Hindawi
Scientific Programming
Volume 2021, Article ID 1780931, 8 pages
https://doi.org/10.1155/2021/1780931

Research Article

On the Feasibility of Fast Fourier Transform Separability Property for Distributed Image Processing

Arturo Téllez-Velázquez [1] **and Raúl Cruz-Barbosa** [2]

[1]*Computer Science Institute, CONACyT–Universidad Tecnológica de la Mixteca, Huajuapan de León, OAX 69000, Mexico*
[2]*Computer Science Institute, Universidad Tecnológica de la Mixteca, Huajuapan de León, OAX 69000, Mexico*

Correspondence should be addressed to Raúl Cruz-Barbosa; rcruz@mixteco.utm.mx

Received 23 May 2021; Accepted 13 August 2021; Published 25 August 2021

Academic Editor: Manuel E. Acacio

Given the high algorithmic complexity of applied-to-images Fast Fourier Transforms (FFT), computational-resource-usage efficiency has been a challenge in several engineering fields. Accelerator devices such as Graphics Processing Units are very attractive solutions that greatly improve processing times. However, when the number of images to be processed is large, having a limited amount of memory is a serious problem. This can be faced by using more accelerators or using higher-capability accelerators, which implies higher costs. The separability property is a resource in hardware approaches that is frequently used to divide the two-dimensional FFT work into several one-dimensional FFTs, which can be simultaneously processed by several computing units. Then, a feasible alternative to address this problem is distributed computing through an Apache Spark cluster. However, determining the separability-property feasibility in distributed systems, when migrating from hardware implementations, is not evident. For this reason, in this paper a comparative study is presented between distributed versions of two-dimensional FFTs using the separability property to determine the suitable way to process large image sets using both Spark RRDs and DataFrame APIs.

1. Introduction

In the image-processing field, two-dimensional convolution is the simplest way to filter images in a spatial domain. However, as long as the *kernel* size increases, processing times increase geometrically [1]. For this reason, the two-dimensional *Discrete Fourier Transform* (DFT), whose applications include image filtering in the frequency domain [2, 3], allows for the reduction (to some extent) of the complexity of operations with respect to two-dimensional convolutions.

Although two-dimensional DFTs improve filtering performance, they still represent a processing challenge due to their high computational complexity of $O(N^4)$ [4], which translates to very long processing times. One solution to this problem is the *Fast Fourier Transform* (FFT), which consists of a set of more efficient algorithms that computes two-dimensional DFTs in considerably shorter processing times.

Even though two-dimensional FFTs are more efficient, and a real benefit can be achieved in applications, they continue to show a high computational cost when processing high-resolution images. As a consequence, a new goal of researchers can be finding alternative ways to compute faster and more efficient FFTs, taking advantage of novel High Performance Computing (HPC) systems.

Recently, the parallel-nature two-dimensional FFTs have allowed users to take advantage of five different types of parallel hardware: Very-Large-Scale Integration (VLSI), Field Programmable Gate Array (FPGA), Graphics Processing Units (GPU), Digital Signal Processors (DSP), and Distributed Processing via clusters.

Most successful efforts for accelerating FFTs can be found in real-time hardware implementations. For example, Pythosh and Magnani [5] presented a VLSI implementation of a 256×256 two-dimensional FFT, modifying the original Cooley–Tukey algorithm to achieve a speed-up of 27x and a

computational complexity of $O(\sqrt{N^2 \log \sqrt{N}})$. This approach divided the original data in several submatrices to compute data concurrently.

Because FPGAs are popular due to their reconfiguration ability, a large number of successful approaches can be found in the literature [6–9]. For instance, Shirazi proposed processing FFTs on images [1] using an FPGA, with which a speed-up of about 180x is achieved. This proposal approaches the DFT separability property by performing several row-wise one-dimensional FFTs, and after these results, several column-wise one-dimensional FFTs are computed.

Another recent effort for performing more efficient FFTs is using other types of accelerators, i.e., Graphics Processing Units (GPU) [10, 11]. One of the most famous parallel FFTs can be found in the NVIDIA® Compute Unified Device Architecture (CUDA) Toolkit. CUDA includes the cuFFT library [12], which provides CUDA-accelerated FFT algorithms that outperform FFTs 10x faster than CPU implementations. Although GPU scalability is more feasible than FPGA/VLSI scalability, increasing the number of accelerators implies higher costs.

Some other FFT implementations take advantage of multicore processing. For example, Kharin et al. [13] provide a DSP software implementation that concurrently computes radix-2 and radix-4 FFTs, achieving valid results up to 3 times faster than sequential implementations. Another implementation uses serial I/O ports to build a DSP network, where every DSP computes part of the entire FFT job. For instance, an FFT parallel implementation is performed with two DSPs [14], where a worker DSP cooperates with a master DSP to concurrently compute several FFTs, achieving a speed-up of up to 2x.

However, hardware-accelerator proposals present two disadvantages: low computational resources and low scalability. As a consequence, they cannot operate with large image sets and the possibility of increasing the number of processing elements is practically null, unfeasible, or very expensive.

Because of this, alternative implementations use *distributed systems* to increase the amount of computational resources in order to scale up processing. For instance, a cluster tool that uses Apache Spark's [15] Resilient Distributed Datasets (RDD) for analyzing seismic data (used in the oil industry) is presented in [16]. It integrates the Breeze [17] FFT library, achieving a speed-up of about 150x using a cluster with 576 processing cores. Another Spark RDD implementation is proposed by Yang et al. [18], who presented a distributed version of the Cooley–Tukey FFT. Although this is not a two-dimensional approach, it uses the divide-and-conquer strategy by breaking down one-dimensional FFTs into several simpler data structures. Here, each structure is computed in a distributed way and results are merged to achieve the final outcome.

Message-Passing-Interface (MPI) distributed approaches [19–22] take advantage of the DFT separability property to distribute multidimensional FFT loads, dividing the processing of several one-dimensional FFTs. This approach focuses on minimizing network traffic to avoid latencies that could degrade FFT performance. However, it can be scaled up only in a single multidimensional array (big array), leaving aside applications that need to process large image sets.

Due to the great variety of FFT algorithms and the tendency to separate data to compute one-dimensional and two-dimensional FFTs, this paper presents the following contributions:

(i) A comparative study on the feasibility of the frequently used *divide-and-conquer* strategy found in the literature, proper to the two-dimensional FFT *separability property*, for large-image-sets analysis (in the frequency domain) in Apache Spark clusters, in terms of computational resource usage.

(ii) A comparative analysis on FFT processing times for large image sets using RDD and DataFrames.

In order to get the best possible performance, the highly optimized Fast Fourier Transform in the West software [23], the Java Native Interface, and the Apache Spark middleware are used. The variables used to determine the feasibility of separability-property usage are execution times, CPU and memory usage, and generated network traffic.

2. Methods

This section presents the main methods used in this work, namely, the basic knowledge of two-dimensional DFT and our proposed methodology for the calculation of Apache-Spark-based distributed FFTs.

One of the most exhaustive operations in image processing is the *Fourier Transform*, which is used for frequency analysis and filtering. The following sections describe details about the two-dimensional Fast Fourier Transform and its separability property.

2.1. The Two-dimensional Discrete Fourier Transform. The DFT [4, 24] of an image of $M \times N$ pixels is represented with the following equation:

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi((xu/M)+(yv/N))}, \quad (1)$$

where $e^{-j2\pi((xu/M)+(yv/N))}$ represents the *Twiddle Factors*. With (1), image frequencies can be split into several sinusoidal components, which can be analyzed separately. Frequently, the imaginary part of the Fourier image is used to eliminate specific frequency components, resulting in low pass, high pass, or band pass filters.

Thanks to the DFT *separability property*, (1), which computes the two-dimensional DFT, can be broken down in multiple one-dimensional DFTs [1], which can be expressed as follows:

$$F(u,v) = \frac{1}{M} \sum_{x=0}^{M-1} f(x,v) e^{-j2\pi(xu/M)}, \quad (2)$$

$$F(x,v) = \frac{1}{N} \sum_{y=0}^{N-1} F(x,y) e^{-j2\pi(vy/N)}, \quad (3)$$

where $e^{-j2\pi(xu/M)}$ and $e^{-j2\pi(yv/N)}$ are also twiddle factors. This property helps to express a two-dimensional DFT in several one-dimensional DFTs applied to rows and columns of images.

However, (1)–(3) produce redundant twiddle factors, which can generate terms that can cancel each other out, so they do not contribute to the final result. An alternative to solve this problem is the use of FFT algorithms. In this paper, some of the most important and recent Fast Fourier Transform algorithms are presented.

2.2. The Fast Fourier Transform. Due to the high computational complexity of a one-dimensional DFT [4], researchers have proposed alternatives to reduce the number of operations required for this procedure. These algorithms are known as *fast* DFTs. One of the most popular is the Cooley–Tukey [25, 26] algorithm, which reduces the one-dimensional DFT algorithmic complexity from $O(MN)$ to $O(M \log_2 N)$, eliminating redundant twiddle factors.

Although computational complexity of this algorithm is lower than that of (2) or (3), it has the disadvantage of requiring that M and N be a power of 2 to operate adequately. This makes this algorithm inefficient, because several zero values must be incorporated into the image (zero padding).

Other more efficient but less intuitive FFTs are found in the literature. For instance, the *Prime Factor* [27] FFT reexpresses the N−size DFT as a $N_1 \times N_2$ two-dimensional DFT, where N_1 and N_2 are coprime. In comparison to the Cooley–Tukey FFT, its execution time is greatly reduced.

Currently, the fastest FFT algorithms can be found in the *Fast Fourier Transform in the West* (FFTW) software. It provides a set of FFT algorithms that can be optimized to specific hardware where it will be executed in order to ensure the fastest FFTs, no matter the values of M or N. In several works, its efficiency and feasibility are shown [21]. An evolved FFTW is also presented in [22], also known as FFTW++.

According to the FFTW documentation, this software is a collection of fast algorithms to compute FFTs in the C programming language that can adapt its underlying structure according to the hardware used. With this, it is possible to ensure high performance.

All FFT algorithms in FFTW are meticulously optimized and computed in two stages as follows:

(1) A planner heuristically learns the fastest way to compute FFTs in a machine where it will be executed. After this, the FFTW planner creates a data structure called the *FFTW plan.*

(2) In this stage, the *FFTW plan* transforms an input array. Plans can be reused later as necessary.

As described above, FFTW provides several algorithms to compute FFTs after a plan is scheduled:

(i) Cooley–Tukey algorithm

(ii) Prime factor algorithm

(iii) Rader's algorithm for prime sizes

(iv) Split-radix algorithm (conjugate pair variation)

(v) FFTW Code generator's algorithms

Also, according to FFTW documentation, it can deliver plans with a degree of rigor. For example, if users need to create plans quickly, regardless of whether the execution of such a plan is optimal or not, then the **FFTW_ESTIMATE** flag must be high. But if users need the fastest possible FFTs according to the underlying hardware, then the most appropriated mode is **FFTW_EXHAUSTIVE**.

Another interesting property of FFTW is the *Words of Wisdom*. Every time an FFTW plan is created, it is stored in a region of memory called the *Wisdom*. If users need to save all plans stored in the Wisdom to reuse them later, they have to export them to a string variable or to a local file system. Consequently, previously saved plans can be imported again to its Wisdom to reuse them as needed.

2.3. The Distributed FFT. In order to determine the feasibility of the separability property of two-dimensional FFTs when large amounts of images are distributedly processed, the following methodology is proposed, where it is assumed that the Apache Spark distributed system [15] is selected. Also, we need to select a Spark-compatible Advanced Programming Interface (API) that provides **RDD** and **DataFrame** distributed image representations. In this case, the Spark Image Processing Toolkit (SIPT) [28] is selected to enable distributed image processing for large image sets.

Before explaining details of this methodology, it is necessary to define the distributed data representation for both APIs. For *real* image-set representation, the following structure has been selected:

(i) Spark SQL's DataFrame representation has 4 columns: **Name** which is a **String**; **Height** and **Width**, which are **IntS**; and **Pixels**, which is represented as **Seq[Byte]** type.

(ii) Spark's RDD representation is comprised of an **RDD [(String, Int, Int, Seq[Byte])]** type.

On the other hand, *complex* image-set structure is as follows:

(i) Spark SQL's DataFrame representation has 5 columns: **Name**, which is a **String**; **Height** and **Width** which are **IntS**; and **RealPixels** and **ImaginaryPixels**, which are represented as **Seq[Float]**types.

(ii) Spark's RDD representation is comprised of an **RDD [(String, Int, Int, Seq[Byte], Seq[Byte])]** type.

Moreover, in order to achieve maximum performance while massively computing FFTs on images, we must select an API with highly optimized FFT algorithms such as FFTW; and a large image set, so a conventional computer (e.g., a desktop computer) cannot process it easily. In this case, the very well-known MIRFLICKR-25000 image set, which is comprised of 25,000 JPEG images that were collected from the Flickr® social network, is used.

In order to ensure that Spark applications gain access to FFTW algorithms, the Java Native Interface (JNI) is contemplated, so we have to create the following:

(i) A wrapper class in the Scala language that allows users to invoke native methods (written in the C programming language)

(ii) A dynamic library (**lib.so** file) linked to the previous wrapper class, which contains FFTW functionalities, written in the Scala language

Once the abovementioned requirements are satisfied, we can now proceed to explain in detail the proposed methodology:

(1) Create a Wisdom with 1D/2D-forward/backward plans using the **FFTW_EXHAUSTIVE** rigor. Subsequently, all Wisdom content is exported to each worker's local file system in the root user's directory (**~/sipt.wisdom**). This step must be executed *offline*.

(2) Once the FFTW Wisdom has been created and exported to file, a single executor of every worker should load it once from the path (**~/sipt.wisdom**), either when the Java Virtual Machine (JVM) is initialized or just before using plans. Because of this, plans should be loaded to Wisdom only once per worker. This should not impact the overall performance.

(3) After all plans have been loaded to every worker's Wisdom, every executor is able to find the appropriate plans in the previously created and loaded Wisdom, for both direct (forward) or inverse (backward) FFTs. In this sense, wrapper-class native methods are called directly and concurrently, ensuring that only plans stored in Wisdom are executed. This means that every executor eventually loads and executes different plans and data so that they gain access to the same worker's Wisdom. Once the Spark application finishes, each worker's Wisdom is forgotten. It is important to clarify that, in order to face an FFTW limitation that prohibits concurrent creation and destruction of plans, the **mutex.h** library has been used to force them to create plans from Wisdom sequentially on the same worker node. Because creation and destruction of plans are performed offline, performance should not be impacted.

(4) Finally, every Spark worker can execute two-dimensional FFTs by using the following:

(a) *Separability*: by using the FFT separability property, i.e., (2) and (3), compute one-dimensional FFT plans (see Figure 1(a)) as follows:

(i) Split rows
(ii) Compute one-dimensional FFT plans on rows
(iii) Merge computed rows and store them in memory in a transposed way
(iv) Split columns
(v) Compute one-dimensional FFT plans on columns
(vi) Merge computed columns and store them in memory in a transposed way

(b) *Nonseparability*: compute two-dimensional FFT plans directly (see Figure 1(b)).

(5) Write results to local file system.

3. Results and Discussion

3.1. The High-Performance Computing Hardware. To accomplish this research, a powerful and well-resourced HPC system with the appropriate software platform is needed so that it can massively process large image sets. For the experimentation, we use an HPC cluster system that is comprised of 4 nodes with the following characteristics:

(i) A Dell® T7600 master node, with 12 threads, 16 GB memory, and 1 TB hard disk drive

(ii) Three Dell® T630 worker nodes, each one with 32 threads, 16 GB memory, and 120 GB solid state drive

(iii) A Giga-bit CISCO high-speed local network

(iv) Ubuntu 16.04 Server operating system

(v) Java Development Kit, version 8

(vi) FFTW version 3.3.8

(vii) Apache Spark 2.3.4

(viii) Spark Image Processing Toolkit, version 1.0 [28]

3.2. Experimental Design. In order to be able to corroborate the feasibility of the separability property in distributed systems such as Apache Spark, three experiments are proposed as follows. For comparison purposes, we consider the speed-up and the percentages of CPU, memory, and network usage as performance measures.

(1) The *first* experiment consists of observing how processing times are influenced when computing 8192 FFTs on images from the MIRFLICKR-25000 image set while the number of cores used to process them is gradually increased (1, 3, 6, 12, 24, 48, and 96 cores).

(2) The *second* experiment consists of observing how processing times are affected while the number of images is gradually increased (128, 256, 512, 1024, 2048, 4096, and 8192 images), using only 1 and 96 cores.

(3) The *third* experiment involves extracting some cluster measures from the Ganglia Monitoring System [29], such as CPU, memory, and network usage, while processing two-dimensional FFTs on 128, 256, 512, 1024, 2048, 4096, and 8192 images, consecutively. First, we launch and evaluate the two-dimensional FFT that uses the separability property and then the version that does not use it, considering both RDDs and DataFrame APIs.

It should be clarified that the first two experiments are performed three times and execution times reported here are averaged. Also, execution times associated with inverse (backward) FFTs are not reported in this paper because their execution times are very similar to those obtained with direct (forward) FFTs.

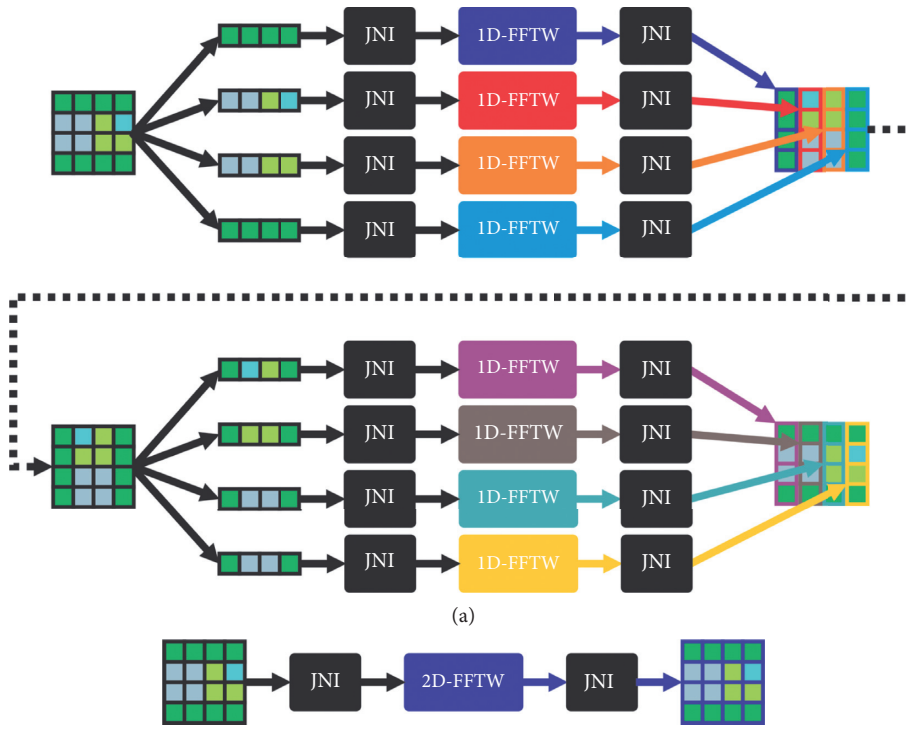


Figure 1: Spark + JNI + FFTW two-dimensional FFT (a) using and (b) not using the separability property.

3.3. Analysis of Results. Given the limited availability of tools that process the FFT for a large number of images, we introduced four versions of distributed FFT (see Section 2.3) in Apache Spark. To achieve this, we use the sequential version of the single precision floating-point (complex to complex) FFTW library [23]. In order to differentiate between each implementation, we will give them a distinctive name in Tables 1–3 as follows:

(1) Using the separability property:

(a) For RDDs: **Sep. (RDD)**.
(b) For DataFrames: **Sep. (DataFrame)**.

(2) Avoiding the separability property:

(a) For RDDs: **Non-sep. (RDD)**.
(b) For DataFrames: **Non-sep. (DataFrame)**.

After carrying out the experimentation, Table 1 presents the processing times comparison results between two-dimensional FFT implementations on 8192 images (taken from the MIRFLICKR-25000 image set), performing a variation in the number of cores, as specified in the *first* experiment described in the previous subsection.

According to Table 1, the best processing times are those obtained with the RDD-based FFT that *does not use* the separability property, as can be seen in all columns of row 4. Note that this happens in all cases, regardless of the number of cores used. As expected, the best result is obtained when using all 96 cores. Notice that the RDD-based FFT is approximately 2x faster than the DataFrame-based FFT when not using the separability property (see 4th and 5th rows) and is approximately 10x faster than when using it (3rd and 4th rows). However, observe that processing times obtained using the separability property are very similar between RDD and DataFrame versions across core variations (2nd and 3rd rows).

Notice in Table 1 that, when using a single core, i.e., when processing is performed sequentially, processing times are very similar to each other. For example, when the separability property is not used, RDD-based FFTs are only 18.8% faster than DataFrame-based FFTs; however, when using the separability property, RDD-based FFTs get processing times that are practically the same as for DataFrame-based FFTs.

Consequently, speed-up results are also presented in Table 2 based on data from Table 1. As can be seen, RDD-based FFTs that do not use the separability property perform computations more efficiently obtaining a speed-up of up to 32x with respect to its sequential version. A general observation is that as the number of cores increases, the processing of almost all FFT versions becomes less efficient. This can be clearly seen in all rows in Table 2 (except in the 4^{th} row), where speed-ups tend to stagnate or degrade, according to Amdahl's Law [30], as the number of cores is increased.

On the other hand, in order to observe the influence on speed-up, throughout the variation of the number of images, Table 3 presents the following results. Observe in row 4 that

Table 1: Processing times results (in seconds) between several 2D FFT implementations, using 1, 3, 6, 12, 24, 48, and 96 cores when processing FFTs on 8192 images from the MIRFLICKR-25000 image set.

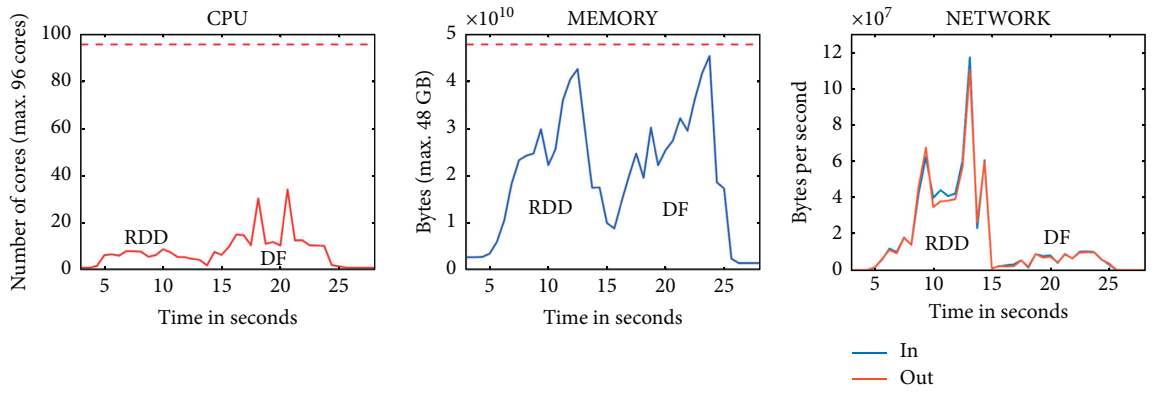
Implementation	96	48	24	12	6	3	1
Sep. (RDD)	320.71	285.88	283.79	420.75	803.02	1460.08	3853.90
Sep. (DataFrame)	318.53	287.94	281.43	412.02	796.41	1467.03	3858.35
Non-sep. (RDD)	31.84	33.28	47.37	80.55	177.63	332.45	1032.62
Non-sep. (DataFrame)	71.85	59.80	79.39	123.94	244.79	478.32	1227.02

Table 2: Speed-up results between several 2D FFT implementations during the core variation of experiment 1.

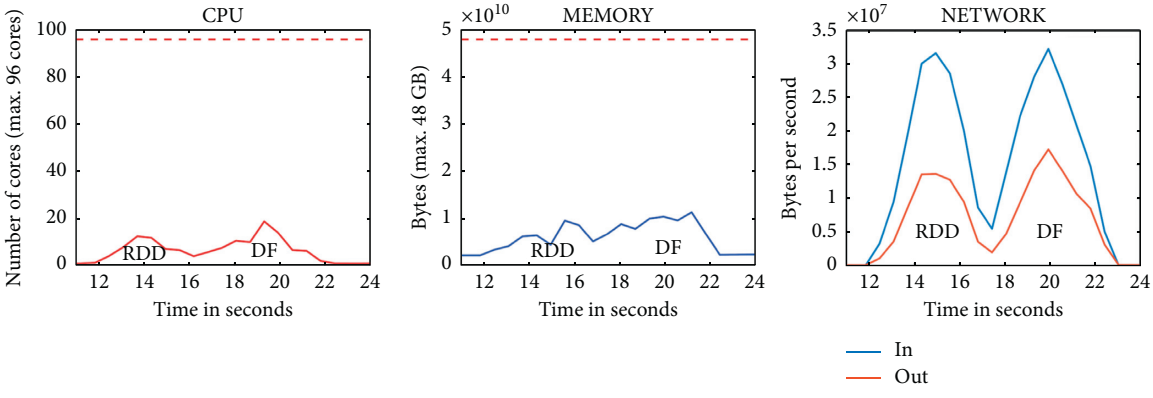
Implementation	3	6	12	24	48	96
Sep. (RDD)	2.64	4.80	9.16	13.58	13.48	12.02
Sep. (DataFrame)	2.63	4.84	9.36	13.71	13.40	12.11
Non-sep. (RDD)	3.11	5.81	12.82	21.80	31.03	32.43
Non-sep. (DataFrame)	2.57	5.01	9.90	15.46	20.52	17.08

Table 3: Speed-up results between several 2D FFT implementations using 1 and 96 cores when processing FFTs on 128, 256, 512, 1024, 2048, 4096, and 8192 images from the MIRFLICKR-25000 image set.

Implementation	128	256	512	1024	2048	4096	8192
Sep. (RDD)	3.09	3.69	4.95	5.60	7.05	7.41	8.44
Sep. (DataFrame)	3.00	3.68	5.56	7.68	9.84	11.57	11.62
Non-sep. (RDD)	3.08	4.02	6.81	11.97	15.80	26.63	30.88
Non-sep. (DataFrame)	2.14	2.67	4.06	7.17	9.86	13.21	17.95



(a)



(b)

Figure 2: CPU, memory, and network usage when processing 2D FFTs on 128, 256, 512, 1024, 2048, 4096, and 8192 images with 96 cores, using both RDD and DF APIs and (a) the separability property and (b) avoiding it.

the best speed-ups obtained are those that were processed by RDD-based FFTs. Note that, regardless of the number of processed images, speed-ups obtained are higher than those obtained by the rest of the implementations. This confirms that RDD-based FFTs that do not use the separability property are the most efficient.

One of the limitations that we faced during this experiment is that versions that use the separability property could not process more than 8192 images. In order to find out the reason for this, the third experiment was carried out, in which the number of images to be processed is consecutively increased until reaching the maximum number possible (ranging from 128 to 8192 images), for both the RDD and DataFrame versions. The obtained results are shown in Figure 2. Observe how both CPU and memory usage increase as the number of images increases over time. According to Figure 2(a), two peaks in memory usage can be seen (when 8192 images are processed), one for the RDD-based version and another for the DataFrame-based version, reaching almost all of the available memory in cluster. In contrast, note in Figure 2(b) that the nonuse of the separability property involves the use of fewer resources. This is the reason why more than 8192 images cannot be processed with the DataFrame-based FFTs that use the separability property.

Although CPU occupancy is not fully exploited, the separability property requires more CPU usage, especially when processing DataFrame-based FFTs. This could be due to the additional processing that a DataFrame requires when inferring data (from a schema) for each one of its records.

Therefore, the use of the separability property directly impacts the two-dimensional FFT performance in Apache Spark due to the network usage because it generates greater incoming and outgoing traffic while *shuffling* data across the network.

After experimentation, several quite interesting findings arise. When processing FFTs on large image sets, using Spark + JNI + FFTW, we found the following:

(i) RDD API is more efficient to process two-dimensional FFTs than DataFrame API, since RDD processing is faster and requires less memory and less computational effort than DataFrame processing.

(ii) When avoiding the separability property while computing RDD-based FFTs, greater speed-ups can be achieved, no matter how many cores are used or how many images are transformed. In addition, the memory usage is lower for this case.

4. Conclusions

This research presented a comparative study for determining the feasibility of the separability property in distributed systems like Apache Spark. Based on the obtained results, we can conclude that the 2D FFT separability property is not feasible in this context. This is because the divide-and-conquer strategy (very useful in several hardware implementations found in the literature), which separates and distributes image information between cluster nodes, generates network traffic that seriously degrades the FFT performance.

In addition, we found that the RDD API is the most appropriate interface for massive FFT processing on large image sets because it requires less memory and less computational effort than DataFrame processing, which can be translated in greater speed-ups.

As future work, a combination of hardware accelerators and Apache Spark is proposed. This could carry out the strategy of divide and conquer within accelerator devices in order to offer the highest possible performance when processing FFTs on large image sets.

Data Availability

We used the public image set MIRFLICKR-25000 which can be found at https://press.liacs.nl/mirflickr/.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

The authors thankfully acknowledge the computer resources, technical expertise, and support provided by the Laboratorio Nacional de Supercómputo del Sureste de México, CONACYT member of the network of national laboratories. This work was partially supported by the Mexican National Council for Science and Technology (Consejo Nacional de Ciencia y Tecnología–CONACYT), under the Catedra program number [1170].

References

[1] N. Shirazi, P. M. Athanas, and A. L. Abbott, "Implementation of a 2-D fast Fourier transform on an FPGA-based custom computing machine," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 282–292, Oxford, UK, September 1995.

[2] S. L. Gonzaga de Oliveira and J. Teixeira de Assis, "A methodology for identification of fingerprints based on Gabor filter," *IEEE Latin America Transactions*, vol. 4, no. 1, pp. 1–6, 2006.

[3] J. Pastore and E. Moler, "Identification of frontal sinus by means of form factors and fourier descriptors," *IEEE Latin America Transactions*, vol. 2, no. 3, pp. 157–161, 2004.

[4] R. C. González and R. E. Woods, *Digital Image Processing*, Prentice-Hall, Hoboken, NJ, USA, 3rd edition, 2007.

[5] T. M. Pytosh and A. M. Magnani, "A new parallel 2-D FFT architecture," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 905–908, Albuquerque, NM, USA, April 1990.

[6] I. Az, S. Sahin, and M. A. Cavuslu, "Implementation of fast fourier and inverse fast fourier transforms in FPGA," in *Proceedings of the 2007 IEEE 15th Signal Processing and Communications Applications*, pp. 1–4, Eskisehir, Turkey, June 2007.

[7] S. A. Ajmal and S. L. Gangadharaiah, "FPGA based area optimized parallel pipelined radix-2^2 feed forward FFT architecture," in *Proceedings of the 2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pp. 1302–1307, Bangalore, India, May 2016.

[8] P. Philipov, V. Lazarov, Z. Zlatev, and M. Ivanova, "A parallel architecture for radix-2 fast fourier transform," in *Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA 06)*, pp. 229–234, Sofia, Bulgaria, October 2006.

[9] I. Sousa Correa, L. Coelho Freitas, A. Klautau, and J. C. Weyl Albuquerque Costa, "VHDL implementation of a flexible and synthesizable FFT processor," *IEEE Latin America Transactions*, vol. 10, no. 1, pp. 1180–1183, 2012.

[10] M. N. Haque and M. S. Udin, "Accelerating fast fourier transformation for image processing using Graphics processing unit," *Journal of Emerging Trends in Computing and Information Sciences*, vol. 2, pp. 367–375, 2011.

[11] A. Gholami, J. Hill, D. Malhotra, and G. Biros, "AccFFT: A library for distributed-memory FFT on CPU and GPU architectures," 2015, https://arxiv.org/pdf/1506.07933.pdf.

[12] "cuFFT, a CUDA Library for Fast Fourier Transformation," 2021, https://developer.nvidia.com/cufft.

[13] A. Kharin, S. Vityazev, V. Vityazev, and N. Dahnoun, "Parallel FFT implementation on TMS320c66x multicore DSP," in *Proceedings of the 2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pp. 46–49, Milan, Italy, September 2014.

[14] H. G. Hen-Geul Yeh, "Parallel implementation of the fast fourier transform on two TMS320C25 digital signal Processors," *IEEE Transactions on Industrial Electronics*, vol. 41, no. 1, pp. 132–135, 1994.

[15] "Spark: Lightning-fast unified analytics engine," 2021, https://spark.apache.org/.

[16] Y. Yan, L. Huang, and L. Yi, "Is Apache Spark scalable to seismic data analytics and computations?" in *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, Santa Clara, CA, USA, November 2015.

[17] "Breeze is a numerical processing library for Scala," 2021, May, https://github.com/scalanlp/breeze.

[18] C. Yang, W. Bao, X. Zhu, J. Wang, and W. Xiao, "A parallel fast fourier transform algorithm for large-scale signal data using Apache Spark in cloud," in *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, vol. 11336, pp. 293–310, Guangzhou, China, December 2018.

[19] L. Dalcin, M. Mortensen, and D. E. Keyes, "Fast parallel multidimensional FFT using advanced MPI," *Journal of Parallel and Distributed Computing*, vol. 128, pp. 137–150, 2019.

[20] 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.

[21] S. G. Johnson and M. Frigo, "A modified split-radix FFT with fewer arithmetic operations," *IEEE Transactions on Signal Processing*, vol. 55, no. 1, pp. 111–119, 2007.

[22] M. Roberts and J. C. Bowman, "Multithreaded implicitly dealiased convolutions," *Journal of Computational Physics*, vol. 1, 2017.

[23] "Fast Fourier Transform in the West," 2021, Aug, https://www.fftw.org/.

[24] A. V. Oppenheim and A. S. Willsky, *Signal and Systems*, Prentice-Hall, Hoboken, NJ, USA, 2nd edition, 1996.

[25] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, p. 297, 1965.

[26] R. E. Blahut, *Fast Algorithms for Signal Processing*, Cambridge University Press, , NY, USA, 1st edition, 2010.

[27] F. Pérez and T. Takaoka, "A prime factor FFT algorithm implementation using a program generation technique," *IEEE Transactions on Acoustics, Speech, & Signal Processing*, vol. 35, no. 8, pp. 1221–1223, 1987.

[28] A. Téllez-Velázquez and R. Cruz-Barbosa, "A Spark image processing toolkit," *Concurrency and Computation–Pract E*, vol. 31, no. 17, 2019.

[29] "Ganglia Monitoring System," 2021, May, http://ganglia.info/?page_id=66.

[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Elsevier - Morgan Kaufmann, Waltham, MA, USA, 5th edition, 2012.