Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 659–664
DOI: 10.15439/2025F1886
ISSN 2300-5963 ACSIS, Vol. 43

A Multithreaded Java-Based Video Encoder for Multicore Systems

Maciej Okoń
0009-0005-7427-6784
Maria Curie-Skłodowska University (UMCS)
ul. Pl. Marii Curie-Skłodowskiej 5, 20-031 Lublin, Poland

Beata Bylina
0000-0002-1327-9747
Maria Curie-Skłodowska University (UMCS)
ul. Pl. Marii Curie-Skłodowskiej 5, 20-031 Lublin, Poland
Email: beata.bylina@mail.umcs.pl

***Abstract*—The growing demand for efficient video compression solutions underscores the increasing significance of research into coding optimization on multi-core systems.**

This paper presents the implementation and performance analysis of a multithreaded video encoder developed in Java and optimized for modern multi-core processors. The encoder performs intra-frame compression of raw YUV 4:2:0 video using a frame-based parallelism approach. This method maximizes CPU core utilization while minimizing thread management overhead.

Tests were conducted on five platforms equipped with multi-core processors from Intel and AMD. The application's execution time was measured for varying numbers of frames. The obtained results demonstrate effective scalability of the encoder as the number of processed frames increases, confirming that Java can be an effective tool for implementing parallel video compression on multi-core systems.

I. Introduction

THE INCREASING demand for efficient video processing continues to test the limits of contemporary computing platforms. Video compression is a computationally intensive task, particularly for high-resolution, high frame rate content, and achieving real-time or near-real-time performance is often essential in applications such as live streaming, video surveillance, or large-scale media processing pipelines. While dedicated hardware accelerators like GPUs are often used for such workloads [1], many real-world applications, especially in server environments, rely on general-purpose CPUs [2]. Numerous studies have explored video encoding algorithms and performance optimization techniques, providing valuable insights into how encoding efficiency can be improved through software and hardware-level enhancements [1], [3], [4].

This paper presents a performance-oriented evaluation of multithreaded video compression on modern server-grade processors using a custom-built encoder implemented in Java. Java was selected for its platform independence and robust support for multithreading, which facilitated the implementation of parallel processing using native concurrency constructs and stream-based data pipelines [5]. The encoder performs intra-frame compression on raw YUV 4:2:0 video and supports both single-threaded and multithreaded execution. Frame-based parallelism was used instead of block-based parallelism to more effectively leverage the high thread counts available on the test machines. This approach ensures sufficient workload distribution across all cores, avoiding granularity limitations that arise when the number of parallel tasks is too low. It also reduces overhead associated with thread management, allowing the encoder to scale efficiently with increasing core counts. Our evaluation measures the total encoding time, starting after the video data is fully loaded into memory and ending once the encoded file has been written to disk. As such, the performance results reflect not only the efficiency of parallel computation, but also task distribution overhead, thread synchronization, and serial operations such as output handling.

Five multicore platforms were tested, representing a range of CPU architectures and core counts. By comparing execution time and speedup across varying frame counts, we explore the strengths and limitations of each system in handling a realistic, mixed-parallelism workload. Rather than isolating idealized, parallel-only scenarios, our measurements reflect the practical end-to-end behavior of CPU-based encoding workflows.

This article is organized as follows. Section II begins with an introduction to video encoding and compression, describing the basic concepts and challenges of reducing video data while maintaining visual quality. It explains how video encoders convert raw video images into a compressed format through algorithmic steps. It also highlights the computational requirements of these processes and the importance of parallelization for performance. Section III presents the main video compression algorithms such as discrete cosine transform (DCT), quantization and motion estimation. It explains how these techniques reduce redundancy and achieve efficient compression. Section IV describes the architecture and internal structure of the encoder, detailing the modular design, block-based processing and the implementation of parallelization at the frame level using multithreading in Java. Section V describes the test environment and methodology, followed by an analysis of the performance results on different systems, comparing single-threaded and multithreaded execution. Section VI presents the conclusions from the experiments and possible directions for further research.

II. Video encoding and compression

Video encoding is a computationally demanding process that transforms raw, high-resolution video data into a compressed format suitable for storage and transmission [6]. This transformation typically involves a series of algorithmic

steps designed to reduce redundancy and preserve perceptual quality.

At the core of most intra-frame compression schemes lies a block-based pipeline consisting of the following operations [7]:

1) **Block partitioning:** the input frame is divided into fixed-size macroblocks (e.g., 16×16 pixels).
2) **Transformation:** each macroblock is converted into the frequency domain using the Discrete Cosine Transform (DCT), enabling more efficient compression by concentrating signal energy into fewer coefficients.
3) **Quantization:** DCT coefficients are scaled and rounded to reduce precision, effectively discarding less perceptually significant data.
4) **Entropy coding:** quantized coefficients are compressed using techniques such as run-length encoding (RLE) to eliminate statistical redundancy.

These stages are computationally expensive, particularly at high resolutions and frame rates. As a result, the performance of a video encoder depends not only on the efficiency of these individual algorithms, but also on how well they are integrated into a scalable execution model. Our approach focuses on frame-based parallelization, where individual frames are processed concurrently across multiple threads.

III. Transform coding of video data and reduction algorithms

TRANSFORM coding is one of the more computationally demanding steps in the encoding pipeline. This process converts visual data from the spatial domain — where it's represented as individual pixel values — into the frequency domain. It is used to isolate the components that carry the most visual weight from those that can be discarded with minimal impact on perceived quality [7]. After the transformation, we reduce the precision of the less important frequency components using quantization. This lossy compression step shrinks the file size drastically, without noticeably affecting how the video looks [8]. The specific technique we rely on for this transformation is the Discrete Cosine Transform (DCT).

A. Discrete Cosine Transform (DCT)

DCT represents a block of an image as a weighted sum of cosine waves oscillating at different frequencies. This shift from pixels to frequency components reveals a useful pattern: most of the important visual information tends to concentrate in just a few low-frequency coefficients, while the high-frequency components — often corresponding to fine detail or noise — can be compressed more aggressively or discarded entirely [9].

In a typical implementation, the process begins by dividing each video frame into smaller blocks, usually 8×8 pixels. Each of these blocks is then transformed using the 2D DCT. The result is a matrix of coefficients that describe the contribution of each frequency component within that block — the top-left corner of the matrix holds the DC (average) value, while the remaining entries describe increasingly fine detail as you move outward. This structure allows us to perform quantization more intelligently. Because human vision is more sensitive to low-frequency information, we can preserve those coefficients with higher precision and apply more aggressive compression to the less perceptible high frequencies.

Mathematically, the 2D DCT (type II, $O(N^4)$) for a block of size $N \times N$ is defined as follows:

$$Y(u,v) = \frac{1}{4} \cdot c(u) \cdot c(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cdot \cos\left(\frac{(2x+1)\cdot u \cdot \pi}{2N}\right) \cdot \cos\left(\frac{(2y+1)\cdot v \cdot \pi}{2N}\right) \tag{1}$$

where:

- $Y(u,v)$ is the resulting DCT coefficient at frequency indices (u,v),
- $f(x,y)$ is the input pixel value at position (x,y),
- $c(k) = \frac{1}{\sqrt{2}}$ when $k = 0$, and $c(k) = 1$ otherwise,
- N is the block size, typically 8.

One practical advantage of DCT is its separability. The 2D transform can be broken down into a 1D DCT applied first along the rows, and then along the columns (or vice versa). This significantly reduces the computational complexity ($O(N^3)$) [10].

B. DCT quantization

Once each image block has been transformed via DCT, we're left with a set of frequency-domain coefficients. While many of these coefficients — especially the ones representing higher frequencies — contribute little to the perceived visual quality, they still occupy space. Quantization reduces this burden by simplifying these values.

The process involves dividing each coefficient by a corresponding value from a quantization matrix and then rounding the result:

$$Q(x) = \text{round}\left(\frac{x}{Q_{i,j} \cdot S}\right) \tag{2}$$

Where:

- $Q(x)$ is the quantized coefficient,
- x is the original DCT coefficient,
- $Q_{i,j}$ is the value from the quantization matrix at position (i,j),
- S is a scaling factor that adjusts the compression strength.

The quantization matrix controls how much detail is kept in different parts of the image. Lower numbers in the matrix correspond to low-frequency components, which capture the general structure and smooth areas of the image—these are preserved more accurately. Higher numbers are assigned to high-frequency components, which represent fine details or noise; these are compressed more aggressively.

A commonly used quantization matrix from the JPEG standard is shown below.

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

C. Motion estimation and compensation

To further improve compression efficiency, modern video encoders utilize the similarity between consecutive frames through motion estimation and compensation - processes that analyze changes between frames and predict what future frames will look like based on the movement of visual elements.

Step 1: Motion estimation. This step works by comparing two consecutive frames to identify macroblocks that have shifted in position. This is usually done by searching within a defined area of the reference frame to find the best match for each block in the current frame. Metrics such as the Sum of Absolute Differences (SAD) or Mean Squared Error (MSE) are used to evaluate how well blocks match. The result of this process is a set of motion vectors, which describe how far and in which direction each block has moved.

Step 2: Motion compensation. This step then uses these vectors to shift corresponding blocks in the reference frame to approximate the content of the current frame. Instead of encoding the full frame, the encoder stores only the difference (residual) between the predicted and the actual frame. This residual typically contains less information than the original frame.

IV. Encoder architecture

The encoder was developed in Java with a modular, object-oriented structure, centered around the core classes `VideoObjectPlane`, `Macroblock`, `Bitstream`, and `VOPHeader`. Each video frame is divided into 16x16 pixel macroblocks, which are then processed using DCT, quantization, and run-length encoding (RLE).

The encoder supports two execution modes: sequential and multithreaded. In the multithreaded version, the `ExecutorService` from `java.util.concurrent` package is used to create a thread pool that matches the number of available CPU cores. This allows each video frame to be processed independently in a separate thread, significantly improving encoding performance. Synchronization is handled through `Future` objects, which act as placeholders for the results of asynchronous tasks and allow results to be collected in order. .

Two levels of parallelism were considered during implementation: frame-level and block-level. Frame-level parallelism was chosen and implemented, with each frame processed in a separate thread. This approach aligned well with the hardware characteristics of the testing environment. While block-level parallelism was explored as a finer-grained alternative, it proved to be significantly slower, and thus not integrated into the final encoder.

The encoder processes raw video data stored in a YUV 4:2:0 file with a resolution of 1920x1080 pixels. This format represents color using separate planes for luminance (Y) and chrominance (U and V), with chrominance subsampled by a factor of two in both horizontal and vertical dimensions. Video frames are loaded using the `loadYUVFrames` function, which reads the raw data from disk, partitions it into individual frames, and converts them to `BufferedImage` objects in RGB color space. Each frame is then encoded as an `I_VOP` object, that then segments the image into individual macroblocks and compresses each block.

Each `Macroblock` object stores pixel data in the same 4:2:0 structure, with luminance represented by 256 samples and each chrominance channel by 64 samples, corresponding to the 16×16 luminance block and its associated 8×8 chrominance blocks. These values are stored in separate one-dimensional arrays for Y, U, and V components and are extracted during macroblock construction from the input RGB image via internal color space conversion. Internally, the pixel data is converted from RGB to YUV, then further processed into 8x8 sub-blocks for DCT and quantization. The DCT implementation uses separable 2D transformation with precomputed cosine values for efficiency and relies on double-precision arithmetic to maintain numerical accuracy during computation.

The result of the encoding process is stored in a binary file, written using the custom `Bitstream` class. The output maintains the original block order thanks to the use of `Future.get()`. This ensures deterministic output even when block processing completes in a different order.

The entire encoder was developed using standard Java libraries, including `java.util.concurrent` for parallelism and `java.awt` for image handling. Due to its modular architecture, the system can be extended in the future with additional features, such as motion estimation for inter-frame coding or dynamic encoding strategies based on scene characteristics like `Video Objects`.

V. Performance analysis and test environment

To evaluate the performance of the implemented video encoder, a series of experiments were conducted on five different computing platforms, each equipped with a distinct server-grade processor, selected to cover a range of architectures, core counts, and generations:

- **C1 (2 x Intel Xeon E5-2670 v3)** – 12 cores / 24 threads, Haswell-EP architecture (22 nm), released in 2014. An older platform with limited performance compared to modern standards.
- **C2 (2 x Intel Xeon Gold 5218R)** – 20 cores / 40 threads, Cascade Lake Refresh (14 nm), released in 2020. A mid-range server CPU offering a good balance between core count and clock speed.

- **C3 (2 x Intel Xeon Gold 6342)** – 24 cores / 48 threads, Ice Lake-SP architecture (10 nm), released in 2021. With improved single-core performance and memory bandwidth support.
- **C4 (2 x Intel Xeon Platinum 8358)** – 32 cores / 64 threads, Ice Lake-SP architecture (10 nm), released in 2021. High core density suited for parallel workloads.
- **C5 (2 x AMD EPYC 9654)** – 96 cores / 192 threads, Zen 4 (Genoa) architecture (5 nm), released in 2022. A high-end processor optimized for massive parallelism and throughput.

Each system processed a standard YUV 4:2:0 1920x1080 video file, with frame counts ranging from 250 to 2000 for the multithreaded (MT) version[1] and from 250 to 1500 for the single-threaded (ST) version[2].Depending on the number of frames, the size of the test files ranged from approximately 741 MB (250 frames) to 5.79 GB (2000 frames). This setup allowed for the analysis of both raw execution time and scalability with increasing frame counts.

To quantify the performance gain achieved through parallelization, the speedup factor was calculated as the ratio of the single-threaded execution time to the multithreaded execution time:

$$S = \frac{T_{\mathrm{ST}}}{T_{\mathrm{MT}}}. \tag{3}$$

The performance data collected across all tested platforms is summarized in Table I, which lists the minimum measured processing times for both single-threaded and multithreaded executions, along with the computed speedup values. The results are presented for increasing frame counts, illustrating how processing time scales with input size. Speedup values were calculated using equation 3. Figures 1 and 2 visualize the speedup and multithreaded processing times across all systems, respectively, providing a comparative view of scalability and performance gains with increased threading and workload.

In general, all platforms demonstrate substantial performance improvements from multithreading, with speedup tending to increase as the number of processed frames grows. However, the magnitude and consistency of these gains vary across systems. Notably, some platforms—particularly `C2` `C3`, and `C4` —exhibit a plateau or even slight dip in speedup around mid-range workloads (750 to 1250 frames), indicating that the platforms either have, or are close to reaching their maximum effective scaling and additional workload does not significantly improve parallel efficiency.

Among the tested systems, the AMD-based `C5` platform initially shows comparatively modest speedup at lower frame counts, which can be attributed to the overhead of managing its high core count when the workload is too small to fully utilize the available resources. As the workload increases, `C5`'s scalability becomes clear: by the 1500-frame mark, it surpasses all other platforms in speedup and achieves the shortest absolute multithreaded processing time at 2000 frames.

Interestingly, `C1`, despite having the poorest absolute processing times, consistently achieves the second highest speedup among all platforms. This indicates that while its baseline single-threaded performance is limited, it benefits strongly from multithreading, scaling efficiently with increasing workload size. In contrast, `C2`, which also features fewer cores, shows more consistent but lower speedup values, plateauing around 12 to 13× speedup across moderate to larger frame counts, and is eventually outperformed by more scalable systems at higher workloads. `C3` and `C4` occupy a middle ground, maintaining solid and relatively stable speedup improvements that exceed those of `C2`, yet do not quite match the top-end performance and scalability of `C5` at larger frame counts.

VI. Conclusion

This work aimed to design, implement, and evaluate a multithreaded video encoder in Java, capable of leveraging the parallel processing capabilities of modern multicore architectures. In pursuit of this goal, the encoder was developed using a modular, object-oriented structure in Java, incorporating key compression techniques such as the Discrete Cosine Transform (DCT), quantization, and run-length encoding (RLE). The implementation included both single-threaded and multithreaded execution modes, enabling a comparative analysis of encoding performance across different hardware platforms and workloads. A series of experiments were conducted using five server-grade systems with varying core counts and architectures, focusing on the encoding of high-definition video data in YUV 4:2:0 format. The objective was to assess the scalability and performance benefits of multithreading for video encoding tasks.

The results demonstrate that all tested CPUs benefit substantially from multithreading, with speedups ranging from approximately 9x to over 24x depending on the system and workload. Among them, `C5` achieved the highest ultimate acceleration, peaking above 24× speedup at larger frame counts, demonstrating superior scalability on high concurrency workloads despite initially modest gains at lower frame counts. Notably, `C1`, despite having the slowest absolute processing times, consistently achieved the second highest speedup, indicating efficient scaling relative to its baseline single-threaded performance. `C3` and `C4` maintained solid and relatively stable speedup improvements, exceeding `C2`'s generally lower but consistent gains, which plateaued around 12 to 13× at moderate to large workloads.

Importantly, our measurements suggest that raw core count alone does not fully explain observed performance. Differences in scaling behavior—especially how speedup grows with frame count—highlight the role of thread orchestration overhead and single-threaded efficiency in managing output and coordination. For example, while `C5` initially showed lower speedup at small workloads, it ultimately surpassed

[1]A total of 2000 frames were used for testing. This value was chosen as the maximum that could be reliably loaded across all systems, given their varying available memory.

[2]Due to long processing times, the single-threaded encoder was only tested up to 1500 frames.

TABLE I
MINIMUM ST AND MT PROCESSING TIMES AND COMPUTED SPEEDUP ACROSS ALL MACHINES

Frame Count	C1			C2			C3			C4			C5		
	T_{ST} [s]	T_{MT} [s]	Speedup	T_{ST} [s]	T_{MT} [s]	Speedup	T_{ST} [s]	T_{MT} [s]	Speedup	T_{ST} [s]	T_{MT} [s]	Speedup	T_{ST} [s]	T_{MT} [s]	Speedup
250	63.702	5.029	12.67	44.123	4.302	10.26	41.383	3.446	12.01	42.752	4.044	10.57	63.874	7.043	9.07
500	141.167	8.460	16.69	87.956	7.202	12.21	83.395	6.052	13.78	85.934	6.769	12.70	131.713	10.498	12.55
750	202.898	11.845	17.13	133.306	10.776	12.37	126.079	8.991	14.02	129.259	9.562	13.52	192.968	11.030	17.49
1000	272.535	15.743	17.31	178.920	14.486	12.35	169.908	12.423	13.68	174.848	13.445	13.00	276.740	14.447	19.16
1250	352.270	18.463	19.08	225.432	18.109	12.45	213.186	15.155	14.07	219.300	16.112	13.61	357.784	18.686	19.15
1500	433.261	22.316	19.41	268.962	21.007	12.80	254.934	17.570	14.51	263.189	18.791	14.01	405.174	16.592	24.42
2000	–	29.113	–	–	29.340	–	–	23.829	–	–	25.065	–	–	23.697	–

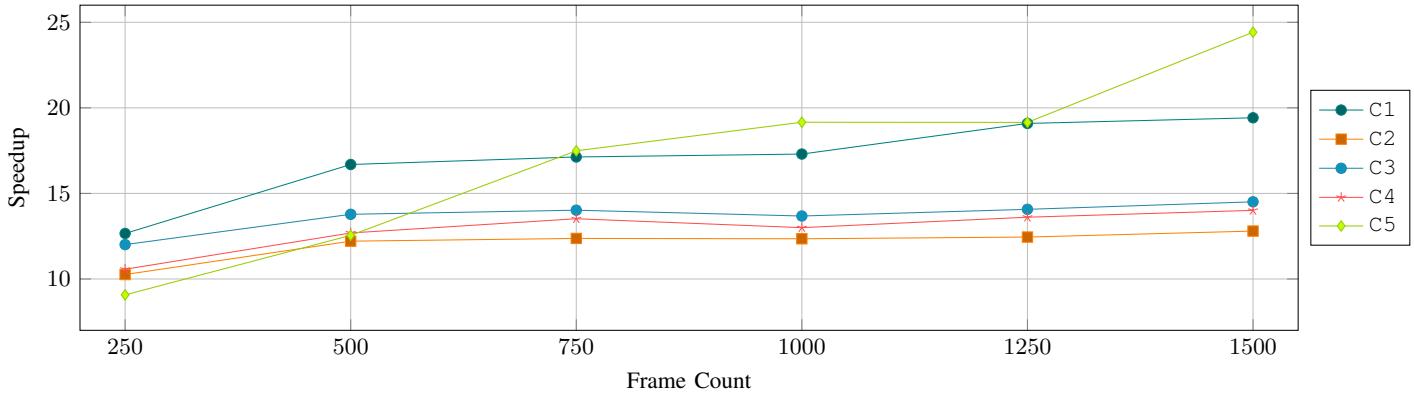

Fig. 1. Speedup of multithreaded processing across all machines

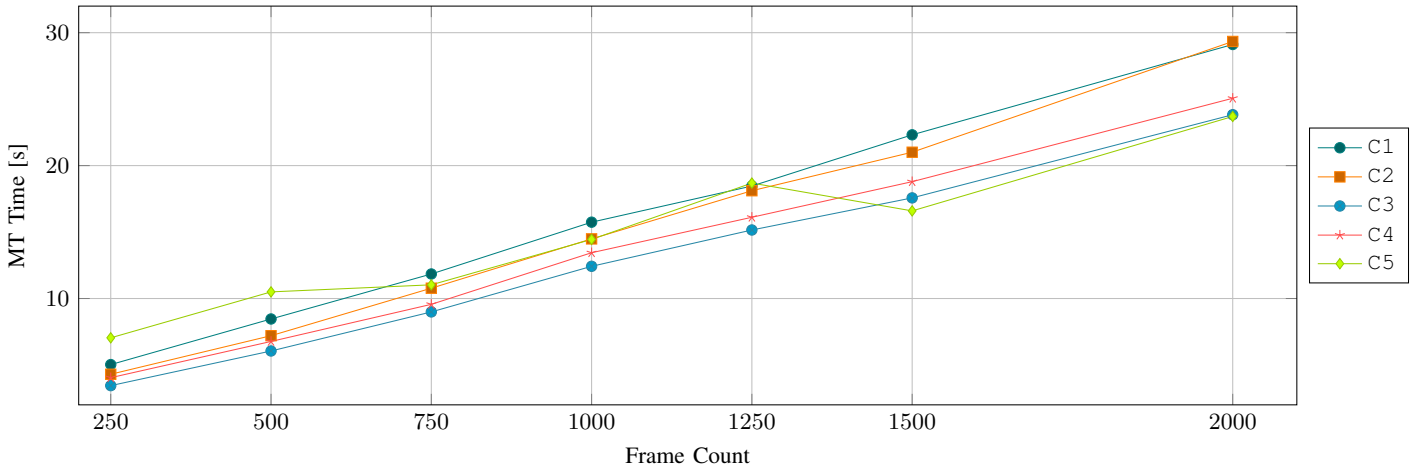

Fig. 2. Comparison of minimum multithreaded processing times for all tested systems

several peers as the workload increased, indicating superior multithreaded scalability at higher concurrency levels.

These findings validate the need to assess full-system performance for parallel workloads, not just isolated parallel execution paths. In practice, the total encoding time is shaped by how well a CPU balances parallel computation with effective scheduling and sequential operations. This is especially relevant in environments without GPU acceleration, where CPUs must take on end-to-end responsibility for video processing.

Future research may explore extending the encoding model to include inter-frame prediction, introducing more complex dependencies across frames, or benchmarking hybrid CPU-GPU configurations. Nevertheless, this work offers a grounded perspective on CPU capabilities in a realistic, high-throughput encoding pipeline, and serves as a foundation for further exploration of CPU-bound multimedia processing.

REFERENCES

[1] S. Katsigiannis, V. Dimitsas, and D. Maroulis, "A GPU vs CPU performance evaluation of an experimental video compression algorithm," in *Proceedings of the 7th International Workshop on Quality of Multimedia Experience (QoMEX)*, Pylos, Greece, May 2015, pp. 1–6. DOI: 10.1109/QoMEX.2015.7148134.

[2] S. Sankaraiah, L. H. Shuan, C. Eswaran, and J. Abdullah, "Performance Optimization of Video Coding Process on Multi-Core Platform Using GOP Level Parallelism," *International Journal of Parallel Programming*, vol. 42, no. 6, pp. 931–947, Dec. 2014. DOI: 10.1007/s10766-013-0267-4.

[3] Y. Zhang, C. Zhang, R. Fan, S. Ma, Z. Chen, and C.-C. J. Kuo, "Recent Advances on HEVC Inter-frame Coding: From Optimization to Implementation and Beyond," *arXiv preprint arXiv:1910.09770*, Oct. 2019. Available: https://arxiv.org/abs/1910.09770.

[4] D. Liu, Y. Li, J. Lin, H. Li, and F. Wu, "Deep Learning-Based Video Coding: A Review and A Case Study," *arXiv preprint arXiv:1904.12462*, Apr. 2019. Available: https://arxiv.org/abs/1904.12462.

[5] Oracle, "Concurrency Utilities," *Java Platform, Standard Edition 8 API Specification*, 2014. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html.

[6] A. J. Kane, "Encoding vs. Decoding," AVIXA, [Online]. Available: https://www.avixa.org/pro-av-trends/articles/encoding-vs-decoding. [Accessed: Dec. 2024].

[7] I. E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. Chichester, U.K.: Wiley, 2003.

[8] D. Le Gall, *MPEG: A Video Compression Standard for Multimedia Applications*. IEEE, 1991.

[9] I.-M. Pao and M.-T. Sun, "Modeling DCT coefficients for fast video encoding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 9, no. 4, pp. 608–616, 1999, doi: 10.1109/76.767126.

[10] S. A. Khayam, "The Discrete Cosine Transform (DCT): Theory and Application," *Course Notes, Department of Electrical & Computer Engineering*, 2003.