

A GPU Accelerated DCT Implementation for Image Compression

Angelamaria Cardone[1(✉)] and Gerardo Di Pascale[2]

[1] Department of Mathematics, University of Salerno, Via Giovanni Paolo II n. 132, 84084 Fisciano, Italy
ancardone@unisa.it

[2] Department of Computer Science, University of Salerno, Via Giovanni Paolo II n. 132, 84084 Fisciano, Italy
https://docenti.unisa.it/angelamaria.cardone

Abstract. The Discrete Cosine Transform (DCT) is a cornerstone of the JPEG standard, nevertheless its direct implementation entails significant computational complexity for full-image processing, due to intensive matrix operations. Building upon the methodology proposed by Haweel *et al.* in 2016, which utilizes a specific transformation matrix to streamline block-based DCT through matrix multiplication, this work proposes an optimized parallel version designed for GPUs. This implementation leverages advanced strategies, including memory coalescing, the reduction of thread divergence, and the efficient management of GPU memory hierarchies. Experimental results demonstrate that the optimized algorithm achieves a remarkable speed-up of 340× compared to traditional CPU-based approaches. Furthermore, the implementation significantly outperforms some existing parallel solutions for GPU, reducing execution time by up to 31% compared to an efficient CUDA-based algorithm and by over 96% compared to the standard cuBLAS library. These performance gains are especially pronounced when processing high-resolution images, highlighting the scalability and computational efficiency of the proposed approach for large-scale visual data.

Keywords: image compression · DCT · GPU computing

1 Introduction

Image compression is one of the keys to achieve speed and efficiency of many applications in the digital world. Transmitting compressed images means "transmit less" and thus be faster. Archiving compressed images means "archive more", hence efficient use of storage. DCT is widely used in the JPEG image compression standard. However, its direct computation has high computational complexity, as it is based on multiplications between very large matrices. The computation of a two-dimensional DCT, using standard procedures, requires a computational cost of $O(N^3)$ for the entire image.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026 Workshops, LNCS 16763, pp. 184–198, 2027.
https://doi.org/10.1007/978-3-032-30530-5_12

In the last decades, GPU computing has emerged as a powerful and highly versatile approach for tackling numerical problems of scientific calculus, which require a huge computational effort. Within this class, there are problems involving large data sets and/or characterized by considerable computational complexity. Just to make some examples: the solution of differential problems of high dimension [3–5,8,11]; images elaboration, as image denoising [6,7], image compression [9,10,16,17] and other issues related to images [13,14].

This paper deals with the study, development and optimization of an advanced image compression algorithm based on the Discrete Cosine Transform (DCT) and their optimized parallel implementation for GPUs. The main goal is to improve computational efficiency and reduce processing time by leveraging the massive parallelism inherent in the CUDA architecture.

In the recent literature, several approaches have been proposed so far for an efficient image compression based on DCT. Most of them rely on accurate and cheap approximations of the transformation matrix [1,2,10] and also suitable for parallel implementations [9]. Following the methodology introduced by Haweel *et al.* in [9], the image is divided into blocks of fixed size and then the approximated DCT of each block is computed, which requires matrix multiplications involving the image blocks, the approximated transformation matrix and its transpose. The considered algorithm has been parallelized and optimized for the implementation on the GPU architecture.

The article is organized as follows. Section 2 illustrates the image compression method based on DCT, and briefly describes some variants of this methods. In Sect. 3 we explain the GPU implementation we propose, with the optimization strategies we adopt to increase efficiency. Section 4 includes some numerical experiments carried out on significant test examples from the specialized literature and measure the accuracy and efficiency of our algorithm, also by comparing it with other existing algorithms. Finally, some concluding remarks are reported in Sect. 5.

2 Image Compression by DCT

Established in 1992, JPEG was the first international image compression standard based on the Discrete Cosine Transform (DCT) [12]. The complete encoding and decoding process involves several stages, as illustrated in Fig. 1.

JPEG compression operates on 8×8 pixel blocks, applying a sequence of transformations to each block independently, to achieve an efficient data representation. **The Forward DCT** (FDCT) step transforms image blocks into the frequency domain, obtaining the coefficient matrix. This transformation itself is lossless. Then, each of the 64 coefficients undergoes a uniform quantization using a carefully designed 64-element quantization matrix Q. The process consists of the integer division with rounding of the coefficients by the quantization matrix. Quantization introduces loss by discarding perceptually less significant details.

The two dimensional DCT of a matrix x of dimension $N \times N$, is the matrix

$$X = T_{DCT}^T x T_{DCT},$$

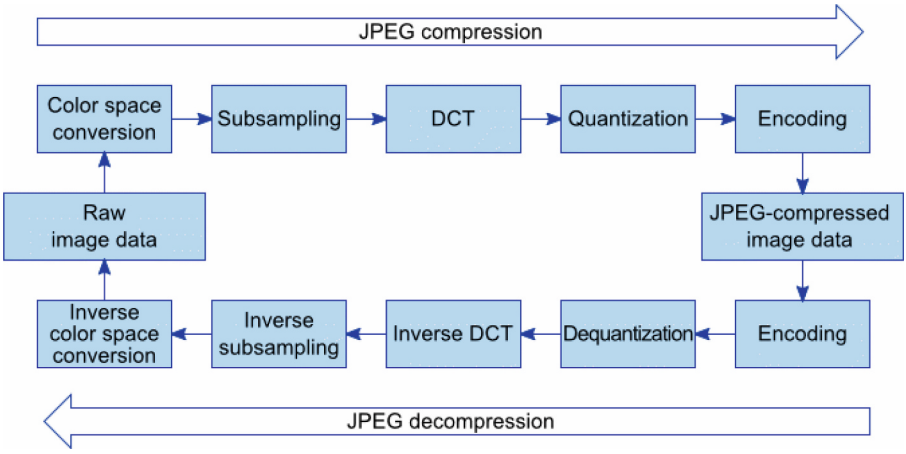

Fig. 1. JPEG Encoding Process scheme.

where A^T denotes the transpose of matrix A, and

$$T_{DCT}(u,v) = \alpha(u)\alpha(v) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \cos\left[\frac{\pi(2i+1)u}{2N}\right] \cos\left[\frac{\pi(2j+1)v}{2N}\right],$$

$$\alpha(u) = \begin{cases} \frac{1}{\sqrt{N}}, & \text{if } u = 0, \\ \frac{2}{\sqrt{N}}, & \text{otherwise} \end{cases}$$

for $u, v = 1, \ldots, N$. We observe that the transformation matrix T_{DCT} is orthogonal, i.e. $T_{DCT}^{-1} = T_{DCT}^{T}$.

In 8-bit precision images, pixel intensities range from 0 to 255. Prior to the DCT transformation, a level shift is performed by subtracting 128 from each pixel value. This centers the dynamic range around zero, shifting it to $[-128, 127]$, which is the standard input format for DCT-based compression like JPEG.

Then, the DCT transformation is applied and a matrix X is obtained. Given a certain quantization matrix Q, each value of X is replaced by the nearest integer of the fraction

$$\frac{X_{ij}}{Q_{ij}}.$$

The inverse process is applied for the decompression of the image.

A number of studies on the DCT-based compression methods tried to identify a transformation matrix T, which accurately approximates the matrix T_{DCT} and reduces the computational effort of the overall coding and encoding process. The goal is to balance accuracy and performance, ensuring that the approximated transformation retains the essential properties of the original DCT while reducing complexity for practical applications. A fundamental property to maintain is orthogonality, which significantly streamlines the decompression process.

Under this assumption, the reconstruction $X = T^T CT$ follows directly from the forward transform $C = TXT^T$, without requiring explicit matrix inversion.

One of the first approximations was proposed by Haweel in [10], where the sign function is applied to the T_{DCT} matrix:

$$T_{SDCT}(u,v) = \frac{1}{\sqrt{N}} \operatorname{sign}\{T_{DCT}(u,v)\}$$

with the signum function operator sign defined as:

$$\operatorname{sign}\{x\} = \begin{cases} +1 & \text{if } x > 0, \\ 0 & \text{if } x = 0, \\ -1 & \text{if } x < 0. \end{cases}$$

Nevertheless, this approach does not preserve orthogonality. Later, Bouguezel *et al.* in [2] improved this approximation by introducing the orthogonal matrix T shown in the next section.

3 Proposed GPU Algorithm for Image Compression by DCT

This section illustrates first the mathematical formulation of the DCT-based image compression method we consider. Then, we describe in detail the strategies we introduce for the GPU implementation, in order to improve efficiency.

3.1 Mathematical Formulation

Here, we explain the mathematical foundation underlying our algorithm, which exploit the matrix formulation of the DCT. Specifically, it is possible to express the DCT of an image block X, of size 8×8, in the form $T \cdot X \cdot T^T$, where T is a transformation matrix specially designed in [2]. This formulation facilitates an efficient implementation on GPUs, thanks to the inherent parallelism of matrix operations.

The transformation matrix used is given by:

$$T = \hat{D}\hat{T}, \tag{1}$$

with

$$\hat{T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 \\ 1 & 0.5 & -0.5 & -1 & -1 & -0.5 & 0.5 & 1 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0.5 & -1 & 1 & -0.5 & -0.5 & 1 & -1 & 0.5 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\hat{D} = \operatorname{diag}\left(\frac{1}{2\sqrt{2}}, \frac{1}{2}, \frac{1}{\sqrt{5}}, \frac{1}{\sqrt{2}}, \frac{1}{2\sqrt{2}}, \frac{1}{2}, \frac{1}{\sqrt{5}}, \frac{1}{\sqrt{2}}\right)$$

We highlight that the matrix T satisfies the orthogonality property, i.e., $T^{-1} = T^T$. This property allows the same matrix to be used for both encoding and decoding. For an image block X of size 8×8, the transformation is applied as follows:

$$C = TXT^T = \hat{D}\left(\hat{T}X\hat{T}^T\right)\hat{D}^T,$$

and the reconstruction is given by:

$$X = T^T CT = \hat{T}^T\left(\hat{D}^T C\hat{D}\right)\hat{T}.$$

3.2 Algorithm Overview

The key stages of a generic compression algorithm that follow JPEG standard include:

1. **Sub-sampling:** Each element of the image matrix is subtracted by a scalar of 128.
2. **DCT Application:** The DCT is performed on each block X_b of size 8×8 of the image matrix X, obtaining the coefficients matrix C.
3. **Quantization:** The resulting coefficients are quantized using a predefined matrix Q.

Conversely, the decompression process reverses these operations:

4. **De-quantization:** The quantized coefficients are de-quantized.
5. **Inverse DCT:** The inverse transformation is applied for each block C_b of the coefficient matrix C.
6. **Up-sampling:** Pixel values are restored by adding 128 to the matrix entries.

3.3 Proposed GPU Implementation

In this section, we present the **HpApprDCT** algorithm and its implementation. Developed from scratch, this method is designed to be the fastest approach while incorporating an extended optimization phase inspired by earlier work [9].

The CUDA programming model follows a host-device architecture, where the CPU acts as the host and the GPU as the device. On the host side, data are first acquired and processed sequentially before being transferred to the device's global memory for massively parallel execution. Parallel operations on the GPU are executed through kernels, which are C functions running concurrently on multiple CUDA threads. These threads are organized into blocks, which can be organized into grids in up to three dimensions, with all threads executing the same kernel as shown in Fig. 2.

The HpApprDCT is applied to images by dividing the image into 8×8 blocks. Image block processing is performed independently, allowing all blocks to be processed efficiently in parallel and distributed across the GPU's parallel computing units.

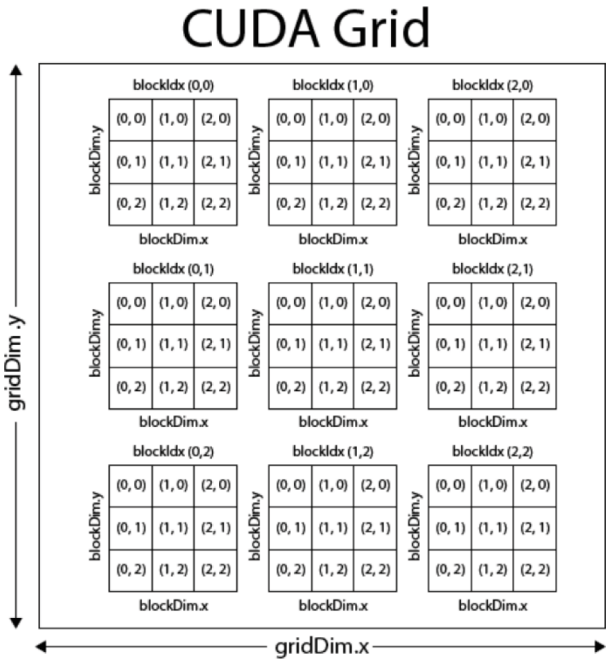

Fig. 2. Threads e Blocks within CUDA grid.

HpApprDCT adheres to the same theoretical framework as [9], with key enhancements.

All compression phases described in Subsect. 3.2 have been fully parallelized by developing dedicated CUDA C kernels to run each operation efficiently.

The compression process consists of the following steps:

1. **Sub-sampling:** The kernel `sub_matrix_scalar()` implemented performs the parallel subtraction of a scalar value of 128 from the image matrix.
2. **DCT Computation:** The kernel `cuda_matrix_dct()` executes the Discrete Cosine Transform (DCT) in parallel by applying the matrix multiplication $T \cdot X_b \cdot T^T$ to each image block X_b within the image matrix X. This parallelized implementation distributes the workload across multiple threads.
3. **Quantization:** The quantization operation is executed in parallel by the `divide_matrices()` kernel, which applies element-wise division of the obtained coefficients by the predefined quantization matrix Q.

The decompression phase follows the inverse sequence of operations, with each step executed in parallel:

1. **De-quantization:** The kernel `divide_matrices()` performs the element-wise multiplication of the quantized coefficients by the predefined quantization matrix Q, restoring them to their original scale.

2. **Inverse DCT Computation:** The kernel `cuda_matrix_idct()` executes the inverse Discrete Cosine Transform (IDCT) in parallel, applying the matrix multiplication $T^T \cdot C_b \cdot T$ to each block C_b of the coefficient matrix C, reconstructing the image data.
3. **Up-sampling:** The kernel `add_matrix_scalar()` applies a parallel addition of a scalar value (128) to the reconstructed image matrix, restoring the pixel values to their original range.

To maximize computational efficiency, several optimizations have been introduced: data transfer from global to shared memory is performed in a coalesced manner to enhance memory access efficiency, the quantization matrix has been relocated to constant memory, which is optimized for read-only access, and further ones explained below.

- First, the **mapping** between image blocks and thread blocks has been **refined**. Previously, a 8×8 thread block was mapped to eight 8×8 image sub-blocks, meaning that a single thread computed an entire row of coefficients. In the new approach, an 8×8 thread block is mapped to a single 8×8 image sub-block, so that an entire row of threads computes one row of coefficients (see Fig. 3).
- The number of blocks and the number of threads per block have been carefully adjusted to **maximize** Streaming Multiprocessor (SM) **occupancy**, ensuring optimal utilization of GPU's computational resources.
- All read-only matrices have been relocated to **constant memory**, which is specifically optimized for such operations. This optimization reduces memory access latency and improves overall computational efficiency by allowing faster data retrieval compared to global memory.
- Finally, the **coalescing technique** has been employed when transferring data from global memory to shared memory (or register). This approach ensures that memory transactions are efficiently grouped, minimizing memory access overhead and maximizing memory bandwidth utilization, thereby enhancing parallel execution performance.

3.4 Implementation Details

The kernel responsible for the DCT receives the image data and the transform matrix as inputs, outputting the coefficients matrix to global memory. Within the kernel:

- Variables required for the DCT are stored in the shared memory.
- Each thread computes its global ID.
- The data are copied from global to shared memory in a coalesced manner, with each thread copying a single element.

With a one-to-one mapping between thread blocks and image blocks, each thread computes a single coefficient as shown in figure (see Fig. 3). It multiplies one

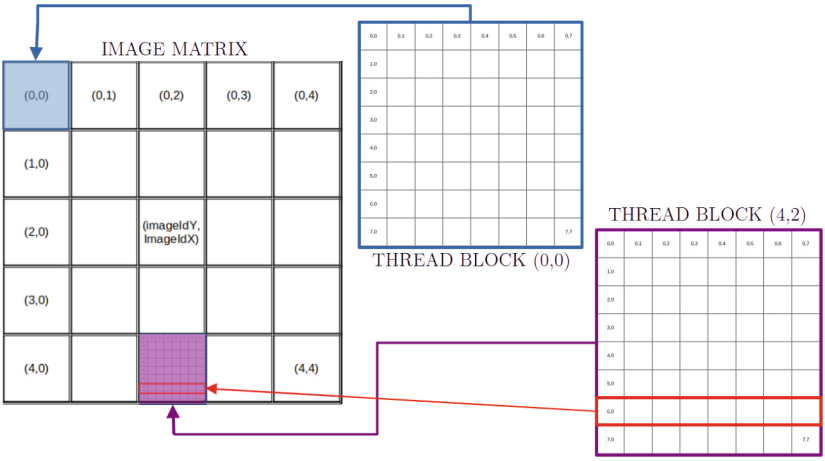

Fig. 3. Mapping between threads blocks and image blocks.

row of T by a column of the image block X_b and then multiplies the resulting value by a row of T to complete the DCT. A synchronization barrier is used between the two multiplications to ensure that intermediate results stored in `shared_matrix[]` are available to all threads.

4 Numerical Experiments

The hardware setup for the benchmarks was a Nvidia Tesla T4, featuring 2560 CUDA cores and 16 GB of GDDR6 memory. The software platform included NVIDIA driver v550.54.15 and CUDA toolkit 12.5, with the code compiled using NVCC version 12.5.82 and the parameters `-arch = sm _75, -O3`. Benchmarks were executed for all algorithms across all available input sizes.

It is worth noting that the proposed algorithm does not rely on Turing specific instructions, but exclusively utilizes CUDA C. Consequently, it is forward compatible with subsequent GPU generations. In order to run the algorithm on newer architectures simply update the compilation command by specifying the desired target architecture in the appropriate flag. (e.g., using `-arch=sm_80` for Ampere or `-arch=sm_90`). Secondly, the proposed algorithm is currently limited to single-GPU execution; scalability across multi-GPU architectures remains a key objective for future research to further enhance computational performance.

We tested four compression algorithms:

- cublasDCT: uses **cuBLAS** library, computes DCT on 8×8 sub-blocks;
- cublasDCTv2: also uses **cuBLAS**, but computes the DCT over the entire image;
- fastApprDCT: an implementation of the algorithm proposed by [9];
- HpApprDCT: the **proposed implementation**, optimized with High performance techniques.

Each assessed on six input sizes ranging from 2^8 to 2^{13}, resulting in a total of 24 runs. To ensure reliable benchmark results, each configuration was executed 100 times on randomly generated images. All execution times were recorded, and the average times were calculated.

4.1 Accuracy Evaluation

In this section, we present an extensive evaluation of the accuracy of our proposed GPU-based image compression algorithm. The analysis is performed by several significant key metrics.

First, we report the test accuracy results using the Percentage Error Energy Norm (PEEN) defined as:

$$\text{PEEN} = \sqrt{\frac{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}[I(m,n) - IR(m,n)]^2}{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} I^2(m,n)}} \times 100, \tag{2}$$

where I is the original image, and IR the reconstructed; the Mean Squared Error (MSE) defined as:

$$\text{MSE} = \frac{1}{m \cdot n}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}[I(m,n) - IR(m,n)]^2. \tag{3}$$

In addition, we compare the compression factor (the ratio between the original size and the compressed one) achieved by our method.

Table 1. Performance evaluation of HpApprDCT on the circuit image, by metrics (M) versus retained coefficients (C).

M\C	6	7	8	9	10	Standard
PEEN	9.92	8.34	8.25	8.13	6.07	4.66
MSE	79.99	56.67	55.32	53.80	30.04	17.67
Compr. Factor	1.49	1.49	1.40	1.35	1.35	1.29

In Table 1, we list the values of PEEN, MSE and the compression factor obtained by varying the number of retained coefficients. In addition, we compare the PEEN, MSE and compression factors achieved with benchmark values, to assess the efficiency of our method in preserving critical image details.

In Fig. 4, we plot PEEN values versus the number of retained coefficients (from 6 to 10) obtained by our algorithm HpApprDCT and by the PADCT algorithm proposed by [9] for comparison .

Finally, we provide visual comparisons by displaying original and compressed images of circuit and cameraman (side by side), which further highlights the practical impact of our approach on image quality.

The Figs. 5 and 6 show original cameraman and original images and the compressed one, respectively, using the standard quantization matrix.

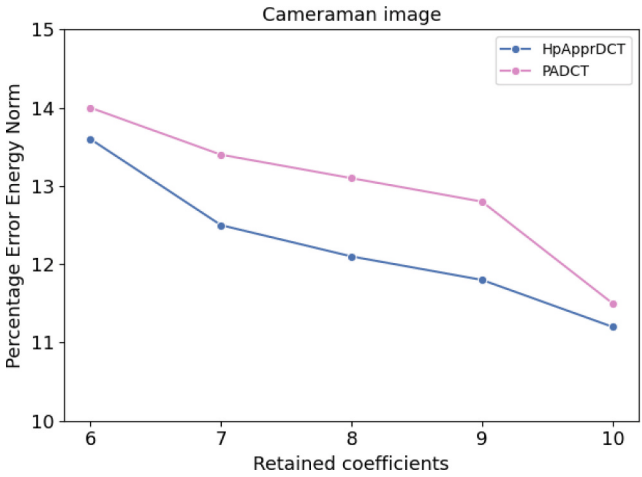

Fig. 4. PEEN values of the PADCT [9] and of the HpApprDCT versus the number of retained coeffcients, from 6 to 10.



Fig. 5. Comparison of cameraman images.

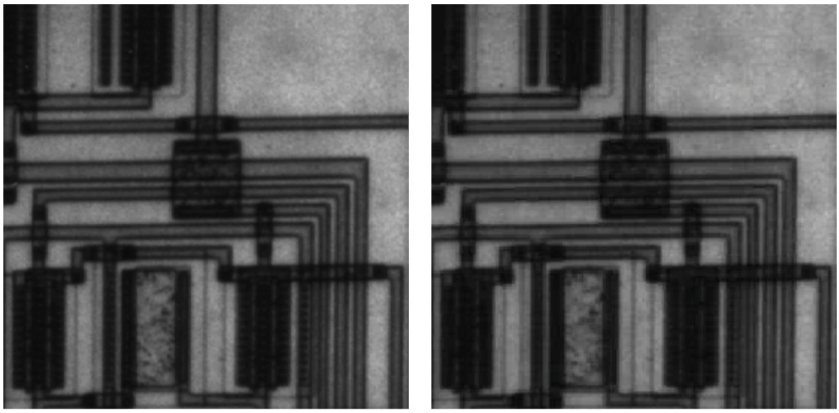

Fig. 6. Comparison of circuit images.

4.2 Efficiency Evaluation

In this section, we measure and compare the execution times of our method against alternative approaches across a range of image sizes. The evaluation is structured to include a detailed tabulation of CPU times for all algorithms, a line chart that illustrates trends in execution time, and a histogram that highlights the speedup achieved. This systematic analysis provides valuable insights into the scalability and performance benefits of our approach.

Table 2. Execution times of algorithms in milliseconds

Image size	DCT on CPU	fastApprDCT	HpApprDCT
256×256	4.7	0.28	0.07
512×512	17.9	0.33	0.12
1024×1024	72.8	0.61	0,30
2048×2048	291.7	1.65	1.04
4096×4096	1255.1	5.80	4.00
8192×8192	5005.1	20.00	14.70

To begin with, we measured the execution times of our algorithm on images of several sizes, ranging from 256 to 8192. Table 2 also shows the execution time of CPU. Next, the line chart, shown in Fig. 7, shows the trends in execution time as the image size increases, clearly illustrating how our GPU implementation scales compared to the others.

Finally, the histogram of Fig. 8 highlights the speedup achieved by the two fastest methods, providing a clear visualization of their performance gains. This

systematic analysis offers valuable insights into the scalability and overall efficiency of our approach.

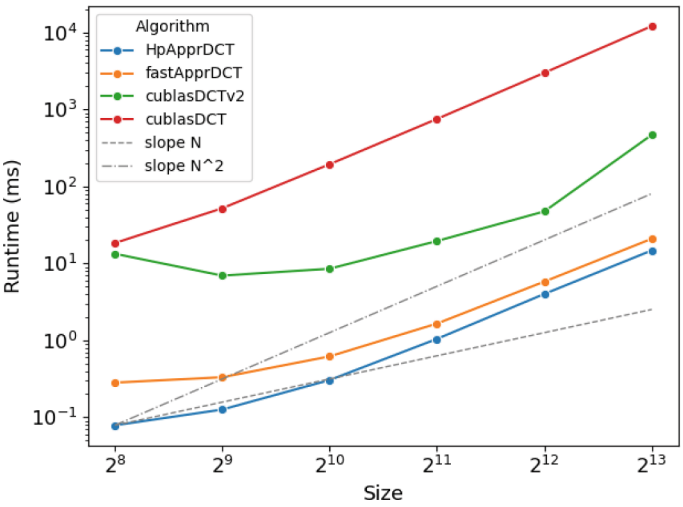

Fig. 7. Comparison of execution times.

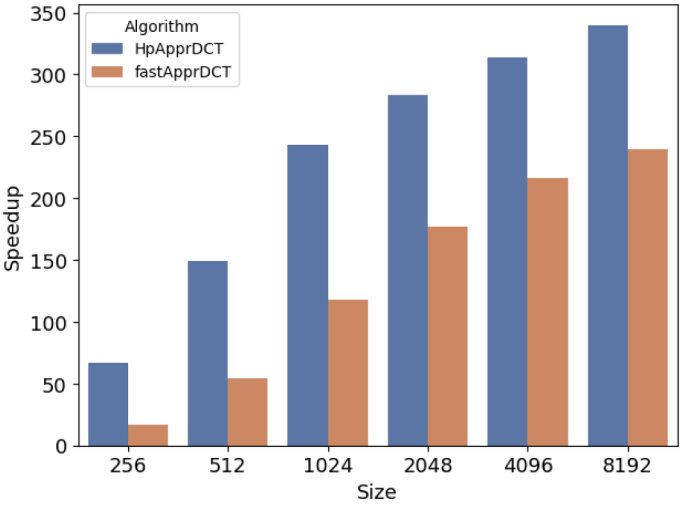

Fig. 8. Speed-up chart.

4.3 Impact of Optimization Strategies

It is important to discuss the relative impact of the individual optimization strategies based on established GPU computing literature.

Since global memory bandwidth is the primary bottleneck in parallel matrix operations [15], the use of **memory coalescing** coupled with **shared memory** is the most critical and beneficial performance enhancement. Indeed, relying on uncoalesced memory access severely bottlenecks computational throughput [18]. Secondarily, the refined **thread mapping** strategy increases the occupancy of the streaming multiprocessor (SM), which is essential to hide memory latency and fully exploit the device's resources [15]. Finally, relocating matrices to **constant memory** has a less dominant impact, but it still effectively enhances the broadcasting of read-only data across warps.

Overall, these optimizations act synergistically to achieve the excellent reported performances and ensure a highly use of hardware resources.

5 Conclusions

This study has presented an optimized implementation of the Discrete Cosine Transform (DCT) on a GPU, focusing on efficient matrix transformation techniques. The key contribution lies in the adoption of advanced algorithms that leverage memory management strategies, such as coalesced memory access and thread divergence reduction, to enhance the overall efficiency of parallel computation on a GPU.

Compared to existing solutions, the proposed algorithm (**HpApprDCT**) has demonstrated a significant performance improvement, reducing execution time by up to 31% compared to the *fastApprDCT* algorithm and more than 96% compared to *cublasDCTv2*, which relies on the cuBLAS library. Furthermore, the proposed algorithm achieves a speed-up of 340× compared to the sequential CPU version, more than 100× higher than the best speed-up obtained by *fastApprDCT*.

This performance gain enables the efficient application of DCT in scenarios where processing speed is critical, such as real-time image compression for transmission purposes.

Future developments may regard the extension the algorithm to support color image processing. As a matter of fact, the current version of the algorithm, operates exclusively on grayscale images, due to the use of the standard C library `libjpeg`., the current implementation. Further developments may regard a more extensive evaluation of the performances of the algorithm on different hardware models.

Acknowledgments. This study was partially funded by INdAM-GNCS project. A. Cardone is a member of Gruppo Nazionale per il Calcolo Scientifico, Istituto Nazionale di Alta Matematica (INdAM).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bouguezel, S., Ahmad, M.O., Swamy, M.: Low-complexity 8× 8 transform for image compression. Electron. Lett. **44**(21), 1249–1250 (2008)
2. Bouguezel, S., Ahmad, M.O., Swamy, M.: A low-complexity parametric transform for image compression. In: 2011 IEEE International Symposium of Circuits and Systems (ISCAS), pp. 2145–2148. IEEE (2011)
3. Cardone, A., De Luca, P., Galletti, A., Marcellino, L.: Solving time-fractional reaction-diffusion systems through a tensor-based parallel algorithm. Physica A Stat. Mech. Appl. **611**, 128472 (2023)
4. Ciaramella, A., De Angelis, D., De Luca, P., Marcellino, L.: Accelerated numerical simulations of a reaction-diffusion-advection model using Julia-CUDA. Mathematics **13**(9), 1488 (2025)
5. Conte, D., D'Ambrosio, R., Paternoster, B.: GPU-acceleration of waveform relaxation methods for large differential systems. Numer. Algorithms **71**(2), 293–310 (2016)
6. Cuomo, S., De Michele, P., Galletti, A., Marcellino, L.: A GPU parallel implementation of the local principal component analysis overcomplete method for DW image denoising. In: 2016 IEEE Symposium on Computers and Communication (ISCC), pp. 26–31. IEEE (2016)
7. Cuomo, S., Galletti, A., Marcellino, L.: A GPU algorithm in a distributed computing system for 3D MRI denoising. In: 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pp. 557–562. IEEE (2015)
8. Conte, D., Cuesta, E., Valentino, C.: Non-stationary wave relaxation methods for general linear systems of Volterra equations: convergence and parallel GPU implementation. Numer. Algorithms **95**(1), 149–180 (2024)
9. Haweel, R.T., El-Kilani, W.S., Ramadan, H.H.: Fast approximate DCT with GPU implementation for image compression. J. Vis. Commun. Image Represent. **40**, 357–365 (2016)
10. Haweel, T.I.: A new square wave transform based on the DCT. Sig. Process. **81**(11), 2309–2319 (2001)
11. Niemeyer, K.E., Sung, C.J.: Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. J. Supercomput. **67**(2), 528–564 (2014)
12. Pennebaker, W.B., Mitchell, J.L.: JPEG: Still Image Data Compression Standard. Springer, New York (1992)
13. Pratx, G., Xing, L.: GPU computing in medical physics: a review. Med. Phys. **38**(5), 2685–2697 (2011)
14. Rea, D., Perrino, G., di Bernardo, D., Marcellino, L., Romano, D.: A GPU algorithm for tracking yeast cells in phase-contrast microscopy images. Int. J. High Perform. Comput. Appl. **33**(4), 651–659 (2019). https://doi.org/10.1177/1094342018801482
15. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 73–82. Association for Computing Machinery (2008). https://doi.org/10.1145/1345206.1345220
16. Santos, L., Magli, E., Vitulli, R., López, J.F., Sarmiento, R.: Highly-parallel GPU architecture for lossy hyperspectral image compression. IEEE J. Sel. Top. Appl. Earth Observ. Remote Sens. **6**(2), 670–681 (2013)

17. Simek, V., Asn, R.R.: GPU acceleration of 2D-DWT image compression in MATLAB with CUDA. In: 2008 Second UKSIM European Symposium on Computer Modeling and Simulation, pp. 274–277. IEEE (2008)
18. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, pp. 1–11 (2008). https://doi.org/10.1109/SC.2008.5214359