A Performance Model of Fast 2D-DCT Parallel JPEG Encoding Using CUDA GPU and SMP-Architecture

Mohammed K. Ali Shatnawi
Department of Electrical and Computer Engineering
University of Ottawa
Ottawa, Canada
mshatnaw@uottawa.ca

Hussein Ali Shatnawi
Department of Computer Science and Information
SAU University
Al-Kharj, Saudi Arabia
h.shatnawi@sau.edu.sa

***Abstract*— The performance of image compression algorithms for big data can be enhanced using parallel computations. JPEG algorithm is a lossy compression method that uses DCT to eliminate high-frequency components. In this paper, we describe a cross-compatible design of JPEG on SMD and GPU architectures. To achieve maximal efficiency, we exploit the substantial parallelism to design an optimized version of JPEG based on thread model. A fair algorithm's evaluation on 24-bit BMP, using several performance metrics, is run on the fully optimized GPU using CUDA and SMP using SESC simulator. Our cross-architectural evaluation results revealed a 25.49 speedup in SESC and 21 in GPU and that CPU outperformed GPU for the JPEG.**

Keywords—JPEG; Parallel JPEG; Fast DCT; GPU; SESC; SMP; Amdahl Law

I. INTRODUCTION

The advent of new multi-core processor technology allowed a variety of options for application designers to improve performance with low costs [1]. This technology was challenged by parallel processing through communicating many computers working together to solve complex computational problems. Furthermore, additional specific knowledge about the multi-core processor architecture and software parallelization is required to fully utilize the parallelism. Obviously, multi-core processors give programmers the chance to execute their code using only the available processing units [1]. Thereby, executing an application using threads where each thread runs on a separate processor will give a great impact of the reduction in overall execution time [2].

Nowadays, multimedia contents are explosively growing over the internet and in many storage spaces, thus the need of efficient images and video compressions become indispensable [3][4][5][7]. JPEG coding provides superior features that meet multimedia processing demands with a very low bit rate [5]. According to JPEG standards, it supports lossy compression of single and multiple components and needs not any resource fork, which provides a quick interchangeable compressed format to ensure maximum compatibility across platforms such as Macintosh, PCs and workstations [11].

Modern PCs may have up to 8-cores (e.g., a new generation of AMD processors) capable of serving user running applications and video IRs. However, the cost of processing video requests is still high and results in high time latency [7]. NVIDIA produced a powerful PIC video equipped with GPUs (graphics processing units) cards that can perform multimedia processing and address fast on-board memory [9][10] and thus providing fast graphics rendering, such as (NVIDIA GeForce GT430) that supports 96 GPU CUDA (compute unified device architecture that provides a programming model for developers) cores for super visual computing. The main GPU's features are [5][7][9][10]: 1) targets level-parallelism by overcoming the limitation of SIMD architecture. 2) Makes use of parallel resources using 100s of ALUs. 3) High tolerance of memory latency. 4) Dedicated architecture for computations through streaming multiprocessors (SMs). 5) On-chip IEEE 754-2008 floating-point standard for single and double precisions.

The GPU consists of global memory accessible by CPU and SIMs that perform in-time computations. Each SIM has several cores, pipelines, cache and CUs. For example: 32 CUDA cores per SIM can compute up to 32 fp32, fp64 and int32 operations/clock. Fig. 1 shows a GPU architecture with parallel code execution.

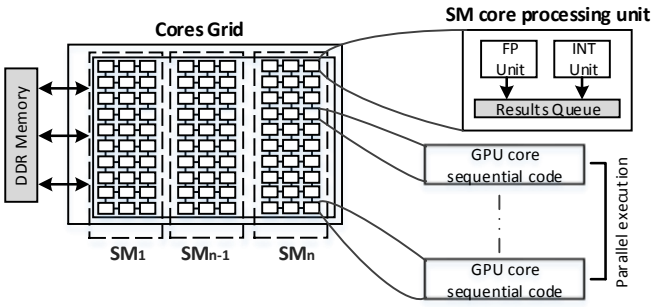

Fig. 1. Parallel code execution on GPU

In order to better evaluate GPU performance, we introduce another kind of Chip Multiprocessors (CMP) at which we can run algorithms in a parallel manner. SESC [15] is a superscalar out-of-order MIPS that can simulates programs as if they run on real Symmetric multi-processors (SMP) machines where all processors are identical. Like other SMP, SESC has several cache levels (I-cache and D-cache) with many parameters and models. The processors are interconnected to enable source-destination delivery using different routing policies. SESC addresses single processor chip multi-core and processors in-

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

memory, thereby, providing a variety of SMP architectural platforms for the development. Unlike SESC, GPU runs code independently of CPU while the first has multiple CPUs to share computational problem. To the best of our knowledge, no other papers on JPEG implementation have considered comparing SESC with GPU on the same algorithm.

In this paper, we evaluate the performance of a full-balanced parallel implementation of the JPEG algorithm on two parallel-supported architectures: GPU (many core systems) and SESC (SMP system). We considered SESC because it's pervasive and has a greater memory latency (compared to the GPU) and provides a standard model for SMP.

The remainder of this paper is structured as follows: section II describes the related work. In section III, we present the parallel JPEG algorithm and provide computational analysis. This is followed by evaluating the algorithm on the two systems in section IV. The experimental results are described in section V. Conclusion and future research directions are conducted in section VI.

II. Background and related works

In the area of parallel implementation and analysis of JPEG coding, Duo et al. [4] implemented the performance of JPEG compression using CUDA GPU and compared the results obtained with serial execution. The authors applied the conventional JPEG on fairly small image sizes and demonstrated a non-optimized algorithm with high running costs. Our work differs from previous parallel JPEG work in that we optimize the algorithm when applicable and compare parallel implementations on different platforms.

Richter et al. [5] modified the design of JPEG encoder that extends from JPEG and JPEG2000 standards. The new design provides fast encoding with a key drawback shown clearly in design complexity. The authors used simple Huffman-run length to decrease running-cost and evaluated their algorithm for single-processor machine.

The authors in [7] evaluated the performance of the existed motion JPEG2000 and MJPEG2000 algorithms (an ISO standard for video format) on CUDA platform. The authors observed that an extra overhead come from intercommunication between GPU cores and the CPU. They compared both algorithms with the sequential implementation for non-run time video frames. A few previous works demonstrated designing of real-time compatible applications.

Pieters et al. [9] observed that running JPEG decoder exploiting the parallelism of GPU could increase real-time image sequences. The implemented parallel algorithm extends from the standard JPEG implementation and provides better speedup for a sequence of images.

Although there exists works for independent evaluation of the parallel JPEG algorithm, optimizing the design of JPEG on multiple architectures that support multi-task computations has not been provided in many other works.

III. Parallel JPEG implementation

In this paper, we exclusively focus on standard image format defined by JPEG, a method that compress image files through five consecutive dependent steps. For further details on JPEG compression, intensive information is provided in [11]. We discuss later the optimization applied to the JPEG algorithm while the following summarizes JPEG steps:

1) Convert from RGB to luminance/chrominance (YCBCR): if the images are grayscales, this step can be skipped. JPEG compression can be tuned by down-sampling the chrominance components with the conversion. In this paper, we use an approximation model to convert colors.
2) Discrete cosine transform (DCT): transforms the image from the spatial domain to the frequency domain. The conventional 2D-DCT transformation is replaced with an enhanced method to achieve maximum performance.
3) Quantization: a predefined 8x8 table that determines the image compression improvement ratio (ICRIF). We stick our design to default values since we are not concerned with the compression ratio reduction.
4) Encoding: this is a critical step in JPEG algorithm and one has to choose the least-cost algorithm to encode the image. The Huffman-run length method is used in our design.

In order to be able to run JPEG on CUDA GPU and SESC, two semi different algorithms are used to obtain optimal throughput. This is because the parameters that control code execution on both architectures differ, for example, if the image is small enough that not all CPUs in SESC involve into computation, the algorithm will not fully utilize the available parallel resources. In contrast, we need to check for service-availability if the image size is too big (no more free cores where some portions of data have to wait to get served). In GPU simulation, the number of available processors is fixed and the algorithm grows as image size grows. For example, if the image is too small, the expected parallel running-time is smaller than the sequential execution for specific reasons that will be discussed in the evaluation section. Also in the GPU, floating-point computations are done in the fused multiply-add (FMA) without referencing to the CPU.

The parallel version of the JPEG algorithm with an estimation of computation complexity is described below. For the remainder of the paper, the following symbols and notations pertain:

A, D	*WxL* matrix representing the input and output image respectively
N	Number of cores supported by the GPU
m_i	Number of occupied cores at round *i*
q	Index of current core or processor
T_{OH}	Overhead time measured in cycles or seconds
T_1	Time taken by a single processor

T_{NP}	Time taken by NP processors
NP	Number of processors (SESC)
S	Speedup: T_1/T_{NP}
η	Efficiency: S/NP, S/N

PARALLEL-JPEG (A: WXL MATRIX)

Instance: Blocks (*B*), JPEG, TH_IDS

1. *Initialize_memory()*
2. *Initialize_processors()*
3. *Set_working_threads()*
4. *Read_bmp_24b()*
5. *Allocate_memory_space()*
6.
7. **TH_IDS** ← *Initialize_threads()*
8. **B** ← *split_to_8x8_blocks(A)*
9. ***Foreach*** block (*b*) in *B*
10. *IF q = N – 2 then* ***wait_for_free_cores()*** *else*
11. **TH_IDS** ← ***Spawn_new_thread****(q++)*
12. ∟*Lock_working_threads();*
13. ∟***Foreach*** element (k) in *b*
14. *B[i, j]* ←YUV(R_k, G_k, B_k)
15. ***Wait_sync_barrier(q)***
16.
17. ***Foreach*** block (*b*) in *B*
18. *IF q = N – 2 then* ***wait_for_free_cores()*** *else*
19. **TH_IDS** ← ***Spawn_new_thread****(q++)*
20. ∟*Lock_working_threads();*
21. ∟*calculate_row_dct()*
22. ∟*calculate_column_dct()*
23. ∟*update(DCT)*
24. ***Wait_sync_barrier(q-q')***
25. ***Free_memory(RGB)***
26.
27. ***Foreach*** block (*b*) in *B*
28. *IF q = N – 2 then* ***wait_for_free_cores()*** *else*
29. **TH_IDS** ← ***Spawn_new_thread****(q++)*
30. ∟*Y[i]* ←*quantize_luminance(b)*
31. ∟*CrCb[i]* ←*quantize_chrominance (b)*
32. ***Wait_sync_barrier(q-q')***
33.
34. ***Iterate over all blocks in B***
35. ∟*Entropy_encding(chrominance)*
36. ∟*Entropy_encding(luminance)*
37. ***Wait_sync_barrier(q-q')***
38. *Update(**JPEG**)*
39.
40. ***Generate_RLC_code()***
41. ***Build_huffman_code_table()***
42. ***Parent:***
43. ***Spawn_new_thread****(N-1)*
44. ∟*set_jpeg_header()*
45. ∟*render_jpeg_24b()*

A. Fast 2D-DCT

The computation of the standard 2D-DCT is divide-and-conquer algorithm. According to the master theorem, the cost-function can be expressed as in (1) giving O(n^4) complexity.

$$T(n) = 64T(n/4) + n^2 \quad (1)$$

The 2D-DCT formula for each 8x8 blocks is expressed as:

$$D_{uv} = \frac{1}{4}\delta(u)\delta(v)\sum_{i=0}^{7}\sum_{j=0}^{7} D[i,j]\cos(\alpha)\cos(\beta) \quad (2)$$

To keep DCT coefficients for every pixel in [-128, 127] range, each block is shifted by 128 with a total of 64 add/subtract operations. This is replaced by subtracting 1024 from DC-coefficient, which involves only performing 1 subtraction, thus, for NxN image, a reduction of ($\frac{63}{64} \approx 98\%$) operations. From (2), the total number of multiply operations required for each block is $2n^2 + 2$, we could use trigonometric relations to eliminate multiplication of cosines, thus saving ~50% of them as in (3).

$$\cos(\alpha)\cos(\beta) = \frac{1}{2}(\cos(\alpha+\beta) + \cos(\alpha-\beta)) \quad (3)$$

The techniques described above could reduce the number of arithmetic required but would not reduce the time complexity. A better version of 2D-DCT described as FCT (fast cosine transform) is used to obtain optimal computations. A parallel FCT shown in Fig. 2 uses 1D-DCT for rows then columns, reducing the complexity to $O(n^3)$. 1D-DCT can be computed using ANN algorithm (for more details, see [12]) that reduces direct matrix multiplication in each 8x8 blocks from 64 to 13 operations. The 1D-DCT is computed using:

$$D_u = \frac{2}{\sqrt{n}}\delta(u)\sum_{i=0}^{W-1}\sum_{j=0}^{L-1} D[i]\cos(\alpha) \quad (4)$$

Moreover, factorizing the computation in a similar manner used in FFT reduces $O(n^2)$ to $O(n\log n)$ for NxN image.

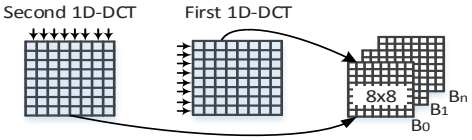

Fig. 2. Parallel block transposes in the FCT

B. Parallel Model Analysis

The proposed algorithm uses the thread model to reduce over-dataflow between the processing units (PUs). The computation cost of this design comprises the initial memory access to fetch image blocks and inter-processor data communication, the second can be neglected since the use of shared global main memory. Fig. 3 depicts the distribution of JPEG processing among available PUs and the main memory.

To fully utilize the parallel resources, each PU receives the same number of data blocks to process. In case of small image dimension, we set a threshold (N_{th}) to limit the number of PUs through calculating the expected execution by setting $N = m_{i+1}$ where $m_{i+1} > m_i$. According to Amdahl's law [13], the maximum speedup that can be achieved in a parallel system is expressed as:

$$S_{max} = \frac{1}{f_s + \frac{f_p}{N}} \quad (5)$$

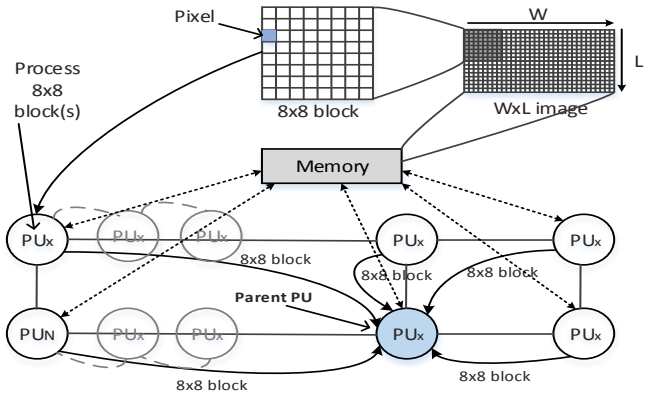

Fig. 3. Intercommunication between PUs

Where f_s, f_p are the serial and parallel fractional times respectively and $f_s, +f_p = 1$. The calculation of the next expected time T_{i+1} for m_{i+1} processors are expressed in (6) through (9):

$$f_p = 2\frac{S_p - 1}{S_p} \tag{6}$$

$$f_p = \frac{S_n - S_m}{\left(1-\frac{1}{n}\right)S_n - \left(1-\frac{1}{m}\right)S_m} \tag{7}$$

$$S_{m_i+1} = \frac{1}{\frac{fp}{m_{i+1}} + (1-f_p)} \tag{8}$$

$$T_{i+1} = \frac{T_1}{S_{m_i+1}} \tag{9}$$

Where n, m are PUs count whose speedups are known and $n > m$. The formula in (6) is not applicable when S_p is superlinear (i.e. $S_p > P$). Efficiency (η) is another important factor that contributes in determining the optimum number of PUs. The optimum number of PUs (N_{opt}) is measured when S and η are maximal and is set when the efficiency falls in a specific range that limits the minimum of accepted value. In this paper, we set the limit to [0.8, 0.99]. Based on the result obtained from (8), we calculate η_{i+1} using (10) and check against the following two cases:

$$\eta_{i+1} = \frac{S_{m_i+1}}{m_{i+1}} \tag{10}$$

Case 1) $\eta_{i+1} > \eta_i$: efficiency is improving, we increase the current value of m_i and let $N_{th} = m_{i+1}$.

Case 2) $\eta_{i+1} \leq \eta_i$: efficiency is either decreased or not improved: If $\eta_{i+1} \in [0.8, 0.99]$, then $N_{th} = m_{i+1}$. Otherwise, $N_{th} = m_i$.

C. Cross-Architectural Design

The design of the algorithm guarantees cross-compatibility on GPU and SMD systems. While GPU optimizes the instruction-level parallelism using an intensive number of PUs, in some problems, where data dependency (data overlapping) exists, parts of non-parallelizable tasks are left to the CPU, adding more latency and overhead in the design. The design also provides cross portability across different hardware since the use of common APIs and C-libraries. In Fig. 4, we illustrate the proposed algorithm and show the cross-compatibility on a sample 24-bit color image.

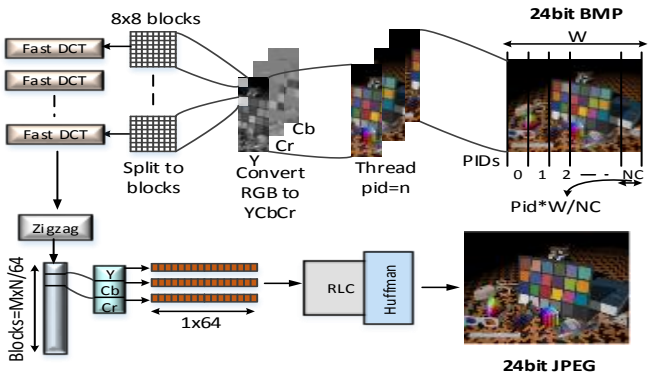

Fig. 4. Algorithm stages for SMP and GPU on vertical split image

IV. Performance Evaluation

Previous works on GPU evaluations showed that the GPU has better performance and higher speedup on selected problems. The evaluations were based on comparing high efficiency GPU with non-optimized multi-processor or multi-core systems. In this paper, we make a fair estimation of fully optimized GPU and an out-of-order optimized SMP architecture that crucially exploits the instruction-level parallelism and underlying hardware.

In this section, we present the experimental performance and the simulation environment and provide the performance metrics. The simulation results are presented in section V.

A. Systems Specifications

The simulations have been carried on CUDA GPU and SESC. Table 1 describes the specifications of both systems.

TABLE I. Evaluation Environment Specs

	System	
	SMD	***GPU***
Architecture	Symmetric[a]	Symmetric[a] grid
Technology	70nm	40nm
Memory	256-bit shared	128-bit global
Processor clock	5GHz	1.4GHz
int, fp registers	128-bit	64-bit
NP, N	[1-32]	96

[a.] Identical processors and GPU cores. [b.] One PU reserved for rendering output.

B. Evaluation Metrics

The number of processors is the most effective factor and cost in speeding up JPEG algorithm, a number of metrics to evaluate the performance are used in this paper as described below:

1. *Speedup (S)*: the improvement of parallel code in the range of [0, N] or [0, NP].

2. *Efficiency (η):* the average speedup of each PU in range of [0, 1].
3. *Efficiency cutoff point (ECP):* the highest number of (N or NP) at which $\eta > 50\%$.
4. *Utilization (U):* the percentage of available utilized resources $U = m_i/NP$ *or* $U = m_i/NP$.
5. *Scalability:* how η maintains constant when input size and (N or NP) increase.
6. *Amdahl's law (S_{max}).*
7. *Karp-Flatt:* define the sequential fraction f_s as in (11). The less the value f_s the better the performance. Let $P = N$ *or* NP, then:

$$f_s = \frac{P-S}{S(P-1)} \tag{11}$$

V. Experimental Results

We quantified our proposed parallel JPEG algorithm using 24-bit BMP images available in [14]. For each image, we run the proposed algorithm on 1, 2, 4, 8, 16 and 32 processors in SESC while, in GPU, number of cores is fixed to 96. We exhaustively verified the accuracy and evaluated the performance metrics listed in section IV-B.

We start analyzing the performance of the proposed algorithm according to f_p and η. The fractional value f_p is proportional to performance. Fig. 5 shows the theoretical fractional f_p values calculated for all images for SMP and GPU. The expected execution time, according to (9) are clarified in Fig. 6 while the efficiency η and the ECP are depicted in Fig.7.

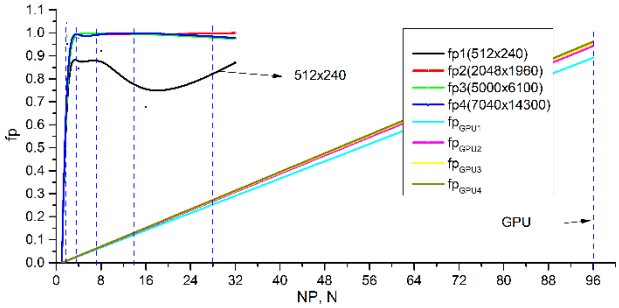

Fig. 5. The theoretical f_p values for 4 BMP images

As shown in Fig. 6, T_{i+1} is a decreasing function of NP and the actual running time is smaller than T_{i+1} mostly for all images, thus, the expected speedup is less than what we achieved in the simulation for both SMP and GPU.

Referring to Fig. 7, ECP_1 represents the cutoff point for image-1 (512x240) and P_{opt3} denotes the optimal number of processors at which η in its highest range and S is very close to NP in SMP. From Fig. 6, we can prove that P_{opt} is not necessary the same as ECP. For image-4 (7040x14300), the value of f_p And η in accordance with Fig.5 and Fig.7 respectively, are greater than 100% for NP = 2 and 16 (normal values fall in [0, 1]), this happens only when S > NP (i.e. Superlinear speedup). Based on our simulation, superlinear speedup occurred because each CPU is working on a smaller set of data in the memory (shared memory), the data handled by any CPU fits better in the cache. In contrast, we cannot have superlinear speedup in case of GPU simulation as the values of f_p we measured were marginally small.

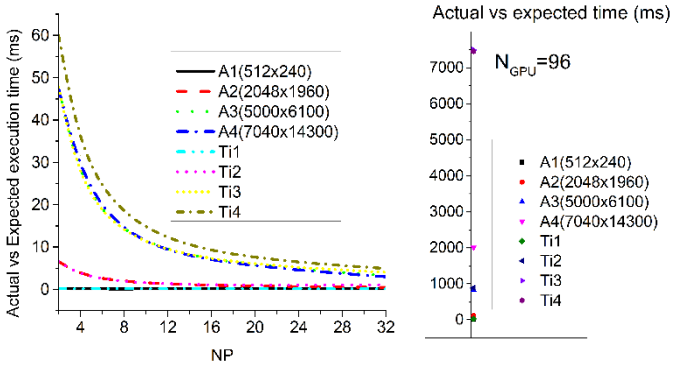

Fig. 6. Encoding times and T_{i+1} values in SMP and GPU simulation

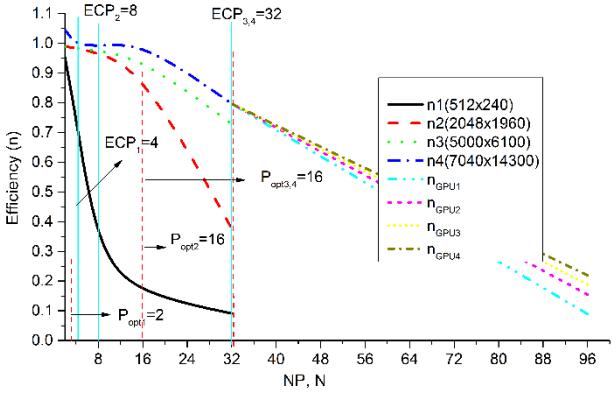

Fig. 7. The efficiency, ECP and the optimal number of processors

Fig. 8 depicts the speedup achieved on a variety sizes of images, when applying the JPEG algorithm on the small images (like 512x240), the execution time increases because creating and synchronizing between the running threads contribute in extra overhead time as well as startup and termination times, thereby, the serial code will clearly perform better.

Evaluating the JPEG algorithm in the GPU is much easier since all PUs are involved in the computation (96 cores). The evaluation metrics for the same sample images are summarized in Table 2 where T_S and T_P denotes the serial and parallel execution time in millisecond respectively.

Table 2 shows that the performance boost in the GPU is not significant even for big f_p values in JPEG algorithm. This is due to the environment startup overhead. Also, based on the performance metrics used in this paper, the efficiency limits P_{opt} to one (i.e. Serial code). However, our simulation showed that the GPU is not the optimal solution and that SMP (CPU) outperformed GPU by 18%.

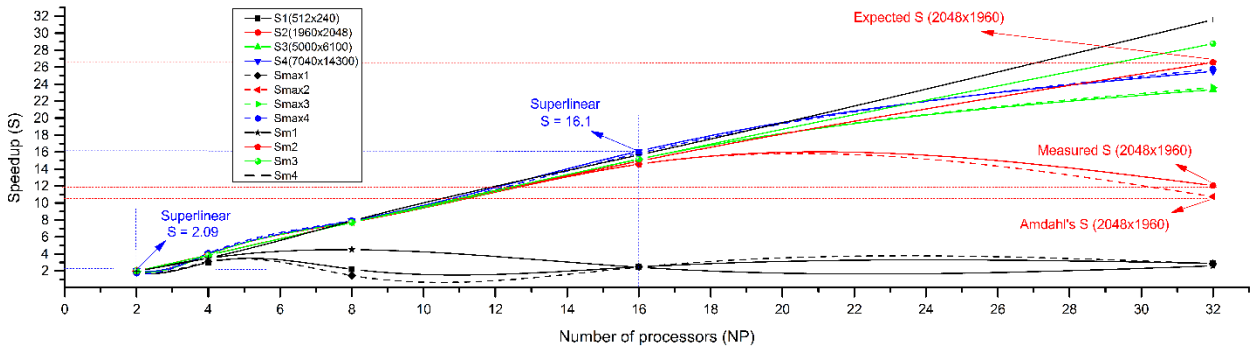

Fig. 8. Runtime speedup for 4 24-bit BMP images on multiple NP in SMP architecture

TABLE II. GPU SIMULATION RESULT

	24-bit BMP images			
	~369KB 512x240	**~12MB 1960x2048**	**~92MB 5000x6100**	**~302MB 7040x14300**
T_S	64	1764	14832	42460
T_P	7.500934	118.55	820.15	2020.82
T_{i+1}	32	891	7495	7435
S	8.53	14.88	18.08	21.01
η	8.89%	15.50%	18.84%	21.89%
ECP	1	1	1	1
f_p	89.21%	94.26%	95.46%	96.24%

The two remaining metrics (utilization and scalability) can be inferred from the evaluation results. The GPU was fully utilized (U = 100%) while in SESC, the utilization varied from 25% to 100%. The algorithm in SESC showed a scalable comportment as depicted in Fig. 8, where the speedup increased as the image size increased, whereas, in the GPU, the speedup increased but with lower system efficiency.

VI. CONCLUSION

We have presented two efficient designs for JPEG encoding for 24-bit BMP images. We could also fairly compare the performance of the GPU with an optimized SMP architecture in more realistic context based on a variety of evaluation metrics. Our simulation yielded in significant speedups in both parallel CPU and GPU executions and showed that CPU performed the proposed JPEG algorithm much better.

For future work, emphasis will be on further enhancement on the implementation of the JPEG algorithm. It can be enhanced using thrust, which develops high-performance parallel application and offering high device efficiency at low processing cost. The algorithm latency time can be reduced by using a shared memory per thread block in GPU instead of the GPU global memory (shared memory, facilitate global memory coalescing and no bank conflict between threads).

REFERENCES

[1] Jan Kwiatkowski, Radoslaw Iwaszyn, "Automatic Program Parallelization for Multicore Processors," in VLDB '06: Proceedings of the 32nd international conference on Very large data bases (2006), pp. 1219-1222.

[2] Kim, Bumho; Lee, Jeong-Woo; Yoon, Ki-Song, "A parallel implementation of JPEG2000 encoder on multi-GPU system," Advanced Communication Technology (ICACT), 2014 16th International Conference on , vol., no., pp.610,613, 16-19 Feb. 2014. doi: 10.1109/ICACT.2014.6779033.

[3] Garg, M. "Performance Analysis of Chrominance Red and Chrominance Blue in JPEG," World Academy of Science, Engineering and Technology, International Science Index 19, (2008), 2(7), 105 - 108..

[4] Liu Duo; Fan Xiao Ya, "Parallel program design for JPEG compression encoding," Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference, pp.2502,2506, 29-31 May 2012. doi: 10.1109/FSKD.2012.6234221.

[5] Le, R.; Mundy, J.L.; Bahar, R.I., "High Performance Parallel JPEG2000 Streaming Decoder Using GPGPU-CPU Heterogeneous System," Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference, pp.16,23, 9-11 July 2012. doi: 10.1109/ASAP.2012.34.

[6] Richter, T.; Simon, S., "On the JPEG 2000 ultrafast mode," Image Processing (ICIP), 2012 19th IEEE International Conference, pp.2501,2504, Sept. 30 2012-Oct. 3 2012. doi: 10.1109/ICIP.2012.6467406.

[7] Datla, S.; Gidijala, N.S., "Parallelizing Motion JPEG 2000 with CUDA," Computer and Electrical Engineering, 2009. ICCEE '09. Second International Conference on , vol.1, no., pp.630,634, 28-30 Dec. 2009. doi: 10.1109/ICCEE.2009.277.

[8] Jesse D. Kornblum. "Using JPEG quantization tables to identify imagery processed by software," Digital Investigation, Volume 5, Supplement, September 2008, Pages S21-S25, ISSN 1742-2876.

[9] Pieters, B.; De Cock, J.; Hollemeersch, C.; Wielandt, J.; Lambert, P.; Van de Walle, R., "Ultra High Definition video decoding with Motion JPEG XR using the GPU," Image Processing (ICIP), 2011 18th IEEE International Conference on , vol., no., pp.377,380, 11-14 Sept. 2011. doi: 10.1109/ICIP.2011.6116527.

[10] Website. "What is gpu accelerated computing," Internet: http://www.nvidia.com.

[11] Website. "Joint photographic experts group," Internet: http://www.jpeg.org.

[12] Y. Arai, T. Agui, and M. Nakajima, "A fast DCT-SQ scheme for images," Trans. of the IEICE, vol. E-71, no. 11, pp. 1095-1097, Nov. 1988.

[13] Hill, M.D.; Marty, M.R., "Amdahl's Law in the Multicore Era," Computer , vol.41, no.7, pp.33,38, July 2008, doi: 10.1109/MC.2008.209.

[14] Website. "Lossless compression software for camera raw images, " Internet: http://www.imagecompression.info/.

[15] Pablo O. Paul S. Website. "Sesc: superscalar simulator," Internet: http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/.