Una implementación paralela de las Transformadas DCT y DST en GPU. Análisis de performance

Erica Montes de Oca[1], Marcelo Naiouf[1], Laura De Giusti[1], Franco Chichizola[1], Javier Giacomantone[1], Armando De Giusti[1,2]

[1]Instituto de Investigación en Informática LIDI (III-LIDI)
Facultad de Informática – Universidad Nacional de La Plata
La Plata, Buenos Aires, Argentina
[2] CONICET
{emontesdeoca,mnaiouf,ldgiusti,francoch,jog,degiusti}@lidi.info.unlp.edu.ar

Resumen. Se presenta la implementación paralela de las Transformadas Coseno Discreta y Seno Discreta en GPU, utilizadas en procesamiento de señales. Se analiza la performance, y el grado de aceleración obtenido en relación a la solución secuencial. A modo de referencia, se compara con la paralelización en un cluster de multicores. Como herramientas de programación se utilizaron CUDA en GPU y MPI para la versión en cluster de multicore. Los resultados experimentales muestran una buena respuesta de las soluciones en GPU en relación a las restantes alternativas.

Palabras clave: procesamiento paralelo, GPU, CUDA, procesamiento de señales, DCT, DST.

1 Introducción

La complejidad creciente de los problemas, sumado a la alta demanda de cómputo y el aumento de la cantidad de datos ha impulsado el surgimiento de diversas arquitecturas multiprocesador. Pensar en reducir los tiempos de ejecución de las aplicaciones con alta demanda computacional se traduce en procesamiento paralelo sobre dichas arquitecturas debido a que las soluciones secuenciales existentes para tales problemas producirían respuestas en la mayoría de los casos no aceptables en tiempo.

La variedad de arquitecturas multiprocesador tales como clusters, multicores, clusters de multicore [1][2][3] y multiclusters [4] permiten la elección de una diversidad de plataformas que ofrecen la posibilidad de obtener soluciones en menor tiempo si se explota el potencial de las mismas. En los últimos años, las Unidades de Procesamiento Gráfico (GPUs) han surgido como una alternativa para el procesamiento de alto desempeño, debido a su gran poder de cómputo y a su costo accesible por tratarse de una arquitectura masivamente comercializada. A partir del 2003, surgió lo que se denomina programación de Propósito General en GPU (GPGPU), que abrió las puertas a la implementación de un conjunto de aplicaciones en distintas áreas como la medicina, las finanzas, el análisis de tráfico aéreo, la física,

la química, el procesamiento de señales, criptografía simétrica, entre otras, que se han visto beneficiadas por el poder de cómputo de las mismas [5][6][7][8].

El concepto de señal se aplica a una variedad de campos tales como las comunicaciones, la aeronáutica y la astronáutica, el diseño de circuitos, el reconocimiento de patrones, la acústica, la óptica, la sismología, entre otros. Una señal es un conjunto de valores que representan el comportamiento de un fenómeno [9]. El procesamiento de señales comprende el procesamiento, amplificación e interpretación de señales, las cuales son originadas por diversas fuentes [10]

Las señales del mundo real tales como el sonido o los espectros de luz son conjunto de números continuos infinitos en el tiempo y el espacio que no son representables directamente en una computadora. Utilizando como base el teorema del muestreo se obtiene un conjunto de valores discretos en un periodo de la señal que permite la digitalización de la misma.

Las trasformadas convierten secuencias del dominio del tiempo y del espacio al dominio de la frecuencia, donde el procesamiento suele ser mucho más eficiente. En estos problemas, el cómputo comprende el cálculo de la transformada por cada elemento de la secuencia con el resto de los elementos que la forman, por lo que los algoritmos resultantes requieren un tiempo considerable [11]. Una manera de reducir el tiempo de cómputo es utilizando procesamiento paralelo explotando las arquitecturas actuales.

El aporte del presente trabajo consiste en mejorar el desempeño del cálculo en las transformadas utilizadas en el procesamiento de señales (en particular las transformadas Seno y Coseno) a través del uso de la GPU. En la Sección 2 se ofrece una descripción de la plataforma CUDA (Compute Unified Device Architecture); la Sección 3 presenta las características relevantes de las Transformadas DCT y DST, dos algoritmos de procesamiento de señales en el dominio transformado. En la Sección 4 se brinda un análisis de performance de las soluciones en GPU. Finalmente, se presentan las conclusiones de este trabajo.

2 GPU-CUDA

Las Unidades de Procesamiento Gráfico (GPUs) han emergido en los últimos años como una plataforma alternativa para el procesamiento paralelo a costo accesible. Su poder de cómputo se ha incrementado gracias a la industria de los videojuegos, que ha producido e impulsado el incremento del rendimiento en cada nueva generación provocando el crecimiento cúbico de la cantidad de procesadores. La performance que se puede obtener se comenzó a explotar para la programación de propósito general a partir de 2003 [7]. El gran interés en las mismas por parte de académicos e industriales permitió el avance de la GPU en otros campos fuera del procesamiento de imágenes.

Ante el creciente interés, Nvidia llevó a cabo la implementación de CUDA y su posterior lanzamiento en 2007. CUDA es una plataforma hardware y software que extiende al lenguaje C con un pequeño conjunto de abstracciones que permiten la programación de propósito general en GPU [12]. La simplicidad que ofrece CUDA al

programador le permite concentrarse en la paralelización del problema y no en el aprendizaje de un lenguaje paralelo complejo.

El flujo de ejecución de un programa CUDA comprende la reserva de memoria en el dispositivo o *device* (GPU), la transferencia de los datos desde el host o CPU al device, la ejecución de un kernel o función, y finalmente la transferencia de resultados al host. Un kernel se invoca desde el host y se ejecuta en la GPU por cada thread, por lo que la arquitectura se la describe como STMD (Simple Thread, Multiple Data).

CUDA define una estructura de organización para los threads que ejecutará: la grilla de bloques de threads que se asigna a una GPU, los bloques que son mapeados a los distintos multiprocesadores, y los threads que (de a 32) son planificados en conjuntos de warps (unidad de planificación de threads) asignados a los procesadores que conforman un multiprocesador [13][14].

La GPU (Fig. 1) está compuesta por un conjunto de SMs (Multiprocesadores de Streams) los cuales se componen de un número de SPs (Procesadores de Streams) que varia de una arquitectura a otras [7][12]. La ociosidad no está permitida por lo que los threads se planifican y se ejecutan para ocultar las latencias de acceso a las memorias.

La jerarquía de memoria en la GPU está compuesta por memorias con características diferentes. La comunicación entre threads se realiza a través de la memoria global, ubicada fuera del chip y compartida por los threads de todos los bloques de la grilla. Esta memoria es la más abundante y es posible realizar operaciones de lectura/escritura sobre ella desde el host o el dispositivo. Los accesos a la misma dependen del tipo de dato utilizado (2 o 4 ciclos). Una de las memorias más utilizadas es la compartida o *shared*, la cual, se comparte sólo por los threads de un mismo bloque. El acceso a la misma por estar dentro del chip es mucho menos costoso que el de la memoria global, sin embargo, el espacio de direcciones disponible es muy reducido (16 KB en placas de capacidad 1.x, 48 KB en placas 2.x), y define la cantidad de bloques que podrán ser asignados a un SM. Además, los threads cuentan con una memoria de registro cuyo acceso es constante pero muy limitado, lo que restringe la cantidad de threads por bloques que se pueden definir si se desea obtener la mayor performance posible [8][13][14][15][16].

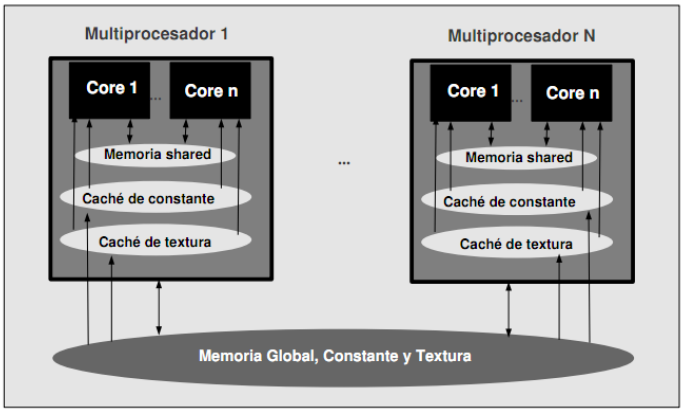

Fig. 1. Jerarquía de memoria de la GPU

3 Procesamiento de señales en el dominio transformado

En esta Sección se realiza una breve descripción de los algoritmos utilizados en el trabajo (DCT y DST).

3.1 DCT

La Transformada Discreta Coseno (DCT) [17] es una de las más utilizadas en los métodos del procesamiento de señales, luego de la Transformada de Fourier. Tiene la característica de ser lineal, ortogonal e invertible en el dominio R^N al codominio R^N. Esto la diferencia de la Transformada de Fourier Discreta (DFT) con dominio R^N y codominio C^N, lo que significa que el dominio transformado comprende el tratamiento de números complejos.

Por su buena capacidad de compactación de la energía en el dominio transformado concentrando la mayor parte de la información en pocos coeficientes transformados, es muy utilizada en algoritmos de compresión. La transformación que se obtiene es independiente de los datos, por lo que el algoritmo aplicado no varía en comparación con otros. Además, permite interpretar los coeficientes en el dominio transformado lo con un gran aprovechamiento de su capacidad de compresión [18].

La Ecuación 1 representa a la DCT:

$$C(u) = \alpha[u] * \sum_{x=0}^{N-1} \frac{\cos(\pi * (2x+1) * u)}{2N} * f(x) \qquad \alpha[u] = \begin{cases} \sqrt{\frac{1}{N}} & u = 0 \\ \sqrt{\frac{2}{N}} & u = 1, 2, \dots, N-1 \end{cases} \quad (1)$$

La DCT parte de la DFT, y puede decirse que es un caso especial para datos con duración finita [18]. Mientras que la DFT descompone las señales en senos y cosenos, la DCT lo hace como la suma de cosenos [10].

3.2 DST

La Transformada Seno Discreta (DST) al igual que la DCT es muy utilizada para algoritmos de compresión [13]. Sin embargo, la DCT presenta mejores propiedades de compactación, ya que la DST suele concentrar la energía media en los coeficientes mayores lo que causa una perdida de eficiencia. A pesar de ello, suaviza señales periódicas formadas por repeticiones discontinuas, y tiene un muy buen comportamiento frente al ruido, comparada con otras transformadas [19][20].

La DST (representada por la Ecuación 2) comparte las mismas características de la DCT, tales como linealidad, ortogonalidad, invertible en el dominio real.

$$S(u) = \frac{\sqrt{2}}{N+1} * \sum_{x=0}^{N-1} \frac{sen\big((x+1) * (u+1) * \pi\big)}{N+1} * f(x) \qquad u = 0, 1, 2, \dots, N-1 \quad (2)$$

4 Descripción de las soluciones

4.1 Soluciones secuenciales

El procesamiento de la señal de entrada se realiza por cada uno de los valores que comprenden dicha señal. Por cada elemento, se calcula la transformada involucrando a todos los demás valores de la secuencia. El código se muestra en la Fig. 2.

```
function void DCT(float *entrada, float *dct){
  int i, j;
  float accum;

  for(i=0;i<N;i++){
    accum = 0.0;
    for(j=0;j<N;j++){
      accum += (cos(PI*i*(2*j+1))/(2.0*N))*entrada[j];
    }
    accum *= (2.0/sqrt((float)N));
    if(i == 0){accum *= 0.707106781};
    dct[i] = accum;
  }
}
```

```
function void DST(float *entrada, float *dst){
  int i, j;
  float accum;

  for(i=0;i<N;i++){
    accum = 0.0;
    for(j=0;j<N;j++){
      accum += entrada[j] *(sin(PI*(i+1)*(j+1))/(N+1));
  }
    accum *= sqrt(2.0/(float)N+1);
    dst[i] = accum;
  }
}
```

Fig. 2. Funciones que realizan la DCT y la DST secuencial 1D

La ejecución de ambas transformadas se realiza de manera lineal sobre el arreglo de señales. Por cada señal se realiza el cálculo de la misma con las N señales que conforman la secuencia. Por lo cual, la complejidad del algoritmo es $O(N^2)$.

4.2 Soluciones GPU-CUDA

La CPU o host debe realizar la reserva de memoria en la GPU de dos vectores para N elementos. Uno de ellos contendrá la secuencia de entrada mientras que el otro el resultado de aplicar la DCT o la DST sobre dicha secuencia. Las señales a procesar son enviadas a la GPU a través de PCI-e. El gran cuello de botella de la GPU se encuentra en la comunicación entre la misma y el host. El esfuerzo de la implementación debido a las consideraciones de bajo nivel referentes a la arquitectura sólo se justifica en aplicaciones de complejidad $O(N^2)$ o superior.

La invocación de la función *kernel* transfiere el control de ejecución a la GPU [12][13], mientras que la CPU queda en espera por los resultados. La capacidad de cómputo masivo que brinda la GPU hace posible la definición de un thread por cada elemento de la secuencia de entrada. Sin embargo, el algoritmo tiene una sección de código que debe realizarse inevitablemente de manera secuencial: comprende el cálculo de la transformada de un elemento con el resto de las *N* señales que componen la secuencia.

Las operaciones más costosas en la GPU son los accesos a la memoria global [14], por lo que la reducción de los mismos está relacionada directamente a la performance alcanzable con dicha arquitectura. Para optimizar el acceso a memoria en el cálculo de la transformada se realiza un procesamiento por porciones. La entrada compuesta

por *N* señales se divide en *b* porciones, siendo el tamaño de *b* igual a la cantidad de bloques de la grilla. Cada porción está comprendida por *t* señales que coincide con la dimensión de los bloques de threads definidos en la GPU. Cada elemento de *b* es cargado a la memoria compartida por cada thread que compone el bloque.

Antes de comenzar el procesamiento, los threads deben sincronizar para garantizar que todos los elementos necesarios para calcular la transformada estén en la memoria shared. Una vez que se ha realizado la ejecución de la transformada con todos los elementos de *b*, debe cargarse una nueva porción del vector de señales de entrada. Todos los bloques de threads que componen la grilla realizan exactamente la misma operación [12][13][15]. Se requiere una segunda sincronización antes de volver a cargar una nueva porción de datos para garantizar el acceso de todos los threads del bloque a posiciones continuas de la memoria global, consiguiendo de esta manera *t* datos en un acceso.

Cuando todos los threads han calculado la DCT o la DST para las *N* señales que componían la entrada, el control vuelve a la CPU enviándole los datos calculados por PCI-e.

5 Resultados obtenidos

5.1 Entorno de prueba

Las pruebas realizadas para los algoritmos secuenciales en CPU se realizaron en un procesador Intel Xeón e5405. Los algoritmos implementados en la GPU se ejecutaron sobre una placa Geforce TX 560Ti con 384 procesadores, con una cantidad máxima de thread de 768 y 1 GB de memoria RAM.

5.2 Resultados experimentales

Los resultados se obtuvieron a partir de un promedio de las ejecuciones realizadas. La Tabla 1 presenta los tiempos de ejecución para los algoritmos secuenciales. La Tabla 2 muestra los tiempos con GPU-CUDA, donde se utilizaron 256 threads por bloque que es la cantidad óptima para la arquitectura utilizada. La Fig. 3 presenta dichos resultados gráficamente.

Tabla 1. Tiempos de ejecución (en segundos) para los algoritmos secuenciales en CPU.

Tamaño del vector de señales	Secuencial - DCT	Secuencial – DST
65536	351,91	373,59
131072	1409,60	1507,67
262144	5638,53	6048,87
524288	22513,35	24218,41
1048576	90061,53	96929,61

Tabla 2. Tiempos de ejecución (en segundos) para los algoritmos paralelos en CUDA.

Tamaño del vector de señales	GPU - DCT	GPU- DST
65536	0,60	0,80
131072	2,38	2,21
262144	9,44	8,16
524288	37,34	32,60
1048576	147,26	130,29

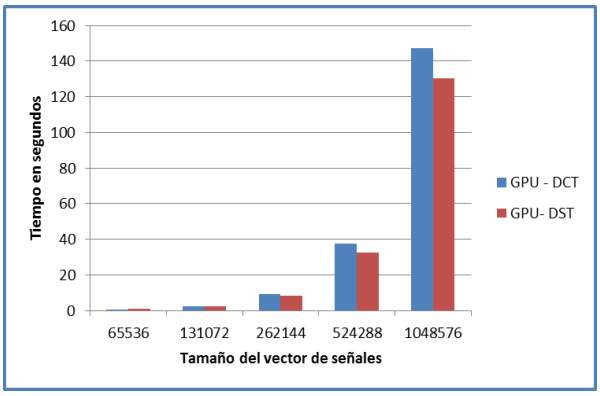

Fig. 3. Tiempo de ejecución en segundos para los algoritmos DCT y DST en GPU, con 256 threads.

Por tratarse de dos arquitecturas diferentes, las métricas utilizadas para los algoritmos paralelos en CPU no son posibles de ser utilizados para comparar performance obtenida con la GPU. Por lo tanto, suele referirse a la ganancia obtenida por el uso de GPU a una aceleración alcanzada, la misma se calcula (de manera similar que el *Speedup*) como el tiempo secuencial (en la CPU) respecto del tiempo paralelo (en la GPU). Esto se muestra en la Fig. 4.

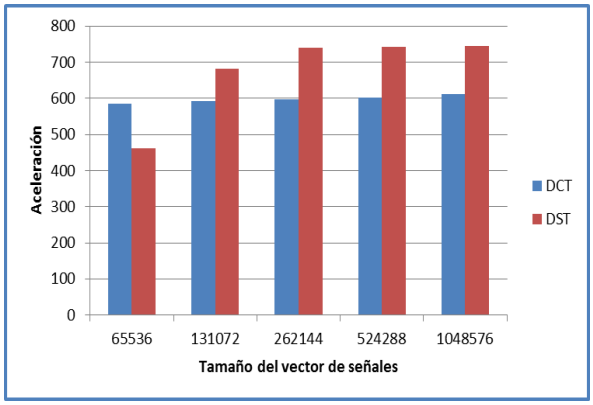

Fig. 4. Aceleración obtenida para los algoritmos DCT y DST en GPU.

Los resultados presentados en la Fig. 4. reflejan la ganancia obtenida en el uso de la GPU en la resolución de las transformadas DCT y DST. Puede observarse que la aceleración obtenida es casi lineal.

Por otra parte, se implementaron las versiones paralelas en MPI con el objetivo de realizar una comparación con la performance obtenida en GPU-CUDA. Las pruebas presentadas para dichas soluciones fueron realizadas en un Blade de 16 servidores (hojas). Cada hoja posee 2 procesadores Quad Core Intel Xeón e5405 de 2,0 GHz, con cachés L2 2x6MB compartida de a par de procesadores, y sistema operativo Fedora 12 de 64bits

Los algoritmos paralelos en MPI para ambas soluciones realizan las siguientes acciones: el vector de señales es dividido en tantas partes como procesos MPI se quiera ejecutar. Cada proceso realizará el cálculo de la transformada para la porción de señales que le corresponde (tal como se detalló en 4.1.).

Las pruebas experimentales se realizaron para una cantidad de 8 y 16 procesos. En el caso de 8 procesos, se ejecutaron en una misma hoja. Para 16 procesos, se utilizaron dos hojas teniendo 8 procesos en cada una. Los resultados se presentan en la Tabla 3.

Tabla 3. Tiempos de ejecución (en segundos) para los algoritmos paralelos en MPI.

Tamaño del vector de señales	MPI (P = 8) DCT	MPI (P = 16) DCT	MPI (P = 8) DST	MPI (P = 16) DST
65536	44,72	26,95	47,61	28,09
131072	178,40	106,61	191,67	109,63
262144	713,62	395,18	772,39	442,73
524288	2854,88	1721,37	3093,45	1842,89
1048576	11435,14	7265,19	14235,23	7204,74

La Fig. 5 presenta las comparaciones de los tiempos de ejecución de los algoritmos DCT y DST en sus versiones paralelas en MPI y GPU.

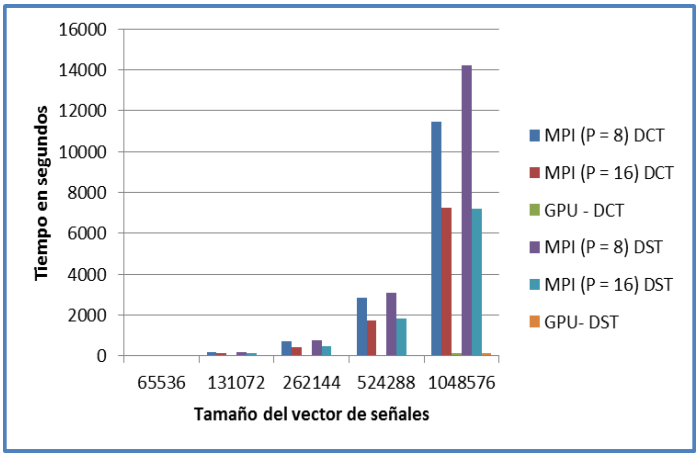

Fig. 5. Comparación de tiempos de ejecución del algoritmo DCT y DST entre MPI y CUDA.

6 Conclusiones y trabajos futuros

La complejidad de los algoritmos secuenciales de tratamiento de señales en el dominio transformado demanda una cantidad de tiempo inaceptable siendo de este modo el paralelismo una solución viable para la reducción de los tiempos de procesamiento de las transformadas en el dominio de la frecuencia.

Por sus características, el procesamiento de las transformadas posibilita la ejecución de las mismas en la arquitectura GPU obteniendo una considerable aceleración con respecto a las versiones secuenciales y paralelas en MPI sobre un Blade (además de un costo monetario considerablemente inferior).

Las líneas de trabajo futuras incluyen la utilización de clusters de GPUs y el estudio del paradigma de programación híbrida utilizando MPI-CUDA para dicha arquitectura. Asimismo, el análisis de consumo energético para estos algoritmos sobre GPUs.

Referencias

1. Thiruvathukal George K., " Cluster Computing ", Copublicado por la IEEE CS y la AIP (2005)
2. Tinetti Fernando G. y Wolfmann Gustavo, "Parallelization Analysis on Clusters of Multicore Nodes Using Shared and Distributed Memory Parallel Computing Models", World Congress on Computer Science and Information Engineering (2009)
3. F. Leibovich, S. Gallo, L. De Giusti, F. Chichizola, M. Naiouf, A. De Giusti., "Comparación de paradigmas de programación paralela en cluster de multicores: Pasaje de mensajes e híbrido. Un caso de estudio.", Proceedings XVII Congreso Argentino de Ciencias de la Computación (CACIC 2011).Págs. 241-250. ISBN 978-950-34-0756-1 (2011)
4. Yun Zhifeng, Lei Zhou , Allen Gabrielle, Katz Daniel S., Jha Shantenu y Ramanujam Jagannathan, "An Innovative Application Execution Toolkit for Multicluster Grids", publicado por IEEE. (2009)
5. Pousa Adrian, Sanz Victoria, De Giusti Armando, "Análisis de rendimiento de un algoritmo de criptografía simétrica sobre arquitecturas multicore", Instituto de Investigación en Informática – LIDI, XVII Congreso Argentino de Ciencias de la Computación, págs. 231-240. (2011)
6. Nvidia Corporation, arg.nvidia.com/object/cuda_home_new_la.html
7. Kirk David B., Hwu Wen-mei W., "Programming Massively Parallel Processors: A Hands-on Approach", Morgan Kaufmann (2010)
8. Piccoli María Fabiana, "Computación de Alto Desempeño en GPU", XV Escuela Internacional de Informática del XVII Congreso Argentino de Ciencia de la Computación, Editorial de la Universidad Nacional de La Plata. (2011)
9. Oppenheim A. V., Schafer R. W, "Discrete-Time signal processing". Prentice Hall (1989)
10. Ireneo Peral Alonso, "Ecuaciones en Derivadas Parciales", Universidad Autónoma de Madrid (2005)
11. Posadas Yague J. L., Benet Gilabert, "Transformada Rápida de Fourier (FFT) e Interpolación en Tiempo Real. Algoritmos y aplicaciones". Universidad Politécnica de Valencia . (1998)
12. Nvidia Corporation, "NVIDIA CUDA C Programming Guide ". (2011)
13. Nvidia Corporation, "CUDA C BEST PRACTICES GUIDE ". (2012)

14. Perez Cristian y Piccoli M. Fabiana, “Estimación de los parámetros de rendimiento de una GPU”, Mecánica Computacional Vol XXIX, págs. 3155-3167. (2010)
15. Nvidia Corporation, “GPU gems”, Pearson Education. (2003)
16. Nickolls John, Dally William J., “The GPU Computing Era”, publicado por IEEE (2010)
17. Crane, “Simplified Approach to Image Proccesing: Classical and Modern Techniques in C”. Prentice Hall (1997)
18. Hernández Cruz Néstor, Espinosa Flores-Verdad Guillermo, “Implementación de la Transformada Discreta Coseno”. IEEE, INAOE. (2002)
19. Albino Luciano Nuta da Costa, Mario Wilson Moraes Pinheiro Junior e Roldão Sereni Neto, Johelden Campos Bezerra, Antônio Marcos de Lima Araújo, “Algoritmo Rápido para Implementação da DSTr”, Instituto de Estudos Superiores da Amazônia, Avenida Governador José Malcher 1145 (2005)
20. Juliano B. Lima, Ricardo M. Campello de Souza, “Transformadas Trigonométricas Discretas: Convoluções e Polinômios de Chebyshev”, Depto de Eletrônica e Sistemas, UFPE, C.P. 7800, Recife, PE (2009)