Performance Optimization of Multithreaded 2D FFT on Multicore Processors: Challenges and Solution Approaches

Semyon Khokhriakov*, Ravi Reddy Manumachu†, and Alexey Lastovetsky†
School of Computer Science
University College Dublin
Dublin, Ireland
*Email: *{semen.khokhriakov}@ucdconnect.ie, †{ravi.manumachu,alexey.lastovetsky}@ucd.ie*

***Abstract*—Fast Fourier transform (FFT) is a key routine employed in application domains such as molecular dynamics, computational fluid dynamics, signal processing, image processing, and condition monitoring systems. Its performance on latest multicore platforms is therefore of paramount concern to the high performance computing community. The inherent complexities however in these platforms such as severe resource contention and non-uniform memory access (NUMA) pose formidable challenges.**

We study in this work the performance profiles of multithreaded 2D fast Fourier transforms provided in three highly optimized packages, FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT on a modern Intel Haswell multicore processor consisting of thirty-six cores. First, we show that all the three routines demonstrate drastic performance variations and therefore their average performances are considerably lower than their peak performances. The ratio of average to peak performance for the 2D FFT routines from the three packages are 40%, 30%, and 24%. We demonstrate that the average and peak performance of FFTW-2.1.5, last updated in 1999, is better than FFTW-3.3.7 suggesting that extensive machine optimization using architecture-specific techniques can be harmful in the long run since hardware platforms undergo drastic changes. We also show that while the average performance of Intel MKL FFT is better than FFTW-3.3.7, it is outperformed by FFTW-3.3.7 for many problem sizes. Also the width of the performance variations for Intel MKL FFT are severe compared to FFTW-3.3.7. Based on our study, we conclude that improving the average performance of FFT by removal of performance variations on modern multicore processors constitutes a tremendous research challenge.

We propose three possible solution approaches to remove the performance variations and suggest future directions.

***Keywords*-fast Fourier transform, multicore, data partitioning, load balancing, performance optimization, code tuning**

I. Introduction

Fast Fourier transform (FFT) is a key routine employed in application domains such as molecular dynamics, computational fluid dynamics, signal processing, image processing, and condition monitoring systems [1]–[5]. It is so fundamental that hardware vendors provide libraries containing 1D, 2D, and 3D FFT routines highly optimized for their processors. For example, highly optimized FFT routines for Intel processors are provided in the Intel Math Kernel library (Intel MKL) [6], for Nvidia CUDA GPUs in cuFFT [7], and for AMD processors in clFFT [8].

The theoretical computational complexity and arithmetic intensity of 2D FFT lies between those for highly memory-bound and highly compute-bound applications. Its computational complexity of $O(N^2 \times log_2 N)$ for a 2D FFT of complex input and output lies between those for highly memory-bound applications ($O(N^2)$ for matrix-vector multiplication MxV of a dense matrix $N \times N$) and highly compute-bound applications ($O(N^3)$ for matrix-matrix multiplication MxM of two dense $N \times N$ matrices). Its arithmetic intensity (I_A) ($I_A = \frac{\#flops}{\#\text{memory accesses}} = O(log_2 N)$) lies between those for highly memory-bound applications (I_A for MxV is 1) and highly compute-bound applications (I_A for MxM is N). Several code tuning techniques (multithreading, Fused Multiply-Add (FMA), SIMD acceleration using specialized instruction sets such as SSE2, AltiVec, etc) have been used to optimize it for different processor architectures.

The performance of FFT, therefore, on modern multicore platforms is of paramount concern to the high performance computing community. To address the twin concerns of increasing performance and high energy efficiency, modern multicore platforms today feature tight integration of cores contending for shared on-chip resources such as Last Level Cache (LLC) and interconnect (For example: Intels Quick Path Interconnect [9]), leading to severe resource contention and non-uniform memory access (NUMA). These inherent complexities however pose significant challenges to FFT achieving good performance on these platforms.

To elucidate the challenges, we use three multithreaded FFT applications for comparison written using the packages FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT. The packages, FFTW-2.1.5 and FFTW-3.3.7, are open-source and are reported to offer FFT routines that perform better than other publicly available FFT software. The FFTW interface is so popular that hardware vendor libraries ([6], [7]) offer optimized implementations of the interface for their processors.

The FFTW-3.3.7 package is installed with multithreading, SSE/SSE2, AVX2, and FMA (fused multiply-add) optimizations enabled. For Intel MKL FFT, we do not use any

Technical Specifications	Intel Haswell Server
Processor	Intel Xeon CPU E5-2699 v3 @ 2.30GHz
OS	CentOS 7.1.1503
Microarchitecture	Haswell
Memory	256 GB
Core(s) per socket	18
Socket(s)	2
NUMA node(s)	2
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	46080 KB
NUMA node0 CPU(s)	0-17,36-53
NUMA node1 CPU(s)	18-35,54-71

Table I: Specification of the Intel Haswell server used to construct the performance profiles.

special environment variables. The version of MKL used is 2017.0.4.

The performance profiles for the applications are obtained on a modern Intel Haswell multicore server consisting of 2 sockets of 18 physical cores each (specification shown in Table I). All the FFT applications compute a 2D-DFT of complex signal matrix of size $N \times N$ using 36 threads. We do not use any special environment affinity variables during the execution of the application. The total number of problem sizes $N \times N$ experimented is around 1000 with N ranging from 128 to 64000 with a step size of 64, $\{128, 192, ..., 64000\}$. The speed of execution of a 2D-DFT of complex signal matrix of size $N \times N$ is calculated using the formula: $\frac{5.0*N^2*\log_2(N)}{t}$, where t is the time of execution of the 2D-DFT.

We experiment with three planner flags, {*FFTW_ESTIMATE*, *FFTW_MEASURE*, *FFTW_PATIENT*}. The performance profiles are shown for only one planner flag, *FFTW_ESTIMATE*. We have performed experiments with two other planner flags, {*FFTW_MEASURE*, *FFTW_PATIENT*}. Table II summarizes for FFTW-3.3.7 the execution times for a subset of exprimented problem sizes. The execution times for these flags however are prohibitively larger compared to *FFTW_ESTIMATE* and severe variations are present. The long execution times are due to the lengthy times to create the plans because *FFTW_MEASURE* tries to find an optimized plan by computing several FFTs whereas *FFTW_PATIENT* considers a wider range of algorithms to find a more optimal plan.

We will be referring frequently to width of performance variations in a performance profile. It is related to the difference of speed between two subsequent local minima (s_1) and maxima (s_2) and is defined below:

$$variation(\%) = \frac{|s_1 - s_2|}{\min(s_1, s_2)} \times 100 \quad (1)$$

To make sure the experimental results are reliable, we

N	FFTW_ESTIMATE	FFTW_MEASURE	FFTW_PATIENT
20160	3	31	5015
20480	16	41	2549
20672	6.5	3004	8228
21120	3.6	31	2746
21440	4	32	1367
21632	14.5	2937	9754

Table II: Execution times in seconds for FFTW-3.3.7 on the Intel Haswell multicore server for three different planner flags.

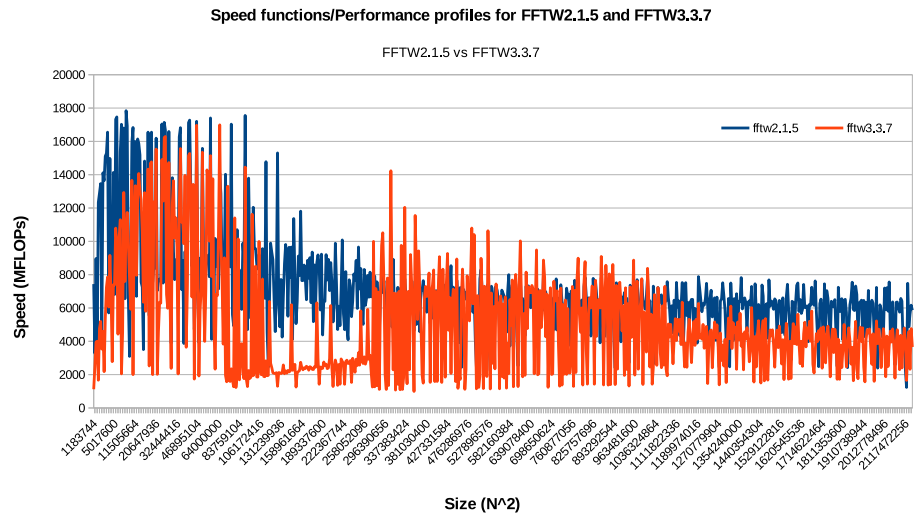
follow a strict statistical methodology, which we describe in detail in the section to follow. Briefly, for each data point in the performance profile, the automation software executes the FFT application repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test. The speed/performance values shown in the graphical plots throughout this work are the sample means obtained using Student's t-test.

Figure 1a, 1b show the performance profiles of FFTW 2.1.5 versus FFTW 3.3.7. Following are the key observations:

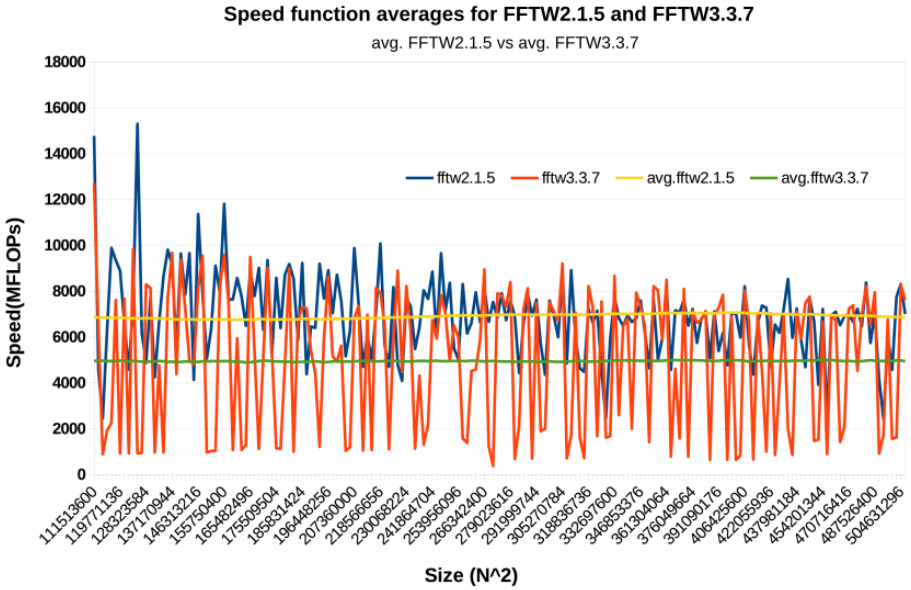
- We can see that the width of performance variations in FFTW-3.3.7 is greater than that for FFTW-2.1.5.
- The peak performance of FFTW-3.3.7 is 16989 MFLOPs ($N = 8000$) whereas that for FFTW-2.1.5 is 17841 MFLOPs ($N = 2816$).
- The average speeds of FFTW-2.1.5 and FFTW-3.3.7 are 7033 MFLOPs and 5065 MFLOPs. FFTW-2.1.5 is better than FFTW-3.3.7 by around 38% (on an average). There are 529 problem sizes (out of 1000) where the performance of FFTW-2.1.5 is better than FFTW-3.3.7.

Figures 2a, 2b present the performance comparisons between FFTW-2.1.5 and Intel MKL FFT. The most important observations are as follows:

- The peak performance of FFTW-2.1.5 is 17841 MFLOPs ($N = 2816$) whereas that for Intel MKL FFT is 39424 MFLOPs ($N = 1792$).
- The average performance of Intel MKL FFT is around 9572 MFLOPs versus 7033 MFLOPs for FFTW-2.1.5. So, on an average, Intel MKL FFT is 36% better than FFTW-2.1.5. Despite Intel MKL FFT demonstrating better average performance than FFTW-2.1.5, its width of variations is greater than that for FFTW-2.1.5. One can see that the variations of Intel MKL FFT fill the picture. This is the reason why Intel MKL FFT demonstrates comparatively poorer average performance despite its high peak performance.
- There are 162 problem sizes (out of 1000) where FFTW-2.1.5 is better than Intel MKL FFT.



(a)



(b)

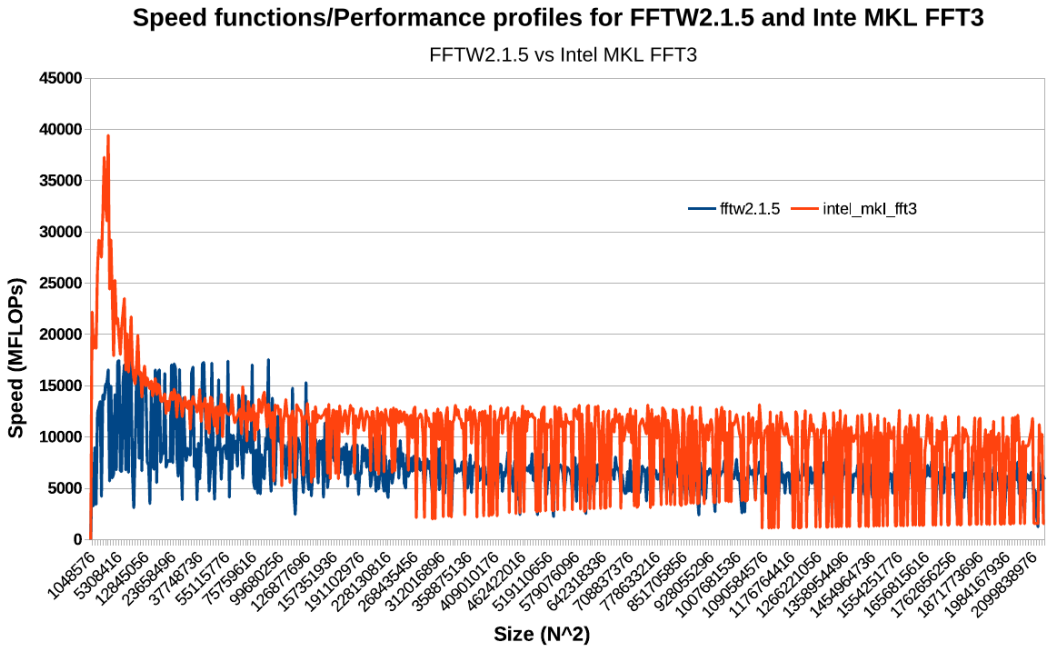
Figure 1: (a). Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-2.1.5 and FFTW-3.3.7. The 2D-FFT applications are executed using 36 threads on a Intel multicore server consisting of two sockets of 18 cores each. (b). The average speeds of FFTW-2.1.5 vs FFTW-3.3.7.

Figures 3a, 3b present the performance comparisons between FFTW-3.3.7 and Intel MKL FFT. The crucial observations are as follows:

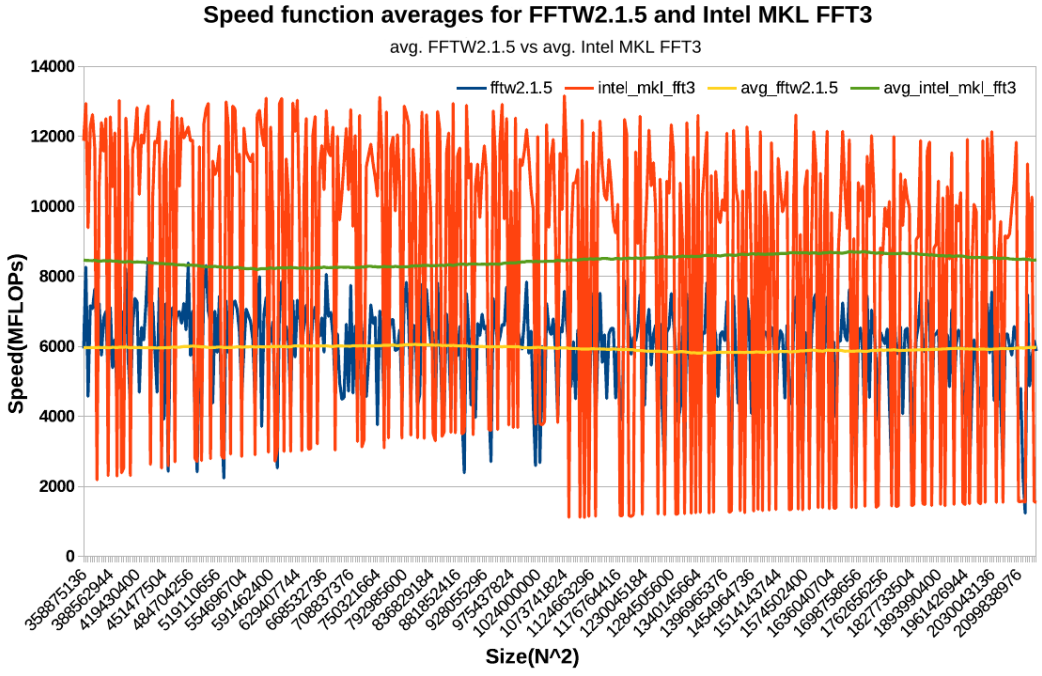
- The peak performance of FFTW-3.3.7 is 16989 MFLOPs ($N = 8000$) whereas that for Intel MKL FFT is 39424 MFLOPs ($N = 1792$).
- The average performance of FFTW-3.3.7 is 5065 MFLOPs and Intel MKL FFT is 9572 MFLOPs, 89% faster. There are 199 problem sizes (out of 1000) where FFTW-3.3.7 outperforms Intel MKL FFT.
- The width of variations for Intel MKL FFT is noticeably greater than that for FFTW-3.3.7.

Based on these comparisons, we make the following important conclusions:

- Extensive nodal optimization of FFT using highly architecture-specific techniques can be harmful in the long run since hardware platforms undergo drastic changes. An exemplar is FFTW-2.1.5 versus FFTW-3.3.7. FFTW-2.1.5, last updated in 1999, outperforms FFTW-3.3.7, which undergoes constant revisions in

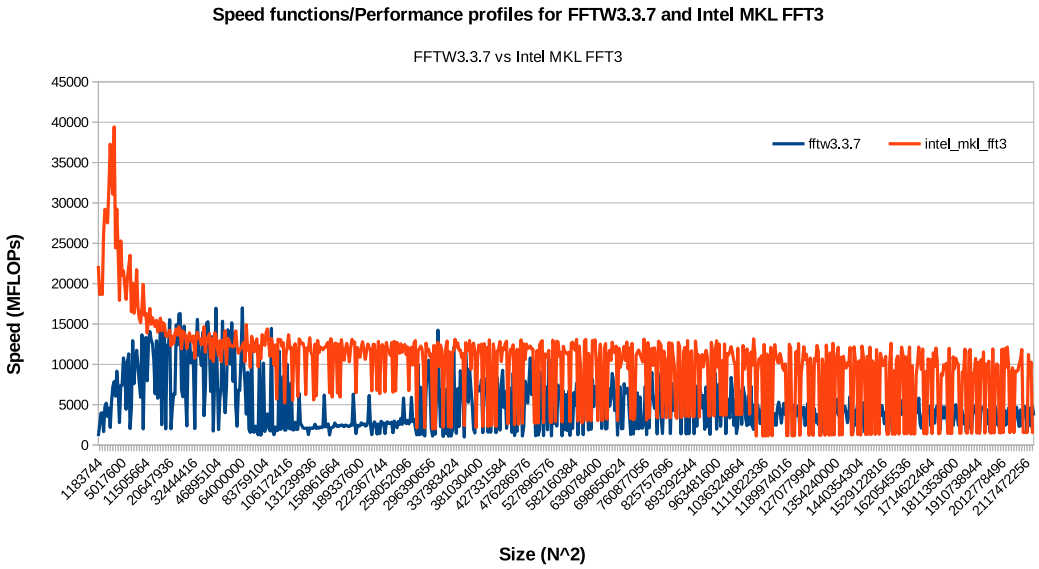


(a)

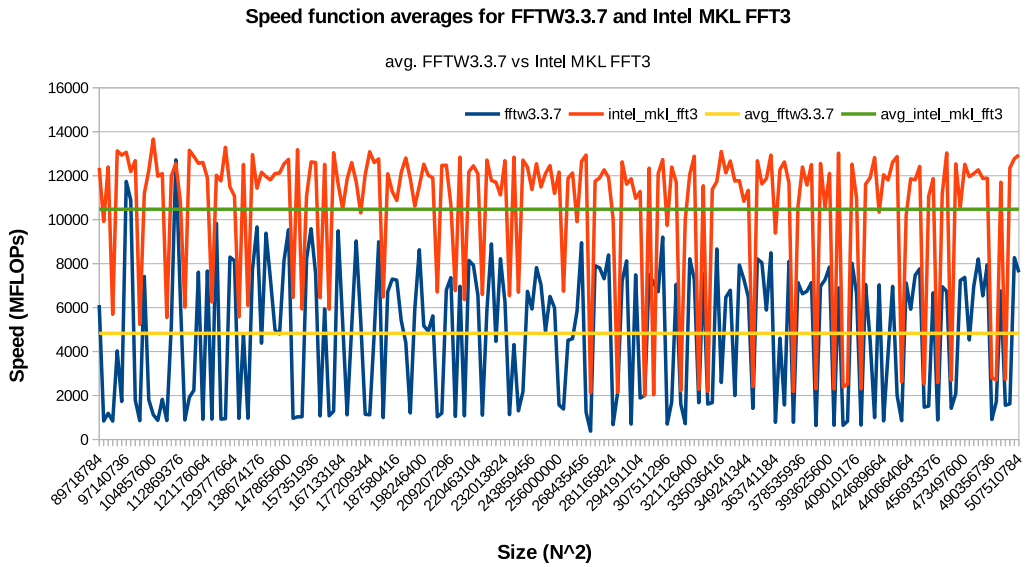


(b)

Figure 2: (a). Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-2.1.5 and Intel MKL FFT. The 2D-FFT applications are executed using 36 threads on a Intel multicore server consisting of two sockets of 18 cores each.. (b). The average speeds of FFTW-2.1.5 and Intel MKL FFT.



(a)



(b)

Figure 3: (a). Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-3.3.7 and Intel MKL FFT. The 2D-FFT applications are executed using 36 threads on a Intel multicore server consisting of two sockets of 18 cores each. (b). The average speeds of FFTW-3.3.7 and Intel MKL FFT.

terms of code optimizations.

- An open source package may perform better than highly optimized vendor package since it employs portable optimizations. A good example is FFTW-3.3.7 versus Intel MKL FFT. Intel MKL FFT is highly optimized for some specific problem sizes but exhibits poor performance for the rest. This can be seen from the width of its performance variations. Though the average performance of FFTW-3.3.7 is lesser than Intel MKL FFT, it outperforms Intel MKL FFT for many problem sizes and its variations are lesser.

We believe that these performance variations will become typical because chip manufacturers are increasingly favouring and thereby rapidly progressing towards tighter integration of processor cores, memory, and interconnect in their products.

There are three solution approaches that can be employed for the optimization of 2D-DFT computation by removal of performance variations. These approaches can be applied, in general, for optimization of data-parallel applications on modern multicore processors for performance.

- *Optimization through source code analysis and tuning*: This approach requires source code modification. It lacks portability if architecture-specific optimizations are used. It has other disadvantages, the most crucial being the disproportion between the time spent tuning the code and the continued long-term portable performance improvements.
- *Optimization using solutions to larger problem sizes with better performance*: This is a portable approach. There has to be a performance model, which given workload size N to solve will output the problem size $N_l(> N)$ that is to be used for padding. While programmatically extending 1D arrays logically is easy, it is not the case for 2D arrays such as matrices and multidimensional arrays.
- *Optimization using model-based parallel computing*: In the current era of multicores where processors have abundant number of cores, one can partition the workload between identical multithreaded routines (abstract processors) and execute them in parallel. This is a portable approach.

We will describe these approaches in detail in the following section.

The rest of the paper is structured as follows. Section 2 presents the experimental methodology to build the performance profiles. Section 3 contains the three possible solution approaches we propose to remove the performance variations. Section 4 contains the related work. Section 5 concludes the paper.

II. Experimental Methodology to Build the Performance profiles

We followed the methodology described below to make sure the experimental results are reliable:

- The server is fully reserved and dedicated to these experiments during their execution. We also made certain that there are no drastic fluctuations in the load due to abnormal events in the server by monitoring its load continuously for a week using the tool *sar*. Insignificant variation in the load was observed during this monitoring period suggesting normal and clean behaviour of the server.
- When an application is executed, it is bound to the physical cores using the *numactl* tool.
- To obtain a data point in the performance profile, the application is repeatedly executed until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test. When we mention a single number such as floating-point performance (in MFLOPs or GFLOPs), it is assumed that we are referring to the sample mean determined using the Student's t-test.

  The function $MeanUsingTtest$, shown in Algorithm 1, describes how the sample mean for a data point is determined. For each data point, the function is invoked, which repeatedly executes the application app until one of the following three conditions is satisfied:
  1) The maximum number of repetitions ($maxReps$) have been exceeded (Line 3).
  2) The sample mean falls in the confidence interval (or the precision of measurement eps has been achieved) (Lines 13-15).
  3) The elapsed time of the repetitions of application execution has exceeded the maximum time allowed ($maxT$ in seconds) (Lines 16-18).

  So, for each data point, the function $MeanUsingTtest$ is invoked and the sample mean $mean$ is returned at the end of invocation. The function $Measure$ measures the execution time using *gettimeofday* function.
- In our experiments, we set the minimum and maximum number of repetitions, $minReps$ and $maxReps$, to 10 and 100000. The values of $maxT$, cl, and eps are set to 3600, 0.95, and 0.025. If the precision of measurement is not achieved before the maximum number of repeats have been completed, we increase the number of repetitions and also the maximum elapsed time allowed. Therefore, we make sure that statistical confidence is achieved for all the data points that we use in our performance profiles.

Algorithm 1 Function determining the sample mean using Student's t-test.

1: **procedure** MEANUSINGTTEST($app, minReps, maxReps, maxT, cl, accuracy, repsOut, clOut, etimeOut, epsOut, mean$)

Input:

The application to execute, app
The minimum number of repetitions, $minReps \in \mathbb{Z}_{>0}$
The maximum number of repetitions, $maxReps \in \mathbb{Z}_{>0}$
The maximum time allowed for the application to run, $maxT \in \mathbb{R}_{>0}$
The required confidence level, $cl \in \mathbb{R}_{>0}$
The required accuracy, $eps \in \mathbb{R}_{>0}$

Output:

The number of experimental runs actually made, $repsOut \in \mathbb{Z}_{>0}$
The confidence level achieved, $clOut \in \mathbb{R}_{>0}$
The accuracy achieved, $epsOut \in \mathbb{R}_{>0}$
The elapsed time, $etimeOut \in \mathbb{R}_{>0}$
The mean, $mean \in \mathbb{R}_{>0}$

2: $reps \leftarrow 0$; $stop \leftarrow 0$; $sum \leftarrow 0$; $etime \leftarrow 0$
3: **while** ($reps < maxReps$) and (!$stop$) **do**
4: $st \leftarrow$ MEASURE($TIME$)
5: EXECUTE(app)
6: $et \leftarrow$ MEASURE($TIME$)
7: $reps \leftarrow reps + 1$
8: $etime \leftarrow etime + et - st$
9: $ObjArray[reps] \leftarrow et - st$
10: $sum \leftarrow sum + ObjArray[reps]$
11: **if** $reps > minReps$ **then**
12: $clOut \leftarrow$ fabs(gsl_cdf_tdist_Pinv(cl, $reps - 1$)) $\times$ gsl_stats_sd($ObjArray$, 1, $reps$) / sqrt($reps$)
13: **if** $clOut \times \frac{reps}{sum} < eps$ **then**
14: $stop \leftarrow 1$
15: **end if**
16: **if** $etime > maxT$ **then**
17: $stop \leftarrow 1$
18: **end if**
19: **end if**
20: **end while**
21: $repsOut \leftarrow reps$; $epsOut \leftarrow clOut \times \frac{reps}{sum}$
22: $etimeOut \leftarrow etime$; $mean \leftarrow \frac{sum}{reps}$
23: **end procedure**

III. Performance Optimization of fast Fourier transform on Multicore Processors: Solution Approaches

In this section, we describe three solution approaches for the optimization of 2D FFT by removal of performance variations. These approaches can be applied, in general, for optimization of data-parallel applications on modern multicore processors for performance. We discuss the advantages and disadvantages of each approach.

Optimization through source code analysis and tuning: This is typically the first approach adopted to improve the performance of an application. The roofline model [10] is used to visually depict the trend of performance gains accrued from code tuning towards the theoretical peak performance of a multicore processor. Using this model, the high optimized scientific applications such as Intel Math Kernel Library (Intel MKL) (BLAS, FFT) consistently demonstrate the superior performance of their codes (such as BLAS) for new platforms.

It has following disadvantages:

- If the code is highly tuned to a specific vendor architecture, its portability to other vendor architectures suffers. It is also debatable (as we show in this paper for FFTW-2.1.5 and FFTW-3.3.7) if the performance improvements carry forward to different generations of the same architecture. Therefore, it lacks portable performance.
- Most high quality codes are proprietary and therefore their sources are not available for inspection and tuning. For example: BLAS, FFT packages that are part of Intel MKL library.
- It will require source code modification. Since the highly optimized packages such as FFTW are written with many man-years of effort for different generations of hardware, any source code change may entail extensive testing to ensure old functionality is not broken. Therefore, it is a time consuming process.

Optimization using solutions to larger problem sizes with better performance: Supposing we are solving a problem where the size of the matrix is N. In this approach, the solution to a larger problem size ($N_l > N$), which has better execution time than N, is used as solution for N. The common approach is the pad the input matrix to increase its problem size from N to N_l and zero the contents of the extra padded areas. It is also a technique that is widely used in different flavours (restructuring arrays, aggregation) to minimize cache conflict misses [11], [12], [13], [14]. It requires no source code modification of the optimized package.

While it is a portable approach, it also has some disadvantages.

- There has to be a performance model, which given N will provide the problem size N_l that is to be used

for padding. In this work, the performance profiles (functional performance models (FPMs)) provide this information.
- While programmatically extending 1D arrays logically is easy, it is not the case for 2D arrays such as matrices and multidimensional arrays. One inexpensive technique is to locally copy the input signal matrix of size N to a work matrix of size N_l, compute 2D FFT of the work matrix and copy the relevant content back to the signal matrix, which is returned to the user. One drawback is the extra memory used for the work matrix.

Optimization using model-based parallel computing: Finally, we propose the third approach, which employs parallel computing. In the current era of multicores where processors have abundant number of cores, one can partition the workload between identical multithreaded routines (abstract processors) and execute them in parallel. This method can be an effective nodal optimization technique especially when it employs realistic performance models of computation and efficient data partitioning algorithms that use the models as input.

Its advantages are:

- It is portable when the performance models of computation used in the data partitioning algorithms do not use architecture-specific parameters.
- It requires no source code modification of the optimized package.
- Less time-consuming programming effort is involved, which is to distribute the workload between identical multithreaded routines (abstract processors) and execute them in parallel.

To distribute the data between the identical multithreaded routines (abstract processors), one can start with homogeneous distribution. But to squeeze out the maximum performance, realistic and accurate performance models and efficient data partitioning algorithms are necessary. The model must not be based on parameters, which are architecture-specific (For example: performance monitoring events (PMCs)). This would compromise the portability of this approach.

Lastovetsky et al. [15] employ this technique to improve the performance of a scientific application, Multidimensional Positive Definite Advection Transport Algorithm (MPDATA), on a Xeon Phi co-processor. Lastovetsky et al. [16], Reddy et al. [17], and Khaleghzadeh et al. [18] are theoretical works that present novel data partitioning algorithms employing this technique for minimization of time and energy of computations for the most general performance and energy profiles of data-parallel applications executing on homogeneous and heterogeneous multicore clusters.

IV. Related Work

We classify our survey of related literature into following categories: a). FFT libraries, b). Parallel FFT solutions for homogeneous and heterogeneous platforms, and c). FFT solutions for GPUs.

A. FFT Libraries

The Fastest Fourier Transform in the West (FFTW) [19] is a software library for computing discrete Fourier transforms (DFTs). It provides routines utilizing threads for parallel one- and multi-dimensional transforms of both real and complex data, and multi-dimensional transforms of real and complex data for parallel machines supporting MPI.

Pekurovsky et al. [20] present a library P3DFFT, which computes fast Fourier transforms (FFTs) in three dimensions by using two-dimensional domain decomposition. Li et al. [21] provide a library offering three-dimensional distributed FFTs using MPI. OpenFFT [22] is an open source parallel package for computing multi-dimensional Fast Fourier Transforms (3-D and 4-D FFTs) of both real and complex numbers of arbitrary input size.

The Intel Math Kernel library (Intel MKL) [6] provides an interface for computing a discrete Fourier transform in one, two, or three dimensions with support for mixed radices. It provides DFT routines for single-processor or shared-memory systems, and for distributed-memory architectures.

Akin et al. [23] present FFTs that are optimized for DRAM by deriving algorithms using custom data layouts and that use efficient memory access patterns

B. Parallel FFT solutions for homogeneous and heterogeneous platforms

Averbuch et al. [24] present a parallel version of the CooleyTukey FFT algorithm for MIMD multiprocessors and demonstrate efficiency of 90% on a message-passing IBM SP2 computer.

Chen et al. [25] analyze the optimization challenges and opportunities of both 1D and 2D FFT including problem decomposition, load balancing, work distribution, and data-reuse together with the exploiting of the C64 architecture features on the IBM Cyclops-64 chip architecture.

Ayala et al. [26] propose a parallel FFT implementation based on 2D domain decomposition and they demonstrate scalability of their solution on extreme scale computers. Almeide et al. [27] consider parallelization of the bidimensional FFT-2D on heterogeneous system using master-slaves approaches. Dmitruk et al. [28] use a 1D domain decomposition algorithm for performance improvement of 3D real FFT. They present techniques for reducing the cost of communications in the communication-intensive transpose operation of their algorithm.

Jung et al. [29] introduce two schemes based on the volumetric decomposition for the optimization of hybrid (MPI + OpenMP) parallelization schemes of 3D FFT. In one scheme

1d_Alltoall, they apply five 1D all-to-all communications among fewer processors and in another, two 1D all-to-all communication and one 2D communication *(2d_Alltoall)*. They state that both schemes show good performance and scalability in 3D FFT calculations.

Song and Hollingsworth [30] present a scalable method for parallel 3D FFT that exploits computation-communication overlap. Their method employs non-blocking MPI collectives to the 2D decomposition method for parallel 3D FFT. Also, they auto-tune 3D FFT for optimization in system environments.

Steinbach et al. [31] present an open-source FFT benchmark suite that allows users to determine the best FFT routine from state-of-theart FFT implementations on a wide variety of hardware.

C. FFT Solutions for GPUs

We review research works that have proposed optimized FFT implementations for GPU platforms. Chen et al. [32] present optimized FFT implementations for GPU clusters. Gu et al. [33] propose out-of-card implementations for 1D, 2D, and 3D FFTs on GPUs. Wu et al. [34] present optimized multi-dimensional FFT implementations on CPU-GPU heterogeneous platforms where the input signal matrix is too large to fit in the GPU global memory. Naik et al. [35] demonstrate good performance improvement of FFT on their heterogeneous cluster compared to a homogeneous cluster.

V. Conclusion

Fast Fourier transform (FFT) is such a key routine employed in application domains such as molecular dynamics, computational fluid dynamics, signal processing, image processing, and condition monitoring systems that hardware vendors now provide optimized libraries for its computation for their processors. Its performance on latest multicore platforms is therefore of paramount concern to the high performance computing community. The inherent complexities however in these platforms such as severe resource contention and non-uniform memory access (NUMA) pose formidable challenges.

We demonstrated the challenges, which are the performance variations in the profiles of applications, by studying three highly optimized multithreaded 2D FFT packages, FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT on a modern Intel Haswell multicore processor consisting of thirty-six cores.

In summary, we showed that for the routines from the three packages, their average performances can be lower than their peak performances. The ratio of average to peak performance for the FFTW-2.1.5 is 40%, for the FFTW-3.3.7 is 30% and 24% for the Intel MKL FFT. We showed that FFTW-2.1.5 outperforms FFTW-3.3.7) by around 38% if considering average performance. Despite Intel MKL FFT being the fastest package for many problem sizes, its variations in the performance profile are severe and it performs poorly for many problem sizes compared to the other two packages. Therefore, we conclude that improving the average performance of the FFT routines by removal of variations is the main research challenge. We then proposed three possible solution approaches to address this challenge, discussed their advantages and disadvantages.

The benchmark codes used to build the performance profiles can be found at [36].

In our future work, we will study optimization of 2D and 3D FFT routines for performance on modern multicore platforms using model-based parallel computing methods presented in [16].

Acknowledgment

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

References

[1] W. Chu and B. Champagne, "A noise-robust fft-based auditory spectrum with application in audio classification," *IEEE Transactions on audio, speech, and language processing*, vol. 16, no. 1, pp. 137–150, 2008.

[2] A. Sapena-Bañó, M. Pineda-Sanchez, R. Puche-Panadero, J. Martinez-Roman, and D. Matić, "Fault diagnosis of rotating electrical machines in transient regime using a single stator currents fft," *IEEE Transactions on Instrumentation and Measurement*, vol. 64, no. 11, pp. 3137–3146, 2015.

[3] M. Kang, J. Kim, L. M. Wills, and J.-M. Kim, "Time-varying and multiresolution envelope analysis and discriminative feature analysis for bearing fault diagnosis." *IEEE Trans. Industrial Electronics*, vol. 62, no. 12, pp. 7749–7761, 2015.

[4] M. Naoues, D. Noguet, L. Alaus, and Y. Louët, "A common operator for fft and fec decoding," *Microprocessors and Microsystems*, vol. 35, no. 8, pp. 708–715, 2011.

[5] J. P. Barbosa, A. P. Ferreira, R. C. Rocha, E. S. Albuquerque, J. R. Reis, D. S. Albuquerque, and E. N. Barros, "A high performance hardware accelerator for dynamic texture segmentation," *Journal of Systems Architecture*, vol. 61, no. 10, pp. 639–645, 2015.

[6] I. Corporation, "Intel MKL FFT - fast fourier transforms," 2018. [Online]. Available: https://software.intel.com/en-us/mkl/features/fft

[7] cuFFT, "Optimized FFT routines for Nvidia graphics processors," 2018. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html

[8] clFFT, "Optimized FFT routines for AMD graphics processors," 2018. [Online]. Available: https://gpuopen.com/compute-product/clfft/

[9] QPI, "Intel quickpath interconnect," 2008. [Online]. Available: https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect

[10] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, Apr. 2009.

[11] K. Ishizaka, M. Obata, and H. Kasahara, "Cache optimization for coarse grain task parallel processing using inter-array padding," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2003, pp. 64–76.

[12] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral, "Forma: A framework for safe automatic array reshaping," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, Nov. 2007.

[13] C. Hong, W. Bao, A. Cohen, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "Effective padding of multidimensional arrays to avoid cache conflict misses," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. ACM, 2016, pp. 129–144.

[14] P. Jiang and G. Agrawal, "Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. ACM, 2017, pp. 24:1–24:11.

[15] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 787–797, 2017.

[16] A. Lastovetsky and R. Reddy, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2017.

[17] R. Reddy and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 160–177, 2017.

[18] H. Khaleghzadeh, R. Reddy, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of Data-Parallel applications on heterogeneous HPC platforms," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[19] FFTW, "Fastest fourier transform in the west," 2018. [Online]. Available: http://www.fftw.org/

[20] D. Pekurovsky, "P3DFFT: A framework for parallel computations of fourier transforms in three dimensions," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C192–C209, 2012.

[21] N. Li and S. Laizet, "2DECOMP and FFT - a highly scalable 2D decomposition library and FFT interface," in *Cray User Group 2010 conference*, 2010, pp. 1–13.

[22] T. V. T. Duy and T. Ozaki, "A decomposition method with minimum communication amount for parallelization of multi-dimensional FFTs," *Computer Physics Communications*, vol. 185, no. 1, pp. 153 – 164, 2014.

[23] B. Akin, F. Franchetti, and J. C. Hoe, "Ffts with near-optimal memory access through block data layouts: Algorithm, architecture and design automation," *Journal of Signal Processing Systems*, vol. 85, no. 1, pp. 67–82, Oct 2016.

[24] A. Averbuch and E. Gabber, "Portable parallel FFT for MIMD multiprocessors," *Concurrency: Practice and Experience*, vol. 10, no. 8, 1998.

[25] L. Chen, Z. Hu, J. Lin, and G. R. Gao, "Optimizing the fast fourier transform on a multi-core architecture," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–8.

[26] O. Ayala and L.-P. Wang, "Parallel implementation and scalability analysis of 3d fast fourier transform using 2d domain decomposition," *Parallel Computing*, vol. 39, no. 1, pp. 58 – 77, 2013.

[27] F. Almeida and L. M. Moreno, "Parallel fft-2d in heterogeneous systems."

[28] P. Dmitruk, L.-P. Wang, W. Matthaeus, R. Zhang, and D. Seckel, "Scalable parallel FFT for spectral simulations on a Beowulf cluster," *Parallel Computing*, vol. 27, no. 14, 2001.

[29] J. Jung, C. Kobayashi, T. Imamura, and Y. Sugita, "Parallel implementation of 3d fft with volumetric decomposition schemes for efficient molecular dynamics simulations," *Computer Physics Communications*, vol. 200, pp. 57–65, 2016.

[30] S. Song and J. K. Hollingsworth, "Computation–communication overlap and parameter auto-tuning for scalable parallel 3-d fft," *Journal of computational science*, vol. 14, pp. 38–50, 2016.

[31] P. Steinbach and M. Werner, "gearshifft – the fft benchmark suite for heterogeneous platforms," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds. Cham: Springer International Publishing, 2017, pp. 199–216.

[32] Y. Chen, X. Cui, and H. Mei, "Large-scale FFT on GPU clusters," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. ACM, 2010.

[33] L. Gu, J. Siegel, and X. Li, "Using GPUs to compute large out-of-card FFTs," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. ACM, 2011.

[34] J. Wu and J. JaJa, "Optimized FFT computations on heterogeneous platforms with application to the Poisson equation," *Journal of Parallel and Distributed Computing*, vol. 74, no. 8, 2014.

[35] V. H. Naik and C. S. Kusur, "Analysis of performance enhancement on graphic processor based heterogeneous architecture: A CUDA and MATLAB experiment," in *Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on*. IEEE, 2015, pp. 1–5.

[36] S. Khokhriakov and R. Reddy, "HCLFFT: Novel model-based methods for performance optimization of 2D discrete Fourier transform on multicore processors," 2018. [Online]. Available: https://git.ucd.ie/manumachu/hclfft.git