. Present a computational cost and performance model. This would need to include not just the operation counts but also costs incurred in data transfer from outside as well as inside computational engines. This needs to be supported by appropriate benchmarks.
2. If visibilities are provided in single precision, should we use single or double precision to compute FFTs? What is the difference in terms of numerical accuracy, if any, in different realistic regimes?
3. Are FFTs on different platforms and from different packages or libraries equivalent and interchangeable in terms of their numerical properties?

In a nutshell, the answers to these questions can be summarised as follows:

1. A simple computational model in terms of operation counts can be proposed. However, the overall computational costs depend strongly on the details of the FFT function and only some figure of merit with respect to peak performance can be given.
2. In the case of single precision visibilities, the difference between computing the FFT in single and double precision is between half and one decimal digits (roughly speaking $\mathcal{O}(10^{-7})$ against $\mathcal{O}(10^{-8})$ normwise error in the test cases and problem sizes examined). This is unlikely to have any major impact except in very specific circumstances, for example if multiple FFTs were to be accumulated in double precision (but then, perhaps, they could be computed in single then accumulated in double precision?).
3. All the platforms and libraries studied show numerical behavior consistent with each other. To all effects, they can be considered fully interchangeable.

Additionally, some comments are included here for the cases in which a full FFT is not required.

1. Only a small portion of the image is required (should we use a DFT, thus bypassing the gridding?)
2. If the sky is mostly empty or in the presence of few very bright sources, Sparse FFT could be used to lessen the computational costs and limit the image to those regions with largest flux.

This report is structured in sections as follows. The second section will describe the computational system and the software used. The third section will describe the model problems used and the criteria for assessment of numerical accuracy and performance. This is followed by two sections: the first reports on the numerical, the second on the computational performance, respectively. The final section will outline the conclusions drawn.

Obviously, if double precision visibility matrices were provided, FFTs would need to be computed in double precision. Errors and performance figures, in these cases, would be the same as those reported for double precision in sections 4 and 5 below.

2. Computational Costs and Complexity

2.1 Number of Operations

The number of floating-point operations required by an FFT has been long debated. It actually depends on a number of factors:

- The prime factors of N , the FFT length
- The algorithm used and how the software is implemented
- The use of composite radices: for example explicit radix-4 components may be used rather than iterating 2 radix-2 butterflies, for reason of efficiency. Although this could increase the number of operations, it could also increase data locality and software pipelining thus leading to better efficiency.

The number of operations for a one-dimensional FFT when N is a power of 2 is often given as

$$N_{ops} \sim 5 N \log_2 N$$

It should be noticed that the situation for prime factors different from 2 is particularly unclear.

A 2-D FFT is equivalent to N FFTs along the columns (or rows) of the data points matrix followed by N FFTs along the rows (or columns) for a grand total of $10 N^2 \log_2 N$ flops. As the matrix of the visibilities is Hermitian, we are interested in Hermitian-to-real 2D FFTs, thus halving the number of operations for a total cost of

$$N_{ops} \sim 5 N^2 \log_2 N$$

It is clear that the number of operations can only be seen as “notional”, in the sense that it allows a tabulation and prediction of performance and some estimates of efficiency. The operation count that we use here results in computational efficiency of between 8 to 15%. It must be stressed that the efficiency is related to the operation count: changing the operation count by some factor, would affect the efficiency by that same factor. For example, doubling the operation count would increase efficiency by a factor 2: thus, the actual computation time would be obviously unchanged.

The actual operation count was measured approximately for the MKL Library tests using Intel hardware counters: this indicates that the number of operations is always larger than the value of N_{ops} we used, but that the difference was modest and rather consistent across different problem sizes. Hence the value of N_{ops} we used should be viewed as a reasonable lower bound on the actual number of operations.

Also FFTW can return an estimate of the number of operations: this, unfortunately, proved to be an unreliable serious underestimate and could not be used.

2.2 Data Sizes

The amount of input as well as output data is: $\mathcal{O}(N^2)$. Temporary storage is required, but its size depends on the algorithm implemented, etc., but is likely to be $\mathcal{O}(KN)$, where $K \leq N$.

2.3 Operations Density and Data Movement

It is now clear that the achievable performance is limited by data-access and very low data reuse as at best the number of operations for each item of data is $\mathcal{O}(\log N)$.

Each FFT sweep, whether for a prime or composite radix, in general would require refreshing the data in the active portions of memory (caches, shared memories, etc.) except for the smallest sizes. So, if only radix-2 were used, we would require $\log_2 N$ data transfers from memories to caches (or shared memory, etc) if radix-2 were bunched into composite radix-4, that would become $\log_4 N$, thus halving the memory transfer required.

The computational structure may use smaller radices than the data transfer structure (for example, using radix-6 but applying the butterfly as a radix-2 followed by a radix-3). Different platforms would require different, and flexible strategies, hence FFT libraries generally formulate a *plan* before carrying out the computation. For example, some libraries may “bunch” radices into very sizable composites (say up to 256 or 512) from the point of view of data transfer, while still applying individual smaller butterflies in sequence. Naturally, such code is very complex and is developed for specific platforms.

Without great details of the algorithms used (not available for MKL and CUFFT) a data model cannot be formed explicitly. However

2.4 Observed Efficiency (Performance Relative to Peak)

As a precise performance model is very difficult to formulate, given the number of operations required and the target platform, we can only give some range for a *figure of merit* which allows us to guess a possible range of expected performance **using the operation count we have defined above.**

This is summarised in table 1 below, based on the benchmarks carried out, where *efficiency* denotes the percentage of peak performance achieved by the computation.

Obviously, the relative low capacity of PCI-Express causes the very low overall efficiency of GPUs when data need to be transferred for each FFT from host. However, we would expect that a number of operations would be carried out on visibility data thus data transfer costs could be neglected for the FFTs.

Single Precision	
CPU Efficiency (multithreaded)	8 – 15 %
GPU efficiency (data on GPU)	<10 – 15 %
GPU efficiency (incl. data transfer)	~ 1%
Double Precision	
CPU Efficiency (multithreaded)	8 – 15 %
GPU efficiency (data on GPU)	10 – 15 %
GPU efficiency (incl. data transfer)	~ 1%

Table 1 2D FFT approximate performance (% of peak), using $N_{ops} = 5 N^2 \log_2 N$

2.5 Partial FFT

For a very sparse image (few sources in a mostly empty sky) the novel sFFT (Sparse FFT) algorithms (<http://groups.csail.mit.edu/netmit/sFFT/> and references therein) could be used. These have shown very considerable speed-ups with respect to standard FFT and is being currently tested by A. Scaife and her team for potential use in the Slow Transients Pipeline and the proposed RM Synthesis functionality for ECP140011.

If only a small portion of the image were required, then DFT could be directly used. However, despite the better data access characteristics, possible gains, if any, should be assessed on an individual case basis.

3. System and Software Used

FFT algorithms are exceptionally important and notoriously difficult to implement in a consistent and efficient fashion. Hardware vendors, such as Intel and NVidia, probably spend larger amounts in developing efficient FFTs than on any other numerical software.

Because of the very low computational density per item of data loaded, typically $\mathcal{O}(\log N)$, tight coordination between hardware and code is required to achieve good performance. In any case, performance tends to be much lower than peak, because of the low data re-use.

Throughout, we have used the SKA testbed codenamed ska4, a high-end two-socket platform, at the University of Oxford. This comprises of the components listed in the following table:

Hardware	
	2U chassis with x16 2.5 inch drive bays
CPUs (2)	Dual socket Intel E5-2690 CPUs, @2.9GHz (3.8GHz turbo), 135W TDP, 20MB Cache, 8GT/s QPI, Quad memory channel (Max. 51.2GB/s)
Motherboard	X9DRW-3LN4F+ Supermicro motherboard, BIOS version 3.00
Memory	64GiB (8GiB x 8) ECC DDR3 1600MHz CL11 Single rank RAM @1.5V (model: M393B1G70BH0-CK0)
	Adaptec 71605 16 internal ports RAID card, PCIe 3.0 (x8), Mini-HD SAS, with 1GB DDR3 Cache
	Sixteen 2.5 inch 128GB SATA3/6Gb OCZ Vector MLC NAND SSD's, ~2TB per node (RAID 0).
	Mellanox FDR Infiniband 112Gbit/s (dual ports@56Gbit/s) Connect-IB PCIe 3.0 (x16) (model: MCB194A-FCAT)
	Quad port Intel i350 GbE
PCI-Express Bandwidth	Max: 16 GB/sec Max measured (separate benchmark): < 10 GB/sec
GPU (1)	NVIDIA K40 (PCIe 3.0) (12 GBytes)
	Max Memory Bandwidth: 250 GB/sec

Table 2 Hardware used in the benchmarks

The platform ran under Linux Gentoo Base System release 2.2 with Kernel Version: 3.14.3-ck. All code was compiled using the Intel C and Fortran compilers version XE 14.0 Update 2 to guarantee best performance of all codes (surprisingly, Intel MKL runs slower under GNU compilers).

We have used the following libraries:

1. Intel MKL Library. Version 11.1, Update 2.
Over the years, Intel has developed and maintained FFT functionality in their Math Kernel Library (Intel MKL). The MKL Library provides a good range of functionality and best performance for multi-core platforms (multi-core CPUs).
2. NVidia CUFFT Library Version 5.5.
NVidia provides the CUFFT Library, which is very much comparable in coverage to the Intel MKL and FFTW Libraries (see below), specifically targeted at GPUs. The available functions appear exceptionally fast for data already residing on a GPU. However, the relatively low PCI-Express bandwidth increases computational cost very considerably when data loading into the GPU are included. The newer version of CUFFT is only marginally faster.
3. FFTW version 3.3.
FFTW is an exceptional, very high quality package, created at MIT and distributed as open source. It is the *de facto* standard package for FFTs. This was compiled specifically for the computational platform using the Intel compiler.

The three libraries use slightly different algorithms for the FFT. In all cases, we have used out-of-place variants (likely to be more efficient). FFT plans (hence computation sequence) vary across the libraries, with obvious impact on performance but remarkably little on numerical properties, which in all cases follow what was expected (and hoped for). In particular, they have different approaches for non-trivial prime factors. While FFTW and I believe MKL employ the Rader algorithm, CUFFT employs the Bluestein algorithm. Both use convolution to carry out computations for non-trivial prime factors, but differ in detail and range of applicability. For example, Rader algorithm does not cater for radix 5 (used in the tests): the effects are noticeable more in FFTW than in MKL (where, obviously, special hardware-targeted algorithmic components have been developed).

Given the remit of this investigation, only 2-d complex-to-real FFTs have been reported here. It should be noticed, however, that the same conclusions are reached for other types of FFTs, as tests have shown (not reported here).

FFT functions available from these libraries are used in a sequence of steps:

1. Create/allocate appropriate data structures
2. Generate an appropriate FFT plan (this varies from Library to Library). This step can be expensive. However, as a plan can be reused for different data and a good plan can make increase the performance of the computational step significantly, plan generation costs are not included here and will not be considered any further. In particular, best plans were computed for FFTW.
3. Carry out the computation
4. Clean up

All benchmarks reported used *elapsed time*, that is, the actual time between issuing a computational request (using a function, calling a routine) and exit from this (user-space time). All times and performance figures are given by computing a *single* 2D FFT.

To ensure as much as possible uniform benchmarking conditions, a single main program was written, calling Intel MKL, FFTW and CUFFT functions. Results were validated individually and also against functions from a different library.

Code is, naturally, available on request. Code was designed so that alternative model problems and error analysis can be easily implemented, including using real data, such as from LOFAR, but still using the three FFT libraries.

3.1 CPU Multithreading and FFTW

It should be noticed that a multithreaded variant of FFTW was not used in these benchmarks.

There are a number of reasons for that: first and foremost, the good performance and scaling with the number of threads of the Intel MKL Library. However, as interest in FFTW is very great, a direct comparison, for a single thread, has been carried out.

CPU Multithreaded FFTs are a rather tricky issue, given the low density of operation with respect to data. Multithreaded FFTs are even more so, as multiple threads need to coordinate access through shared memory structures (in itself a daunting problem). Vendors, such as Intel, AMD and Nvidia invest considerable resources in optimizing FFTs, which are seen as essential to their software portfolio.

Running in parallel multiple FFTs, say one per thread, is problematic above a certain problem size, as considerable memory access clashes may occur. Intel MKL also would allow the use of a two-level parallelism, using OpenMP to parallelise across FFTs, MKL threads within each FFT, for potentially best performance across all problem sizes (not studied in this report). This avenue is not open to FFTW which uses OpenMP.

4. Model Problems

In order to create a model problem and assess the quality of the results, the following procedure was devised:

1. Generate an $N \times N$ real matrix A using double precision (the “model sky”, so to speak)
2. Carry out the forward real-to-complex 2-D FFT $B = \mathcal{F}_2(A)$ using double precision (“interferometry”)
3. Round the complex Fourier matrix down to single precision $\hat{B} \leftarrow \text{round}(B)$ (“measurement”)
4. Carry out the backward complex-to-real 2-D FFT $C = \mathcal{F}_2^{-1}(\hat{B})$ using either single or double double precision (“imaging”).

Three different types of matrices A were used

1. Random matrices with entries uniformly distributed between 0 and 1. These give good indications of the intrinsic quality of the algorithm and allow a quantitative comparison between different algorithms.
2. Zero matrices with N non-zero entries (the “sources”), all equal to 1, at random positions in A .
3. Zero matrices with 1 non-zero entry (the “strong source”) set arbitrarily to 10^5 .

For random matrices (model problem 1) error must be normalized over the largest component or the norm of the original data. In general, the error in 2-D FFT should be bound, normwise by a function such as

$$\|C - A\|_F \leq \epsilon f(N) \|A\|_F$$

where ϵ is the machine accuracy, $f(N)$ is a “slowly varying function of the problem size” and $\|\dots\|_F$ denotes the Frobenius norm (ideally the 2-norm, which is more costly to compute). Indeed, for 2-D FFT, $f(N) \sim 1$ as indeed the results in Section 4 show.

For zero matrices with a few non-zero entries (model problem 2 and 3), the analysis of the error (particularly important to understand whether single precision is adequate or double precision would be required), consisted of these steps:

1. Set to zero the entries in C corresponding to the non-zeros in A
2. Compute the error as $\|C\|_F$

Here, the idea is to provide some indication as how different precision would affect the capability of processing weaker structures in the sky in the presence of stronger sources.

Given the structure of the code, alternative model problems can be very easily implemented, including real data, such as data from LOFAR, etc.

5. Numerical Performance

All numerical results obtained were in line with expectations. Only complex-to-real FFTs were considered (see also comments above).

Errors in Table 3 were computed as

$$\epsilon = \frac{\|A - C_x\|_F}{\|A\|_F}$$

where the subscript x denotes either single or double precision; the final part of the table shows the difference between the various methods

$$\Delta_{j,k} = \frac{\|C_j - C_k\|_F}{\|C_j\|_F}$$

Precision		N	500	1000	2000	4000	8000	10000	15000
Single	FFTW		1.21E-07	1.26E-07	1.36E-07	1.36E-07	1.40E-07	1.53E-07	1.54E-07
	MKL	1	1.19E-07	1.32E-07	1.27E-07	1.33E-07	1.38E-07	1.45E-07	1.50E-07
		2	1.19E-07	1.26E-07	1.27E-07	1.33E-07	1.38E-07	1.45E-07	1.50E-07
		4	1.19E-07	1.26E-07	1.27E-07	1.33E-07	1.38E-07	1.45E-07	1.50E-07
		8	1.19E-07	1.26E-07	1.27E-07	1.33E-07	1.38E-07	1.45E-07	1.50E-07
		12	1.19E-07	1.26E-07	1.27E-07	1.33E-07	1.38E-07	1.45E-07	1.50E-07
CUFFT		1.30E-07	1.42E-07	1.47E-07	1.47E-07	1.59E-07	1.82E-07	1.91E-07	
Double	FFTW		1.27E-08	2.17E-08	1.42E-08	1.72E-08	2.14E-08	1.33E-08	2.20E-08
	MKL	1	1.27E-08	2.17E-08	1.42E-08	1.72E-08	2.14E-08	1.33E-08	2.20E-08
		2	1.27E-08	2.17E-08	1.42E-08	1.72E-08	2.14E-08	1.33E-08	2.20E-08
		4	1.27E-08	2.17E-08	1.42E-08	1.72E-08	2.14E-08	1.33E-08	2.20E-08
		8	1.27E-08	2.17E-08	1.42E-08	1.72E-08	2.14E-08	1.33E-08	2.20E-08
		12	1.27E-08	2.17E-08	1.42E-08	1.72E-08	2.14E-08	1.33E-08	2.20E-08
CUFFT		1.27E-08	2.17E-08	1.42E-08	1.72E-08	2.14E-08	1.33E-08	2.20E-08	
Single	MKL-FFTW		1.51E-07	1.55E-07	1.66E-07	1.73E-07	1.76E-07	1.94E-07	1.96E-07
	MKL-CUFFT		1.64E-07	1.76E-07	1.83E-07	1.87E-07	1.99E-07	2.19E-07	2.31E-07
	FFTW-CUFFT		1.67E-07	1.81E-07	1.92E-07	1.92E-07	2.03E-07	2.30E-07	2.31E-07
Double	MKL-FFTW		3.07E-16	3.17E-16	3.26E-16	9.34E-12	3.55E-12	3.01E-12	3.71E-12
	MKL-CUFFT		3.67E-16	3.68E-16	3.85E-16	9.34E-12	3.60E-12	4.24E-12	5.08E-12
	FFTW-CUFFT		3.55E-16	3.61E-16	3.80E-16	7.36E-14	5.85E-13	3.03E-12	4.16E-12

Table 3 Errors for random matrix (case 1)

Errors in Tables 4 and 5 shows different errors, computed as

$$\epsilon = \|C_x\|_F$$

where the subscript x denotes either single or double precision; the final part of the table shows the difference between the various methods

$$\Delta_{j,k} = \|\bar{C}_j - \bar{C}_k\|_F$$

where the bar denotes matrices with the non-zero elements of the initial matrix A set to zero (corresponding to perfect removal of sources). Of course, errors computed as in Table 1 and not reported here, show the same behaviour as in Table 1.

Precision		N	500	1000	2000	4000	8000	10000	15000
Single	FFTW		3.41E-06	5.16E-06	7.99E-06	1.15E-05	1.69E-05	2.00E-05	2.57E-05
	MKL	1	3.66E-06	5.42E-06	7.92E-06	1.17E-05	1.73E-05	2.03E-05	2.59E-05
		2	3.66E-06	5.42E-06	7.92E-06	1.17E-05	1.73E-05	2.03E-05	2.59E-05
		4	3.66E-06	5.42E-06	7.92E-06	1.17E-05	1.73E-05	2.03E-05	2.59E-05
		8	3.66E-06	5.42E-06	7.92E-06	1.17E-05	1.73E-05	2.03E-05	2.59E-05
		12	3.66E-06	5.42E-06	7.92E-06	1.17E-05	1.73E-05	2.03E-05	2.59E-05
CUFFT		3.87E-06	6.43E-06	9.53E-06	1.34E-05	2.04E-05	2.69E-05	2.59E-05	
Double	FFTW		5.65E-07	7.99E-07	1.13E-06	1.60E-06	2.27E-06	2.54E-06	3.10E-06
	MKL	1	5.65E-07	7.99E-07	1.13E-06	1.60E-06	2.27E-06	2.54E-06	3.10E-06
		2	5.65E-07	7.99E-07	1.13E-06	1.60E-06	2.27E-06	2.54E-06	3.10E-06
		4	5.65E-07	7.99E-07	1.13E-06	1.60E-06	2.27E-06	2.54E-06	3.10E-06
		8	5.65E-07	7.99E-07	1.13E-06	1.60E-06	2.27E-06	2.54E-06	3.10E-06
		12	5.65E-07	7.99E-07	1.13E-06	1.60E-06	2.27E-06	2.54E-06	3.10E-06
CUFFT		5.65E-07	7.99E-07	1.13E-06	1.60E-06	2.27E-06	2.54E-06	3.10E-06	
Single	MKL-FFTW		4.60E-06	6.76E-06	1.04E-05	1.54E-05	2.29E-05	2.70E-05	3.46E-05
	MKL-CUFFT		5.15E-06	8.20E-06	1.21E-05	1.73E-05	2.64E-05	3.31E-05	4.46E-05
	FFTW-CUFFT		5.04E-06	8.07E-06	1.22E-05	1.74E-05	2.63E-05	3.31E-05	4.43E-05
Double	MKL-FFTW		2.29E-14	1.47E-14	2.19E-14	3.25E-14	6.74E-10	5.39E-10	5.21E-10
	MKL-CUFFT		2.43E-14	1.89E-14	1.69E-10	4.49E-14	7.43E-14	5.39E-10	4.13E-10
	FFTW-CUFFT		1.20E-14	1.77E-14	1.69E-10	4.32E-14	6.74E-10	1.06E-12	4.19E-10

Table 4 Errors for zero matrix with N sources of unit strength (Case 2)

Precision		N	500	1000	2000	4000	8000	10000	15000
Single	FFTW		6.25E-03	7.70E-03	7.70E-03	9.05E-03	8.95E-03	1.05E-02	1.37E-02
	MKL	1	1.01E-02	1.12E-02	1.12E-02	1.19E-02	1.37E-02	1.22E-02	1.38E-02
		2	1.01E-02	1.12E-02	1.12E-02	1.19E-02	1.37E-02	1.22E-02	1.38E-02
		4	1.01E-02	1.12E-02	1.12E-02	1.19E-02	1.37E-02	1.22E-02	1.38E-02
		8	1.01E-02	1.12E-02	1.12E-02	1.19E-02	1.37E-02	1.22E-02	1.38E-02
		12	1.01E-02	1.12E-02	1.12E-02	1.19E-02	1.37E-02	1.22E-02	1.38E-02
CUFFT		1.10E-02	7.38E-03	9.27E-03	1.50E-02	1.50E-02	1.69E-02	1.71E-02	
Double	FFTW		2.53E-03	2.91E-03	2.91E-03	2.50E-03	2.50E-03	2.46E-03	2.47E-03
	MKL	1	2.53E-03	2.91E-03	2.91E-03	2.50E-03	2.50E-03	2.46E-03	2.47E-03
		2	2.53E-03	2.91E-03	2.91E-03	2.50E-03	2.50E-03	2.46E-03	2.47E-03
		4	2.53E-03	2.91E-03	2.91E-03	2.50E-03	2.50E-03	2.46E-03	2.47E-03
		8	2.53E-03	2.91E-03	2.91E-03	2.50E-03	2.50E-03	2.46E-03	2.47E-03
		12	2.53E-03	2.91E-03	2.91E-03	2.50E-03	2.50E-03	2.46E-03	2.47E-03
CUFFT		2.53E-03	2.91E-03	2.91E-03	2.50E-03	2.50E-03	2.46E-03	2.47E-03	
Single	MKL-FFTW		1.02E-02	1.18E-02	1.18E-02	1.43E-02	1.54E-02	1.53E-02	1.89E-02
	MKL-CUFFT		1.56E-02	1.38E-02	1.48E-02	1.96E-02	1.97E-02	2.09E-02	2.14E-02
	FFTW-CUFFT		1.23E-02	1.08E-02	1.22E-02	1.70E-02	1.68E-02	1.93E-02	2.20E-02
Double	MKL-FFTW		3.02E-11	3.31E-11	3.33E-11	4.25E-11	4.18E-11	3.47E-11	5.15E-11
	MKL-CUFFT		3.46E-11	3.51E-11	3.80E-11	5.43E-11	5.25E-11	4.16E-11	5.94E-11
	FFTW-CUFFT		3.03E-11	2.42E-11	2.89E-11	4.12E-11	4.35E-11	3.58E-11	5.53E-11

Table 5 Errors for zero matrix with 1 source of strength 100,000 (Case 3).

As the tables show, the difference between double precision and single precision FFT amounts to between a half to one decimal digit in all cases examined. Differences between the results from FFTW, Intel MKL and CUFFT are in agreement with each other within the expected tolerance. Variations across the three methods fall within expected tolerances and are due to different ordering of operations, different plans etc.

In use, FFTW, Intel MKL and CUFFT can be used to all effects interchangeably to compute the FFTs.

Case 2 and particularly 3 deserve some further comments. It had been argued that double precision FFT would be required from single precision visibilities in order to extract the weaker structures in a sky image. This appears only partially supported by the results. Consistently, across the FFT Libraries and problem sizes, the difference between single and double precision results amount to around half a decimal digit (roughly speaking a factor of 4 or less). Single precision results have been shown to have error of order $\mathcal{O}(10^{-7})$ in line with expectations (and hopes).

Whether the extra costs (and memory) required by employing double precision are justifiable for minor gains in errors is a question beyond the scope of this report, but which should none the less be raised as appropriate.

6. Computational Performance

Table 6 reports the computational time (benchmarks) in seconds for a number of problem sizes only for the model problem 1. Other model problems return very similar performance figures.

Single Precision 2D FFT execution time (all times in seconds)											
N	FFTW	Intel MKL (N.threads are indicated)								CUFFT	
		1	2	4	6	8	10	12	14	memcpy	
500	0.0012	0.0011	0.0006	0.0004	0.0003	0.0003	0.0002	0.0002	0.0002	0.0002	0.0008
512	0.0009	0.0008	0.0004	0.0004	0.0003	0.0002	0.0002	0.0002	0.0002	0.0002	0.0008
1000	0.0053	0.0048	0.0027	0.0015	0.0010	0.0008	0.0007	0.0006	0.0005	0.0003	0.0023
1024	0.0047	0.0046	0.0024	0.0016	0.0011	0.0009	0.0007	0.0006	0.0006	0.0003	0.0024
2000	0.0224	0.0251	0.0136	0.0082	0.0059	0.0051	0.0044	0.0040	0.0040	0.0008	0.0092
2048	0.0233	0.0265	0.0140	0.0098	0.0067	0.0057	0.0047	0.0044	0.0041	0.0007	0.0095
4000	0.1103	0.1164	0.0613	0.0398	0.0260	0.0215	0.0205	0.0184	0.0176	0.0030	0.0357
4096	0.1166	0.1226	0.0640	0.0421	0.0282	0.0257	0.0222	0.0213	0.0204	0.0025	0.0365
8000	0.5078	0.5192	0.2731	0.1670	0.1200	0.0970	0.0934	0.0803	0.0800	0.0115	0.1423
8192	0.5157	0.5469	0.2889	0.1850	0.1315	0.1090	0.0907	0.0858	0.0858	0.0094	0.1455
10000	0.8746	0.7818	0.3979	0.2268	0.1649	0.1458	0.1337	0.1288	0.1273	0.0232	0.2285
15000	2.2398	1.7923	0.9252	0.5621	0.4548	0.3704	0.3376	0.3195	0.3146	0.0667	0.5292
16384	2.3225	2.2198	1.1470	0.7371	0.4892	0.4436	0.3856	0.3642	0.3338	0.0444	0.5872

Double Precision 2D FFT execution time (all times in seconds)											
N	FFTW	Intel MKL (N.threads are indicated)								CUFFT	
		1	2	4	6	8	10	12	14	memcpy	
500	0.0016	0.0015	0.0008	0.0006	0.0004	0.0003	0.0003	0.0002	0.0002	0.0002	0.0013
512	0.0015	0.0013	0.0007	0.0005	0.0004	0.0003	0.0002	0.0002	0.0002	0.0002	0.0013
1000	0.0083	0.0074	0.0039	0.0021	0.0014	0.0011	0.0009	0.0008	0.0007	0.0005	0.0045
1024	0.0089	0.0075	0.0041	0.0025	0.0017	0.0015	0.0013	0.0012	0.0011	0.0004	0.0046
2000	0.0363	0.0406	0.0212	0.0127	0.0091	0.0077	0.0072	0.0068	0.0068	0.0015	0.0181
2048	0.0467	0.0417	0.0223	0.0145	0.0107	0.0105	0.0086	0.0075	0.0072	0.0013	0.0187
4000	0.1789	0.1974	0.0974	0.0581	0.0422	0.0352	0.0328	0.0306	0.0309	0.0056	0.0714
4096	0.1881	0.2053	0.1056	0.0675	0.0494	0.0474	0.0409	0.0400	0.0391	0.0050	0.0731
8000	0.8133	0.7796	0.3999	0.2353	0.1761	0.1559	0.1627	0.1319	0.1316	0.0223	0.2844
8192	0.8401	0.7909	0.4099	0.2621	0.1927	0.1792	0.1491	0.1440	0.1491	0.0230	0.2951
10000	1.3616	1.2009	0.6096	0.3621	0.2631	0.2314	0.2284	0.2172	0.2077	0.0467	0.4549
15000	3.2772	2.8136	1.4506	0.8547	0.6303	0.5364	0.5077	0.5581	0.6193	0.1339	1.0496
16384	3.6416	3.3374	1.8158	1.0501	0.7700	0.7104	0.6408	0.6058	0.5709	0.0923	1.1759

Table 6 Execution times in seconds for model problem 1.

Notice the following:

- Cost for the plan generation (very expensive for FFTW) are neither considered nor included.
- Intel MKL is fully multithreaded. Computational times are reported for various numbers of threads.

Single Precision 2D FFT performance (all figures in GFlops/second)											
N	FFTW	Intel MKL (N.threads are indicated)								CUFFT	
		1	2	4	6	8	10	12	14		memcpy
500	9.4	10.6	20.0	28.0	37.4	43.1	56.0	62.3	59.0	57.2	14.5
512	12.8	14.0	26.8	32.8	45.4	49.2	69.4	73.7	78.6	67.0	14.8
1000	9.5	10.3	18.4	33.4	47.9	63.9	73.3	89.0	92.3	160.8	21.2
1024	11.2	11.4	22.0	32.0	49.5	57.6	71.8	85.9	87.4	191.2	21.6
2000	9.8	8.8	16.1	26.6	37.5	43.3	50.0	55.0	54.6	270.1	23.8
2048	9.9	8.7	16.5	23.6	34.6	40.8	48.8	53.0	57.0	314.7	24.4
4000	8.7	8.2	15.6	24.1	36.9	44.5	46.7	52.1	54.4	323.4	26.8
4096	8.6	8.2	15.7	23.9	35.7	39.2	45.3	47.2	49.4	410.0	27.6
8000	8.2	8.0	15.2	24.9	34.6	42.8	44.4	51.7	51.8	362.4	29.1
8192	8.5	8.0	15.1	23.6	33.2	40.0	48.1	50.8	50.8	465.3	30.0
10000	7.6	8.5	16.7	29.3	40.3	45.6	49.7	51.6	52.2	286.3	29.1
15000	7.0	8.7	16.9	27.8	34.3	42.1	46.2	48.8	49.6	234.1	29.5
16384	8.1	8.5	16.4	25.5	38.4	42.4	48.7	51.6	56.3	423.1	32.0

Double Precision 2D FFT performance (all figures in GFlops/second)											
N	FFTW	Intel MKL (N.threads are indicated)								CUFFT	
		1	2	4	6	8	10	12	14		memcpy
500	6.9	7.5	14.4	20.2	30.4	38.0	44.3	49.1	56.6	45.2	8.4
512	7.9	9.0	17.0	22.5	33.1	41.1	50.0	56.2	58.1	52.2	8.8
1000	6.0	6.7	12.9	24.2	34.9	44.8	56.9	66.0	69.9	104.9	11.2
1024	5.9	7.0	12.7	21.2	31.5	35.6	41.8	45.2	49.5	125.4	11.3
2000	6.0	5.4	10.3	17.3	24.2	28.4	30.5	32.0	32.3	145.5	12.1
2048	4.9	5.5	10.3	15.9	21.6	21.9	26.7	30.8	32.2	172.4	12.4
4000	5.4	4.9	9.8	16.5	22.7	27.2	29.2	31.3	30.9	170.2	13.4
4096	5.4	4.9	9.5	14.9	20.4	21.2	24.6	25.2	25.7	200.0	13.8
8000	5.1	5.3	10.4	17.6	23.6	26.6	25.5	31.4	31.5	186.0	14.6
8192	5.2	5.5	10.6	16.6	22.6	24.3	29.2	30.3	29.3	189.6	14.8
10000	4.9	5.5	10.9	18.3	25.3	28.7	29.1	30.6	32.0	142.4	14.6
15000	4.8	5.5	10.8	18.3	24.8	29.1	30.7	28.0	25.2	116.6	14.9
16384	5.2	5.6	10.3	17.9	24.4	26.4	29.3	31.0	32.9	203.6	16.0

Table 7 Performance figures in Gflops/second for model problem 1 (using $N_{ops} = 5 N^2 \log_2 N$).

For CUFFT, two sets of results are reported: one with data already placed in the GPU (no heading), and one including data transfer to the GPU (heading: “memcpy”)

Very approximate computational rates are shown in the table below, using the number of operations $N_{ops} = 5 N^2 \log_2 N$ (discussed in the section “Computational Costs and Complexity”

above). The tables clearly show that CUFFT has very considerable speed advantage over the Intel MKL Library. However, when data transfer to GPU is required and considered, it is the Intel MKL FFT, which is clearly better.

The single-thread performance of FFTW and Intel MKL are very comparable. The two methods differ in the plan selection and algorithmic details.

7. Conclusions

The analysis of the tests carried out lead to the following conclusions.

1. A simple “exact” computational cost model cannot be generated, given the strong dependence on algorithms, prime factors, software organization, data transfer rates, etc. However, we have used throughout a consistent number of operations and the results show that performance figures can therefore be seen and analysed consistently.
2. For single precision visibilities, the computational costs incurred in using double precision rather than single precision FFTs do not appear justified except, perhaps, in special cases
 - a. Differences between single and double precision correspond, in all cases examined, to adding an extra half decimal digit (i.e. 2-3 bits at most) to results that are already accurate in all cases to almost numerical precision (7 decimal digits).
 - b. Computational costs are higher for double precision particularly in the case, as expected, for CUFFT (double the cost)
3. Point 2. above does not apply, of course, to double precision visibilities with a number of significant digits exceeding single precision (about 7 decimal digits).
4. The different libraries and platforms have been shown to produce solutions very much consistent with each other. As far as it can be ascertained, the libraries and platforms studied here must be viewed as wholly interchangeable.
5. Performance figure support the possibility that more than one solution would be needed.
 - a. Because of the low computation to data transfer ratio (low data reuse) performance figures are in all cases $\leq 15\%$.
 - b. Consistently, double precision is twice as expensive as single precision for all FFTs examined, as expected
 - c. FFTW.
In line of principle, it would be possible to use multi-threaded FFTW, but that would require special compilation, etc. Incorrect compilation of FFTW would result in poor performance. It is not recommended to use FFTW on Intel CPUs.
 - d. MKL.
For use on CPUs, Intel MKL Library should be used. Performance is slightly better than FFTW using a single core, and shows good scalability when the low computational density of FFTs are taken into account.
 - e. CUFFT.
Performance for data already residing in the GPU memory is extremely high (up to ten times higher than multi-threaded MKL). However, when data do not already reside in the GPU, PCI-Express represents a very considerable bottleneck and multi-threaded MKL FFTs should be used instead. It should also be noticed that forthcoming PCI_E4 (not yet released) would ameliorate slightly (up to 32 GB/sec) the situation, but not resolve it. Hence, a heterogeneous computing solution, where data are transferred to GPU for specific computations) should be avoided.

All the libraries studied here have been produced at great cost by teams of specialists. FFTs are very far from trivial and are the subject of advanced investigations. Such libraries should be used whenever possible. Also, the efficiency of code compiled from sources depends very much on the compiler, the platform, the source etc.