JPEG Encoding on Fine-Grain Manycore Platforms

By

THOMAS ABBOTT
THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL AND COMPUTER ENGINEERING

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Bevan M. Baas, Chair

Professor Hussain Al-Asaad

Professor Venkatesh Akella

Committee in Charge

2023

Abstract

JPEG encoding is a powerful image compression algorithm capable of compressing image data at the cost of image quality. A variety of architectures implement JPEG encoding, each leveraging either serial execution superiority (general-purpose programmable processors), massive parallelization abilities (GPUs), or dynamic architecture arrangements (FPGAs). However, all these architectures need help to simultaneously handle the serial and parallel components of the JPEG encoding algorithm. This thesis proposes 29 JPEG encoder implementations on the KiloCore platform (a fine-grain manycore processor array), compares each algorithm to one another, and compares the top algorithms to designs on differing architectures.

This work benchmarks throughput, throughput per area, energy per megapixel encoded, and energy-delay product across 29 KiloCore JPEG encoder versions. Furthermore, this work compares the top KiloCore designs against JPEG implementations on a Xilinx Zynq-7000 FPGA (VISENGI), TI C66x Embedded Processor, Intel i9 9900 CPU (libjpeg-turbo), and Intel Platinum 8168 with an Nvidia A100 GPU (nvJPEG).

JPEG encoding implementations on KiloCore require low amounts of energy while still reaching competitive throughput. JPEG encoding implementations on KiloCore achieve higher throughput than the C66x and Intel i9 9900 JPEG encoders by at least 6.6×. JPEG encoding implementations on KiloCore have the lowest area usage and have the highest throughput per area by 1.45× to 100×. JPEG encoding implementations on KiloCore have the lowest energy per megapixel encoded of tested general-purpose processors, by 1.88× to 100×. Finally, JPEG encoding implementations on KiloCore boast a 20× to 261,733× lower energy-delay product than its general-purpose industry competition.

Acknowledgments

Thank you to my advisor and mentor, Professor Baas. His mentorship through my undergraduate and graduate education inspired me further to apply myself to your digital design and VLSI classes. Furthermore, his guidance in my research helped me stay motivated and focused.

Thank you to Dr. Bohnenstiehl for his work on KiloCore and for guiding me in my development using the KiloCore software toolchain.

Thank you to the previous graduate students of the VCL labs whose documentation helped guide me through my thesis.

Thank you, Professor Akella and Professor Al-Assad, for your time and effort in reviewing my thesis.

Thank you, Derek Li, for assisting with finding performance metrics of competing designs. Thank you, VISENGI, for providing performance data of your IP block upon request.

Most importantly, I thank my family, friends, and partner for their support throughout my academic journey. It would not have been possible without any one of you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

JPEG Encoding is a unique algorithm that has both serial and parallel components. Innovations in GPUs, CPUs, and FPGAs have all contributed to higher efficiency in JPEG encoding, but each architecture contains disadvantages. KiloCore represents a unique architecture that can simultaneously take advantage of the inherent parallelism in JPEG encoding's DCT-II and quantization steps while excelling at the serial tasks of encoding and bitstream combinations.

JPEG encoding assists with the compression of images with minimal loss of quality. However, with high-fidelity images, current JPEG encoders can take orders of magnitudes longer than lower-resolution images. GPU solutions like nvJPEG can address the throughput problem. However, they use a wasteful amount of energy and area to accomplish the simple task. Many-core processor arrays can fill the gap between having a competitive throughput and not using an overwhelming amount of energy and area on a given chip. Furthermore, JPEG encoding occurs mainly in video and photography editing circumstances, so users who are not videographers or photographers likely prefer to use their silicon real estate differently. Hence, fine-grain manycore processor's programmability allows the area to adapt to other needs (where an ASIC or hardware accelerator would take up space).

In the grander scheme, video and photo encoding and decoding are growing more popular with social media websites and streaming services. Therefore, a chip that can reprogram itself to

accelerate a given codec will be precious, and this thesis is one step closer to fine-gran manycore processors like KiloCore filling this niche.

1.2 Thesis Organization

- The remainder of this thesis is organized as follows:
- Chapter 2 outlines the JPEG specification and relevant algorithms for color space conversion, the DCT-II, quantization, run-length encoding and Huffman encoding.
- Chapter 3 reviews relevant KiloCore chip architectural information.
- Chapter 4 introduces the JPEG encoding algorithms implemented.
- Chapter 5 showcases the JPEG encoding algorithms simulation results, including throughput, area, energy per megapixel encoded, throughput per area, and energy-delay product.
- Chapter 6 compares the most competitive KiloCore implementations with general-purpose processors, GPUs, and FPGAs.
- Chapter 7 summarizes the thesis and provides starting points for future work.

Chapter 2

Background of JPEG Encoding

2.1 Overview

The JPEG encoding algorithm exploits photos' low-frequency nature to compress image data without sacrificing too much image quality [1]. Therefore, the first step in the algorithm is to convert the color space of the image data (likely RGB) to YCbCr. This transform intends to consolidate more information in one channel (the luminance or Y channel) instead of evenly spreading across multiple channels as RGB does. Consequently, the Cb and Cr channels can compress into much smaller sizes as they aid the visual fidelity of the image much less than the Y channel.

Images are transformed to the frequency domain using the 2D discrete cosine transform (DCT-II). For the average realistic photo, it is much more common for high-frequency components of an image to be near zero. Like how the color transformation consolidates more information into one channel, the DCT-II consolidates the information in an 8-by-8 block to low-frequency elements.

The non-integer output of the DCT-II is quantized to allow for the binning of similar elements. Finally, to prioritize low-frequency parts of the DCT-II matrix, quantization tables provide weight to corresponding regions of the matrix.

The low-frequency parts of an 8-by-8 block after DCT-II (and quantization) are toward the top-left of the matrix, and the highest-frequency details are toward the bottom-right.

Reorganizing the order of the matrix to go from low frequency to high frequency is called zigzagging.

Next, the zigzagged AC output is run-length encoded by counting how many zeros have come before the next value. Consequently, runs of zeros are consolidated out of the data to save space. Furthermore, the size of the non-zero value encountered prevents any extra bits from being used to specify a smaller value (i.e., the value three needs only two bits while twenty-one needs five bits). Finally, the first element in the block is the DC coefficient, and it is difference-encoded with the DC coefficient of the previous block in the same channel (the last value is 0 for the first block).

Finally, the zero count and size pairs are Huffman encoded to save space further. Huffman encoding allows more frequent pairs to take less space (pairs with low zero counts or sizes), while less frequent pairs (pairs with large zero counts or sizes) may take more space. Statistically, this tradeoff compresses data further, depending on what Huffman table encodes the pairs.

Finally, the process is repeated across all 8x8 blocks of an image and then paired with a header that contains information about the settings of a given JPEG file. The JPEG specification defines multiple possible JPEG encoding methods, but this thesis covers baseline, Huffman, 8-bit color, lossy, and 4:4:4 JPEG encoding. Alternative modes are outlined below; however, the presented implementations do not support them.

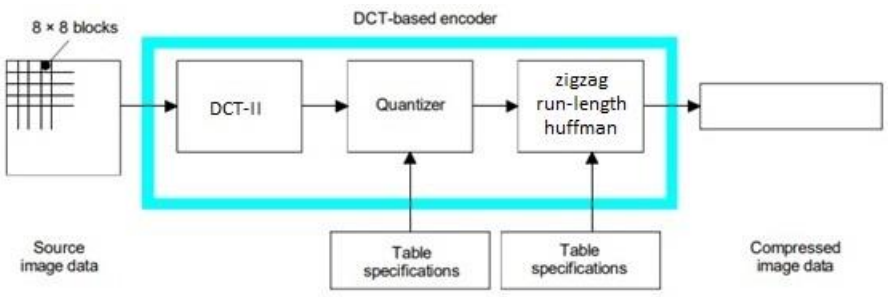


Figure 2.1: JPEG encoding diagram [1]

2.2 Baseline and Progressive Formats

2.2.1 Baseline Format

A baseline formatted JPEG file encodes its 8-by-8 blocks one at a time, starting in the top left corner of the image, traveling right, and then wrapping to the next row [1]. When decoding the image, each row appears one at a time. When internet bandwidth is limited, or many photos must appear quickly at once, loading the image this way can be disadvantageous. In those cases, a progressive JPEG format is preferred; however, our architecture implements only a baseline format.

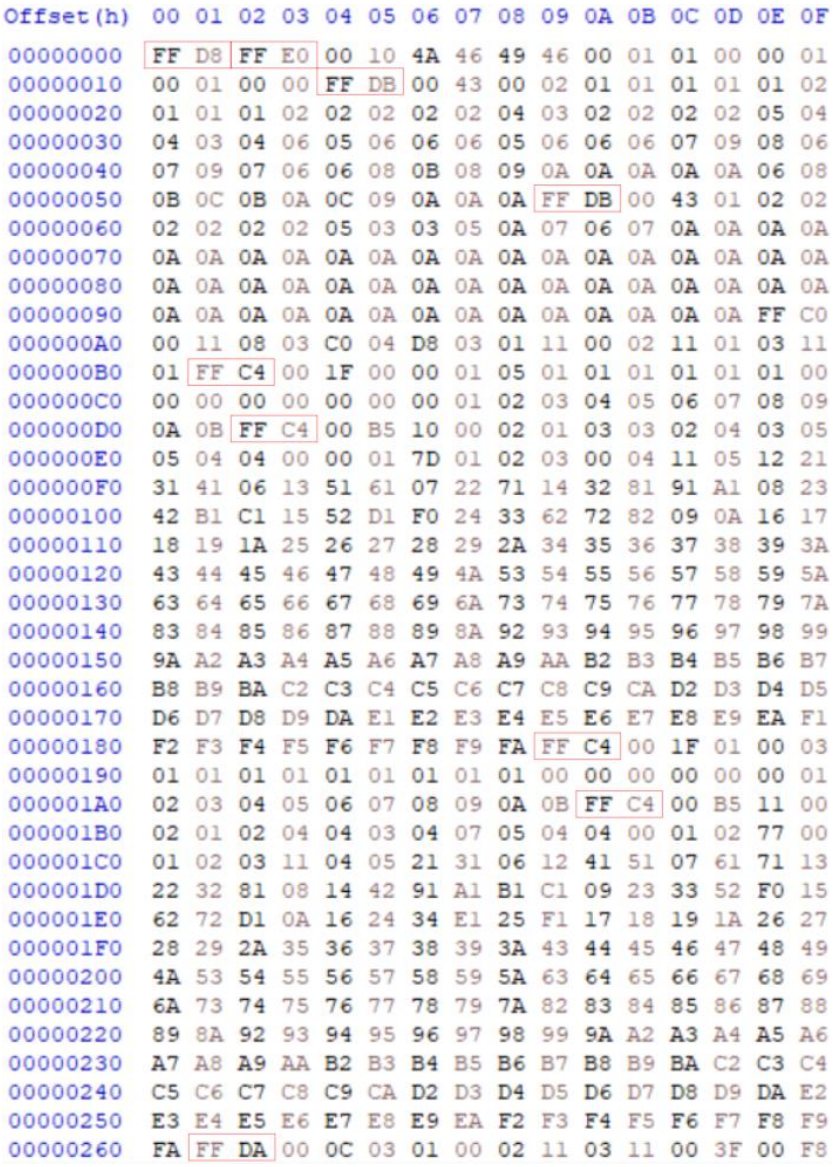
2.2.2 Progressive Format

Progressive JPEG formats allow the decoder to determine the whole image in low fidelity on its first pass and then build image quality as the number of passes increases [1]. This can be advantageous for high-resolution images that are decipherable at lower resolutions. In addition, websites with multiple high-resolution images needing to load immediately should leverage progressive JPEGs; however, the benefit of a progressive format is not so clear when the internet speed is too fast for the baseline load time to matter. The additional processing cost of encoding and decoding a progressive JPEG can arguably make it the lesser of the two options in the case of a high-speed internet connection.

2.3 Header Organization

JPEG reserves unique two-byte tags to denote specific settings for the decoder of the given JPEG file [1]. Typically, these are at the top of the binary file, then the encoded data follows, and finally, it terminates with an end-of-image tag. There are 623 bytes encoded in a standard JPEG header to store the size, quantization tables (2), Huffman tables (2), color subsampling mode, color depth, greyscale information, a general description, and more. Most encoders can replicate this header before each image, changing only the picture size and the quantization tables (which adjust with the quality level of the encoding). Some encoders use a Huffman table customized to the specific photo's data for the highest possible compression. In this case, it also needs to be encoded differently on each image produced.

In Figure 2.2, a sample JPEG header is provided, where each two-byte tag is boxed in red. Following each tag is data is typically the size of the data to follow, and the information relevant to said tag. For example, 0xFFDB is followed by 0x0043 which denotes that the information to follow will be 67 bytes (excluding the original tag, but including the size), and then either 0x00 or 0x01 is encoded to denote which quantization table is being stored. The next 64 bytes includes the 64 values present in the given quantization table. A similar formatting applies for most tags, and more information is outlined in the JPEG specification [1].



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	FF	D8	FF	E0	00	10	4A	46	49	46	00	01	01	00	00	01
00000010	00	01	00	00	FF	DB	00	43	00	02	01	01	01	01	01	02
00000020	01	01	01	02	02	02	02	02	04	03	02	02	02	02	05	04
00000030	04	03	04	06	05	06	06	06	05	06	06	06	07	09	08	06
00000040	07	09	07	06	06	08	0B	08	09	0A	0A	0A	0A	0A	06	08
00000050	0B	0C	0B	0A	0C	09	0A	0A	0A	FF	DB	00	43	01	02	02
00000060	02	02	02	02	05	03	03	05	0A	07	06	07	0A	0A	0A	0A
00000070	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A
00000080	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A
00000090	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	FF	C0
000000A0	00	11	08	03	C0	04	D8	03	01	11	00	02	11	01	03	11
000000B0	01	FF	C4	00	1F	00	00	01	05	01	01	01	01	01	01	00
000000C0	00	00	00	00	00	00	00	01	02	03	04	05	06	07	08	09
000000D0	0A	0B	FF	C4	00	B5	10	00	02	01	03	03	02	04	03	05
000000E0	05	04	04	00	00	01	7D	01	02	03	00	04	11	05	12	21
000000F0	31	41	06	13	51	61	07	22	71	14	32	81	91	A1	08	23
00000100	42	B1	C1	15	52	D1	F0	24	33	62	72	82	09	0A	16	17
00000110	18	19	1A	25	26	27	28	29	2A	34	35	36	37	38	39	3A
00000120	43	44	45	46	47	48	49	4A	53	54	55	56	57	58	59	5A
00000130	63	64	65	66	67	68	69	6A	73	74	75	76	77	78	79	7A
00000140	83	84	85	86	87	88	89	8A	92	93	94	95	96	97	98	99
00000150	9A	A2	A3	A4	A5	A6	A7	A8	A9	AA	B2	B3	B4	B5	B6	B7
00000160	B8	B9	BA	C2	C3	C4	C5	C6	C7	C8	C9	CA	D2	D3	D4	D5
00000170	D6	D7	D8	D9	DA	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	F1
00000180	F2	F3	F4	F5	F6	F7	F8	F9	FA	FF	C4	00	1F	01	00	03
00000190	01	01	01	01	01	01	01	01	01	00	00	00	00	00	00	01
000001A0	02	03	04	05	06	07	08	09	0A	0B	FF	C4	00	B5	11	00
000001B0	02	01	02	04	04	03	04	07	05	04	04	00	01	02	77	00
000001C0	01	02	03	11	04	05	21	31	06	12	41	51	07	61	71	13
000001D0	22	32	81	08	14	42	91	A1	B1	C1	09	23	33	52	F0	15
000001E0	62	72	D1	0A	16	24	34	E1	25	F1	17	18	19	1A	26	27
000001F0	28	29	2A	35	36	37	38	39	3A	43	44	45	46	47	48	49
00000200	4A	53	54	55	56	57	58	59	5A	63	64	65	66	67	68	69
00000210	6A	73	74	75	76	77	78	79	7A	82	83	84	85	86	87	88
00000220	89	8A	92	93	94	95	96	97	98	99	9A	A2	A3	A4	A5	A6
00000230	A7	A8	A9	AA	B2	B3	B4	B5	B6	B7	B8	B9	BA	C2	C3	C4
00000240	C5	C6	C7	C8	C9	CA	D2	D3	D4	D5	D6	D7	D8	D9	DA	E2
00000250	E3	E4	E5	E6	E7	E8	E9	EA	F2	F3	F4	F5	F6	F7	F8	F9
00000260	FA	FF	DA	00	0C	03	01	00	02	11	03	11	00	3F	00	F8

Figure 2.2: Sample JPEG header with tags outlined in a red box

2.4 Lossy and Lossless Formats

2.4.1 Lossy Format

A lossy JPEG is the most common format of a JPEG, as loss allows for the most amount of compression [1]. Loss occurs during the DCT-II phase, where the nature of computing the DCT-II incurs a rounding error. The quantization stage amplifies this loss because numbers are

scaled and rounded, leading to easily compressible data but with lower fidelity. This work focuses on the lossy format, the de facto standard for JPEG encoding.

2.4.2 Lossless Format

JPEGs also come in the lossless variety to compete with PNG and RAW file formats [1]. The JPEG lossless format skips DCT-II and quantization and instead jumps into a predictor to further compress the image before Huffman or Arithmetic encoding.

2.5 Color Spaces and Subsampling

2.5.1 RGB to YCbCr Transformation

Transforming RGB to YCbCr is often necessary to create a JPEG image; the JPEG standard lists equations for the calculation to better use the YCbCr format [2]. Typically, the YCbCr format has both headroom and footroom, meaning the range of Y is from 16 to 235, and the range of CbCr is from 16 to 240. This headroom is useful in other digital formats, but for JPEG encoders, it lowers the possible color depth of the signal. Therefore, the footroom of all the signals is subtracted from the value and then scaled to take up the data's full 8-bit or 12-bit space. Equations 2.1, 2.2, and 2.3 show the conversion after headroom and footroom have both been removed [2]. Although there is a 12-bit JPEG format, this work covers only 8-bit JPEG encoding. Finally, each number is level shifted such that 0 is the mean value; in the 8-bit case, this means subtracting 128 from the value and storing it in 2's complement form.

$$Y = 0.299R + 0.587G + 0.114B \tag{2.1}$$

$$Cb = -0.1687R - 0.3313G + 0.5B + 128 \tag{2.2}$$

$$Cr = 0.5R - 0.4187G + 0.0813B + 128 \tag{2.3}$$

2.5.2 Color Subsampling

One channel's information can be prioritized using color subsampling. Typically, JPEG encoders use a 4:2:2 or a 4:4:4 color subsampling mode. 4:4:4 takes no preference as to what channel to collect, whereas 4:2:2 skips every other CbCr value as the Y channel has a much more substantial impact on the given image's fidelity. Equations 2.4 and 2.5 show how to compute the given column or row (x_i or y_i) given the max column or row (X or Y), the sampling factor (H_i or V_i), and the maximum sampling factor across each color component (H_{max} or V_{max}) [1]. This work explores the 4:4:4 format as it creates the most work for encoding.

$$x_i = \left\lceil X \times \frac{H_i}{H_{max}} \right\rceil \tag{2.4}$$

$$y_i = \left\lceil Y \times \frac{V_i}{V_{max}} \right\rceil \tag{2.5}$$

2.6 2D Discrete Cosine Transform

2.6.1 Series Definition of the DCT-II

The series definition of DCT-II is given in Equation 2.6 [1]. A DCT-II transform and a 2D DCT-II transform are not the same thing. DCT-II is a specific DCT type corresponding to the DFT of 4N real inputs. A 2D DCT-II is when one performs the DCT-II across all the rows or columns of a matrix and vice versa. Equations 2.6 through 2.7 show a 2D DCT-II for 8-by-8 dimensional matrices.

$$S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16} \tag{2.6}$$

where

$$C_u,\ C_v = \begin{cases} \frac{1}{\sqrt{2}} & u, v = 0 \\ 1 & otherwise \end{cases} \tag{2.7}$$

2.6.2 Matrix Transformation Definition of the DCT-II

Equations 2.8 and 2.9 give the matrix transformation definition of the DCT-II [3]. Equation 2.8 defines building the 8-by-8 transformation matrix where p is the column index and q is the row index, and Equation 2.9 shows how to apply the transform to an arbitrary 8-by-8 block Λ. The benefit of this equation over 2.6.1's series representation is the possible parallelism to exploit in the matrix multiplication and the no need to take the cosine of any element. This work avoids costly cosine calculations. The matrix transformation definition contains 1024 multiplies and 896 additions, resulting in two 8-by-8 matrix multiplications. When using a GPU, using this parallelizable algorithm may be beneficial.

$$T_{pq} = \begin{cases} \dfrac{1}{2\sqrt{2}} & p = 0, 0 \le q \le 7 \\ \dfrac{1}{2}\cos\dfrac{\pi(2q+1)p}{16} & 1 \le p \le 7, 0 \le q \le 7 \end{cases} \tag{2.8}$$

$$A = T \times \Lambda \times T^{*} \tag{2.9}$$

2.6.3 AA&N Algorithm to Compute the DCT-II

The final algorithm to compute the DCT-II is Arai, Agui, and Nakajima's (AA&N) Algorithm, denoted in Figure 2.3 [4]. It collapses the total operations of calculating a 2D DCT-II to only 144 multiplies and 464 additions. Also, 64 of these multiples are with the quantization step of the JPEG algorithm meaning the actual computational cost is closer to 80 multiplies. Figure 2.3 is the DCT butterfly diagram of the AA&N algorithm. Shaded dots denote addition, arrows indicate negation, and boxes with constants represent multiplication by the constant in the box. Since the algorithm is computationally much faster than the method discussed in 2.6.2, JPEG encoders prefer it.

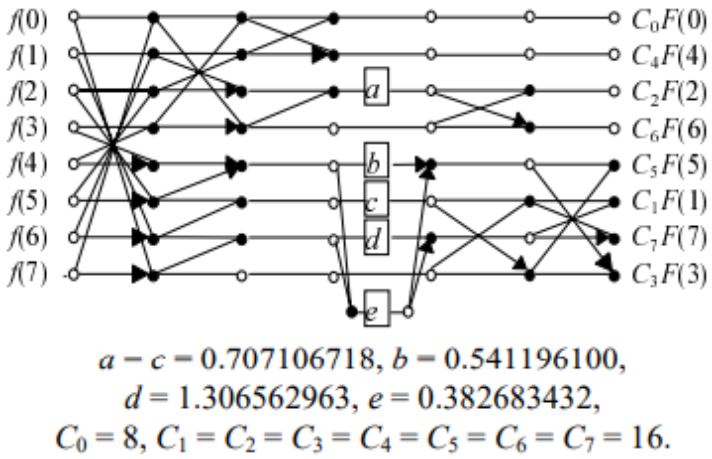


Figure 2.3: AA&N DCT algorithm [4]

2.6.4 Precision Considerations

Each method of computing the DCT-II ends up with close to the same result when using double floating-point precision; however, in situations where accuracy is limited, they will end up with slightly different results. Rounding error does not imply that the previous equations are approximations but rather is a result of computing with non-integer values. The JPEG standard explicitly mentions this and purposefully does not put a strict precision requirement on the DCT-II method to allow for further innovations in the calculation.

2.7 Quantization

2.7.1 Overview

Elements of a given 8-by-8 block are scaled-down and quantized. The encoder specifies quantization tables, typically scaled down or up depending on the identified quality factor. The quantization table gives inherent weights to certain parts of the 8-by-8 blocks over another. Usually, these weights are given to the lower frequency values, although that is not a requirement. A given JPEG specifies two quantization tables; typically, one is used for the Y channel, while the other is for the CbCr channels. The JPEG format requires an accurate

rounding method after the element-wise division (Equation 2.11) [1]. Quantization tables are embedded in the header of a given JPEG file to allow a future decoder to reverse the process. Quantization tables can become intellectual property, and in this work, we use the sample quantization tables provided by the JPEG standard and scale them appropriately with a quality factor. Tables 2.1 and 2.2 contain the JPEG standard's quantization tables [1].

$$Sq_{vu} = \text{round}\left(\frac{S_{vu}}{Q_{vu}}\right) \tag{2.10}$$

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

Table 2.1: JPEG standard T.81's sample luminance quantization table [1]

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Table 2.2: JPEG standard T.81's sample chrominance quantization table [1]

2.7.2 Quality and Quality Factor (QF)

Scaling the quantization table of an encoder can change the quality and compression ratio of the resultant image. The scale factor applied to the quantization table is called the quality factor (QF), and this work focuses on profiling performance when the quality factor is equal to 0.1 which typically provides a 10:1 compression ratio in an image. Smaller quantization factors create more work for encoding steps following quantization, similarly larger quality factors result in less work for encoding steps following quantization.

2.8 Zigzag

After quantizing a matrix, the elements are read in a zigzag fashion (Figure 2.4) to increase the likelihood of a string of zeros [1]. The operation exploits the fact that there is a significant chance of higher frequency elements of a given 8-by-8 being zero, allowing for more effective compression.

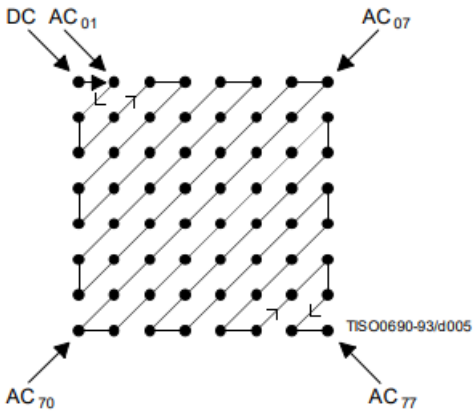


Figure 2.4: Zigzag order [1]

2.9 AC Coefficient Run-length Encoding

In JPEG, runs of consecutive zeros are run-length encoded. The zigzagged block is converted to a series of bytes [1]. In a given byte, the top 4 bits specify the length of the run of

zeros, and the bottom 4 bits are the size of the nonzero value in bits (3 would be 2 bits in size, for example). In the event of a negative number, JPEG encodes value without its sign bits in 1's complement format. There are two special bytes reserved, 0xf0 and 0x00. 0xf0 denotes a string of 16 zeros with no value to encode after it; this is a ZRL. No more nonzero values are left in the block when 0x00 is encoded. For this reason, 0x00 is the EOB or end-of-block signal, and there is never a value coded after it. If the last element of an 8-by-8 block is nonzero, an EOB signal is not encoded. The first value in an 8-by-8 block is the DC coefficient, which is not run-length encoded.

2.10 DC Coefficient Difference Encoding

The first element of an 8-by-8 block is the DC coefficient (Figure 2.5), and it is encoded differently than the following AC coefficients [1]. The DC coefficient is difference-encoded with the previous block's value in the same channel, as described in Equation 2.12 (where *PRED* is the DC coefficient of the previous block and DC_i is the coefficient of the current block) [1]. The first DC coefficient is encoded as is, given there is no previous block. DC components between blocks should be similar in magnitude, and thus difference encoding should leave a smaller value to encode into the JPEG. Difference-encoding breaks the block-level parallelism present in the previous steps.

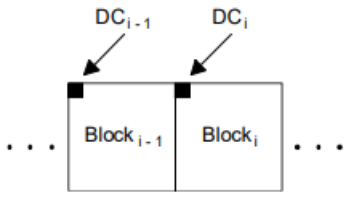


Figure 2.5: Differential DC encoding [1]

$$DIFF = DC_i - PRED \tag{2.11}$$

2.11 Huffman and Arithmetic Encoding

2.11.1 Huffman Encoding

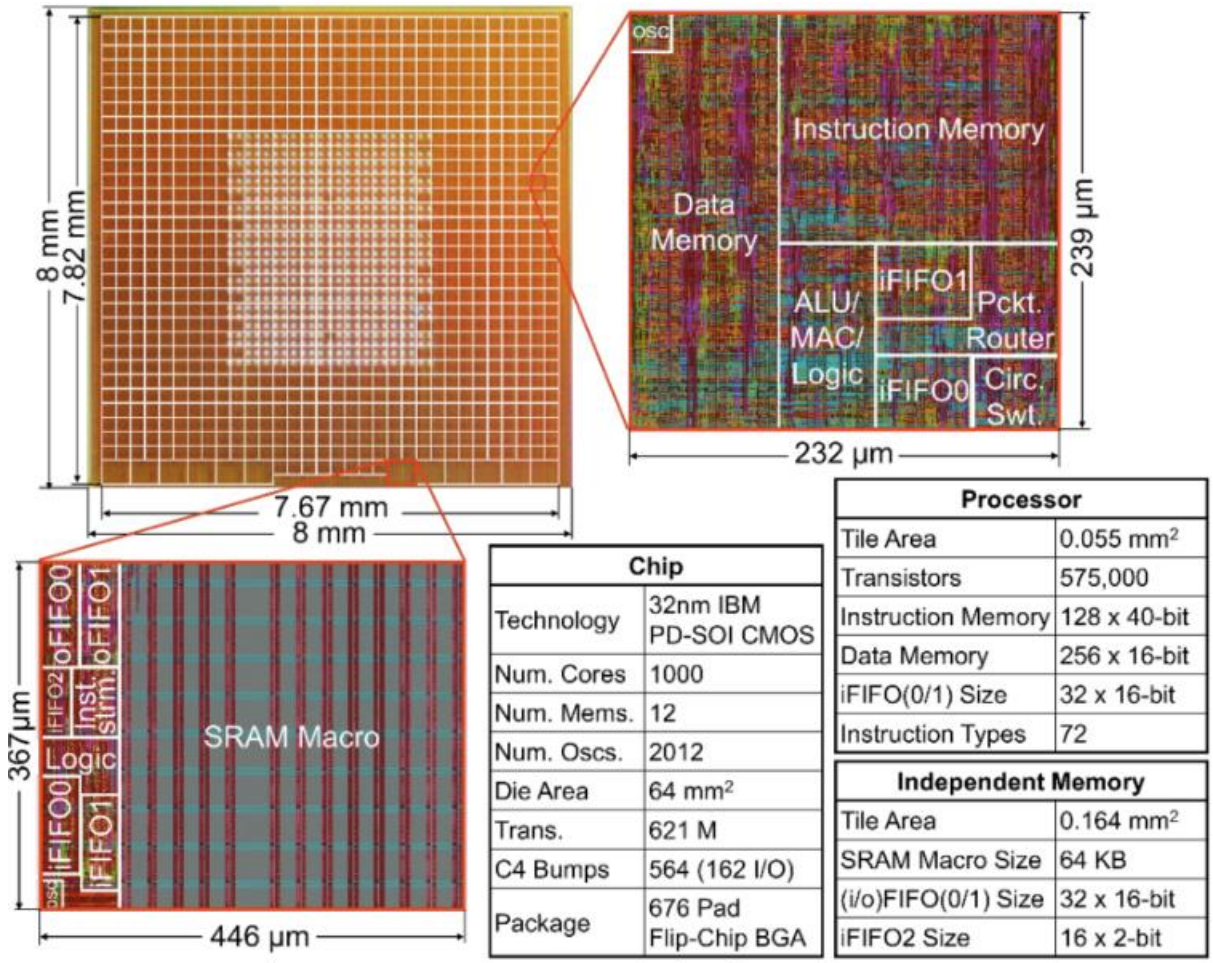
Huffman encoding shortens the length of the more familiar characters in a string while increasing the size of the least common characters [1]. The Huffman codes in JPEG are always less than or equal to 16 bits but can be as small as 2 bits. The Huffman tables in JPEG specify a value of all possible combinations of run-length encoded bytes, except those where the value size is greater than 10, as these values are impossible after DCT-II. There are 2 DC coefficient Huffman tables and two AC Coefficient Huffman tables. Of the two tables reserved for each of these cases, it is possible to assign a color channel to either one in the header, although it is typical to set one table to the Y channel and the other to the CbCr channels. Advanced JPEG encoders can compute an optimal Huffman table for each color channel for each JPEG; however, this inherently takes multiple passes over the data and thus is not recommended for throughput reasons. Instead, this work uses the provided standard Huffman tables in the JPEG specification.

2.11.2 Arithmetic Encoding

Arithmetic encoding is an advanced encoding technique supported by the JPEG standard [1]. However, not many JPEGs use arithmetic encoding as encoding techniques are patentable and thus legally barred in some cases. This work does not support arithmetic encoding to avoid legal complications and align itself with competing results.

Chapter 3

Background of the KiloCore Platform



Chip	
Technology	32nm IBM PD-SOI CMOS
Num. Cores	1000
Num. Mems.	12
Num. Oscs.	2012
Die Area	64 mm²
Trans.	621 M
C4 Bumps	564 (162 I/O)
Package	676 Pad Flip-Chip BGA

Processor	
Tile Area	0.055 mm²
Transistors	575,000
Instruction Memory	128 x 40-bit
Data Memory	256 x 16-bit
iFIFO(0/1) Size	32 x 16-bit
Instruction Types	72

Independent Memory	
Tile Area	0.164 mm²
SRAM Macro Size	64 KB
(i/o)FIFO(0/1) Size	32 x 16-bit
iFIFO2 Size	16 x 2-bit

Figure 3.1: Die photo of the KiloCore array and core specification information [5].

3.1 Overview

KiloCore represents the 3rd generation of manycore processor architectures from the UC Davis VCL laboratory [5]. It contains 697 energy-efficient cores that are all independently programable (MIMD). Each of the cores can communicate with its adjacent neighbors using dual-clock FIFOs, and when a core is not in use, it can power down independently. In addition,

there are 14 memory modules containing 64 KB of memory each; they can host data or instructions information. Finally, the chip comprises packet switch routers, a circuit switch network, and independent core clock oscillators.

3.2 Relevant Architectural Highlights

3.2.1 Core Information

Each KiloCore core allows for two inputs through 32 x 16-bit FIFOs [5]. Cores can clock to 1.78 GHz using 1.1 V regardless of what instructions are issued, and cores are the most energy efficient at 1.20 GHz using 0.9 V [5]. Additionally, the cores can reach 2.29 GHz, but only when avoiding “critical paths related to ALU carry and zero flags” [20]. Each core allows 128 40-bit instructions and 256 16-bit words [5]. It is only possible to get the total 256 words out of the data memory if 128 words of the data are entirely independent of the other 128 words of the data. For example, if you would like to add two elements to the data memory, they would need to come from two different 128 x 16-bit memory banks. The processor may replicate the same data in both memory banks to use fewer cycles during a read, sacrificing space in the data memory.

3.2.2 FIFO Information

Each core contains two 32 x 16-bit input FIFOs. However, FIFOs slow their operation when nearing 24 words to prevent overflows [5]. Only the first 24 writes to the input FIFOs are guaranteed to occur without any stalls. If a core needs to transfer a large amount of data to another core, sending 24 words to each input FIFO in alternating order is recommended. Consequently, the first input FIFO is more likely to be empty before it is written to again.

3.3 Programming on KiloCore

Programming on KiloCore is done using KiloCore assembly, C++, or (under limited circumstances) Python. The simulator for KiloCore will compile C++ to LLVM using Clang and then convert Clang's output to KiloCore assembly [5]. As with all compilers, it is essential to write C++ code in a manner that the assembly output code is optimized.

Specialized pragma codes such as "pragma unroll" are supported and can drastically affect performance. The branch predictor can also be influenced by explicitly labeling a conditional as unlikely or likely when programming. Branch flags always force the branch predictor to assume the branch is taken, increasing performance.

Finally, the KiloCore simulator allows the user to change speed and voltage data on the process to generate accurate throughput and power information. The simulator reports the time of the first output, the final output, the energy used, branch prediction accuracy, core count, and total core utilization. The simulator runs with minimal overhead; however, unoptimized applications can take a significant time to simulate.

Chapter 4

JPEG Implementation on the KiloCore Platform

4.1 Overview and Testing

4.1.1 Overview

This chapter showcases 29 working implementations of JPEG encoders on the KiloCore platform. Architectural details are outlined for each implementation, including core layout, code changes, and drawbacks. All implementations are for 4:4:4 baseline JPEG encoding using the sample quantization tables and scaling them based on a provided quality input. Sample JPEG tables are used during Huffman encoding. Although all the Huffman and quantization tables are replaceable, there are no hardcoding decisions based on the given qualities of these specific sample tables. The sample tables focus on algorithmic improvements rather than compression or quality considerations.

4.1.2 Testing

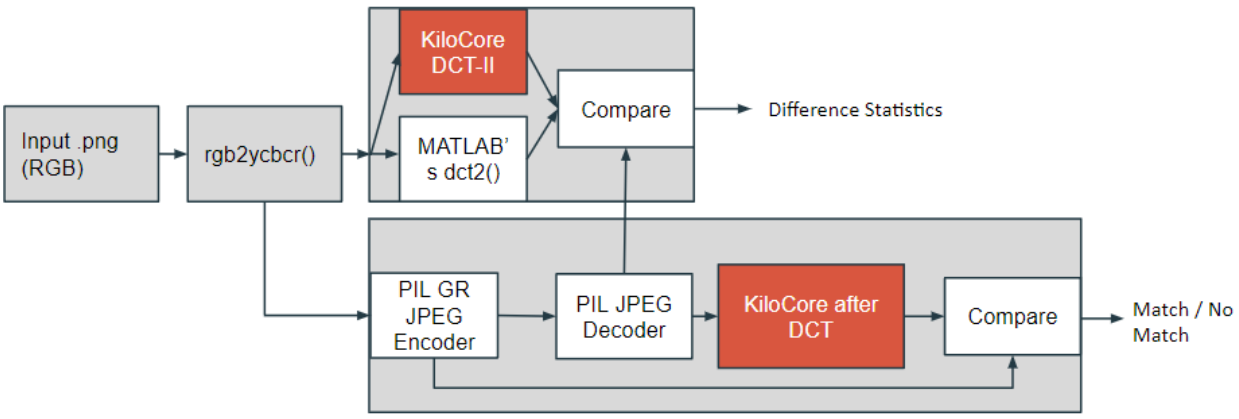


Figure 4.1: Testing block diagram for JPEG implementations, using the PIL Python JPEG library as a golden reference

Testing for JPEG encoders is challenging due to the loose precision requirement of the JPEG specification. Due to this, two accurate JPEG encoders can have different outputs while still being compliant. The DCT transform method and precision result in differing outputs between JPEG encoders. The PIL Python JPEG library, built on libjpeg, is used as a reference to validate KiloCore implementations.

First, a sample input file is converted to YCbCr and then passed to KiloCore's DCT-II cores, MATLAB's build in dct2() function, and PIL's JPEG Encoder [3,6]. PIL's JPEG Encoder's output is then decoded to the DCT coefficients and provided as input to a MATLAB script that compares it with the KiloCore implementation and the MATLAB built-in function implementation. A quantization table of all ones avoids additional errors in the decoding process; furthermore, the output from KiloCore's and MATLAB's functions is rounded as JPEG quantization rounds the numbers; thus, comparisons of the rounded outputs have a more significant impact on image fidelity. Finally, there is a comparison to see the magnitude and frequency of discrepancies between the reference MATLAB built-in DCT-II. Table 4.1 summarizes the results of the comparisons, showing that KiloCore's DCT-II computation is closer to the reference MATLAB implementation than PIL for the given input. PIL is compliant; however, PIL's method used to compute the DCT-II is prone to more rounding errors.

	KiloCore DCT-II Matrix Transform Algorithm	*KiloCore DCT-II AA&N Algorithm*	*PIL's DCT-II AA&N Algorithm*
Max Error	1	1	1
Min Error	-1	-1	-1
Average |Error|	0.0063	0.0177	0.0625

Table 4.1: DCT-II accuracy comparisons

This process repeats for both DCT-II algorithms implemented on KiloCore, the matrix multiplication method outlined in 2.6.2, and the AA&N method outlined in 2.6.3. Furthermore, the decoded PIL output was passed to the remaining KiloCore JPEG cores to confirm that the output from KiloCore perfectly matches the output from PIL. Finally, implementations that did not change DCT-II algorithms were compared with previous implementations to ensure both outputs matched perfectly. The same justifications of compliance stand for all versions. In fact, for most versions (sans version 1 and 2), the output should perfectly match the previous version's output.

KiloCore does not support floating point operations, so fixed point 16-bit operations are used for the DCT-II transform. However, fixed-point operations create errors between version 1 of the JPEG encoder and the MATLAB built-in function, even though they use the same matrix transformation method to calculate the DCT-II. Although the AA&N fixed point method is not as accurate as the fixed-point matrix transformation method, it is the method of choice for most implementations because it is significantly faster and the industry standard for JPEG encoders.

Finally, images used for testing include all exceptional cases (covered in Section 2.9) relevant to JPEG: ZRL (sixteen consecutive zeros), EOB (the last value of the 8x8 pixel matrix is zero), and no EOB (the last value of the 8x8 pixel matrix is nonzero), ensuring that any image, regardless of data, will successfully encode on every encoder version. Images particularly challenging due to color depth were also used, for example, an all-white image. RGB to YCbCr conversions that fail to remove headroom or footroom are unable to display colors like true white. Every JPEG version completes proper color conversion and can show the full-color range.

4.1.3 Relevant Abbreviations

The following abbreviations are used in future diagrams, and are clarified for the reader's reference:

- DCTII_p1: The first part of the calculation of the 2-dimensional discrete cosine transform (DCT-II) which calculates 1-dimensional discrete cosine transforms (DCT) of the eight horizontal rows of pixels within each 8x8 pixel block (Section 2.6).
    - DCTII_p1a: Same as DCTII_p1, but with only the top four rows.
    - DCTII_p1b: Same as DCTII_p1, but with only the bottom four rows.
- DCTII_p2: The second part of the calculation of the 2-dimensional discrete cosine transform (DCTII) which calculates 1-dimensional discrete cosine transforms (DCT) of the eight vertical rows of pixels within each 8x8 pixel block (Section 2.6).
    - DCTII_p2a: Same as DCTII_p2, but with only the left four rows.
    - DCTII_p2b: Same as DCTII_p2, but with only the right four rows.
- Quantize_Y/Quantize_CbCr: The element-wise multiply of a given 8x8 pixel block with respective quantization tables and proper rounding (Section 2.7).
- RLE: Run-length encoding a given 8x8 pixel block (Section 2.9).
- Huffman_Y/Huffman_CbCr: Huffman encoding (Section 2.11) and difference encoding a given 8x8 pixel block (Section 2.10).
- Organizer/Organizer_i/Organizer_h: Concatenates the variable length bit streams into one final output.
- Header: Outputs the JPEG header before the image data (Section 2.3).

- Zigzag: Performs the zigzag operation (Section 2.8) on a given 8x8 pixel block.
- Compress_Y/Compress_CbCr: Concatenates entropy codes into a single bit stream for a given 8x8 pixel block.
- Encode_Y/Encode_CbCr: Performs run-length encoding (Section 2.9), Huffman encoding (Section 2.11), and difference encoding (Section 2.10) on a given 8x8 pixel block.
- Quantize_zigzag_Y/Quantize_Zigzag_CbCr: Performs both quantization operations (see Quantize_Y/Quantize_CbCr) and zigzags the given 8x8 pixel block (see ZZ).
    - Quantize_Zigzag_Y_p1/Quantize_Zigzag_CbCr_p1: same as Quantize_Zigzag_Y/Quantize_Zigzag_CbCr, but only operates on the even matrix indexes of a given 8x8 pixel block.
    - Quantize_Zigzag_Y_p2/Quantize_Zigzag_CbCr_p2: same as Quantize_Zigzag_Y/Quantize_Zigzag_CbCr, but only operates on the odd matrix indexes of a given 8x8 pixel block.
- RGB-2-Y/RGB-2-Cb/RGB-2-Cr: Performs RGB to YCbCr conversion to the respective channel (Section 2.5).
- Sizer: computes the length in bits of run-length encoded codes.
- Color_Pass: Distributes 8x8 pixel blocks amongst the proper pipelines.
- Pass: Funnels bit streams from pipelines to the organizer core(s) in the proper order
- Chain: Serves as a buffer

- Byte_Stuff/Byte_Stuff_p1/Byte_Stuff_p2: Inserts 0x00 bytes when a 0xff byte is naturally encountered in the output stream to remain complaint with the JPEG standard.

4.2 JPEG Encoder Version 1

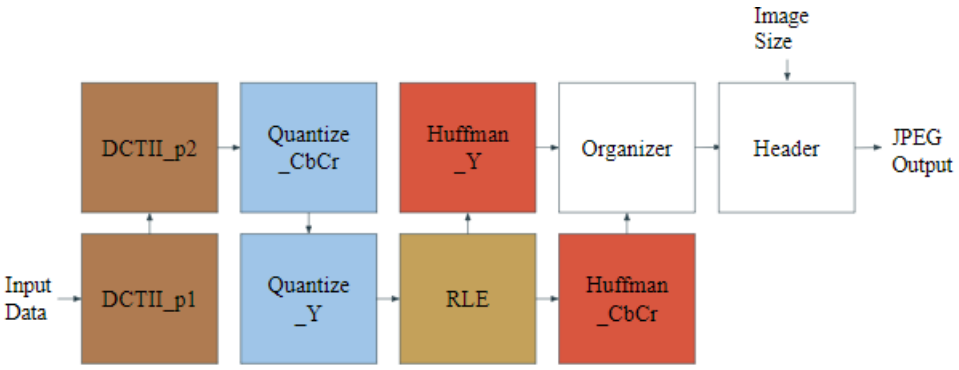


Figure 4.2: JPEG Encoder Design 1

The first implementation of the JPEG encoder focused on functionality over performance. It uses the least number of cores within reason. Quantize_CbCr and Quantize_Y are the quantization cores for their respective color channels. They pipeline the quantization process as one KiloCore core cannot fit both quantization tables and allow for rounding intermediates.

The DCT-II cores are divided into two parts as KiloCore cannot store an 8-by-8 matrix and the output of both matrix multiplications without running out of space. It can, however, store the input 8-by-8 matrix and then output the matrix after one matrix multiplication. Furthermore, multiple iterations of the DCT-II cores exist to extract the most precision from the operation. Initially, the signed 8-bit input multiplies a fixed point s0.15 value from the matrix multiplication and then truncates back down to s15.0 so that KiloCore could pass the data to the following core in one cycle. After realizing the possible output range for the DCT-II multiplication could yield only 9-bit numbers before the decimal point, the next generation truncated to s9.6 before entering

part two. These values were then multiplied by s0.14 values of the transformation matrix (to prevent overflow) and turned into s11.4 outputs for quantization. Finally, cores are updated to include rounding. Rounding is done by Equation 4.1, where "z" is the value to be rounded, "LSB" is the value of the least significant bit after the decimal point, and "truncate" removes all bits after the decimal [7]. The special case z = xx…x.100…0 means the value after the decimal point has a 1 in the most significant position and zeros everywhere else. This rounding method is unbiased.

$$\text{round}(z) = \begin{cases} \text{truncate}\left(z + \frac{1}{2}LSB - 1\right), & z < 0 \text{ and } z = \text{xx} \ldots \text{x.} 100 \ldots 0 \\ \text{truncate}\left(z + \frac{1}{2}LSB\right), & \text{otherwise} \end{cases} \tag{4.1}$$

Quantization occurs in the Quantize_CbCr and Quantize_Y blocks (Figure 4.2). Quantize_CbCr comes first to allow the Y channel to pass through it and compute the quantization concurrently with the Cb channel. Next, the two cores are separated to store the full 8-by-8 sample quantization table provided by the JPEG specification and intermediate values needed in rounding without overflowing the data memory. Quantization uses the same rounding method described in Equation 4.1.

The RLE block in Figure 4.2 is the run-length encoding core. It handles both run-length encoding and zigzagging. It uses an algorithm that computes the following value to read from the input 8-by-8 matrix based on the previous value read. It then generates the run-length codes, combines them with the value size to be encoded as described in the JPEG specification, and sends both the code and value to be Huffman encoded in the following core.

Huffman encoding happens in Huffman_Y and Huffman_CbCr (Figure 4.2). The Y and CbCr channels have their own core for Huffman encoding to fit their respective Huffman tables into data memory. Although the name mentions only AC, DC difference-encoding occurs in

these blocks. The codes are sent to the following core once the Huffman code is found from the look-up table.

Organizer (Figure 4.2) is the only core not explicitly discussed or highlighted in the JPEG specification. It combines each channel's variable length bit streams into one-bit stream. Huffman_Y and Huffman_CbCr cannot do this without knowing the output of each other's cores; therefore, Organizer handles this. The last input into the design is always 0x8000, which is an unachievable output for all cores, signaling them to pass that value to the next core without processing. This value then ends at Organizer to allow the last byte in the image to be stuffed.

Finally, the header core takes the size of the image as an input. It outputs the corresponding header of the image before passing all outputs from the organizer directly to the output. All outputs are one byte wide in this design.

4.3 JPEG Encoder Version 2

Version 2 uses JPEG Encoder Design 1 (Figure 4.2) but replaces the DCT-II matrix transformation algorithm with the AA&N DCT-II algorithm. Fixed point s1.14 format values represent the five constants in the AA&N algorithm, and the output of part one of the transformation is s10.2, and the output of part two of the transformation is s13.2. s10.2 is used instead of s10.5 to prevent 32-bit operations from happening in part two of the transform to balance the pipeline better. The accuracy numbers in part 4.1.2 reflect this choice.

Minimizing the number of cycles it takes to read in an input, the KiloCore architecture uses address generators. Likewise, writing values to the output of the cores also uses address generators [5].

4.4 JPEG Encoder Version 3

Version 3 uses JPEG Encoder Design 1 (Figure 4.2) and the previous changes from version 2. Version 3 is the first of many cores to push work upstream away from the organizer core to prevent the anticipated bottleneck. At this point in development, the goal was to parallelize the channels of the JPEG encoder; however, that would prove ineffective with how much work the organizer core currently does. It was suspected that the organizer core would create a significant bottleneck negating any performance benefits from parallelizing the channels.

The organizer core no longer calculates the size of the AC Huffman or DC Huffman codes but instead it passes that information in the following input from either Huffman_Y or Huffman_CbCr. As a result, the naïve size algorithm is *O(n)*, creating a significant delay in the organizer core. By moving it upstream, the eventual parallelization of the channels helps minimize this delay.

4.5 JPEG Encoder Version 4

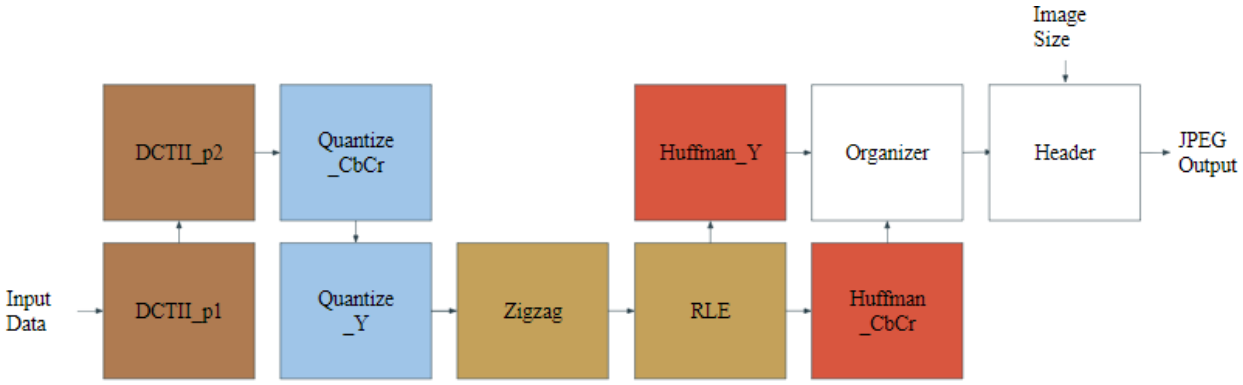


Figure 4.3: JPEG Encoder Design 2

Version 4 is the first core to use JPEG Encoder Design 2 (Figure 4.3). Version 4 includes all the updates of version 3 while also pulling the zigzag task out of the RLE core and placing it in its own separate ZZ core. In addition, the zigzag algorithm now reorganizes the input data on-

the-fly, storing a minimal amount before writing it back to the input. By skipping index calculations on the zigzag process and not keeping any unnecessary values in the zigzag core, the overall throughput is expected to increase.

4.6 JPEG Encoder Version 5

Version 5 uses JPEG Encoder Design 2 (Figure 4.3) and builds upon improvements present in Version 4. Although simple, Version 4 updates the rounding algorithm in the DCT-II and quantization cores by removing one comparison per round. Previously the two conditions in Equation 4.1 were checked separately, but now one logical and comparison and equivalency check handles both conditions simultaneously.

4.7 JPEG Encoder Version 6

Version 6 uses JPEG Encoder Design 2 (Figure 4.3). In addition to Version 5's rounding fixes, Version 6 increased the word width of the organizer core's output to 16 bits. Therefore, minimizing the number of times the organizer core needs to write to its output and helping relieve the future anticipated bottleneck.

4.8 JPEG Encoder Version 7

Figure 4.4: JPEG Encoder Design 3

Version 7 is the first core to parallelize the color channels of the input using JPEG Encoder Design 3 (Figure 4.4). As a result, there is no dependency between color channels besides the need to concatenate the bit streams in the correct order. This fact allows the three independent color channels to be parallelized.

Pass_0 and Pass_1 are added to the design to aid with passing one block at a time to each of the channel's pipelines. The top track is reserved for Y, the middle channel for Cb, and the bottom for Cr.

The quantization cores have been updated to no longer anticipate a two-stage quantization process. Instead of six total cores from scaling up, there are only three for this reason. Similarly, the RLE core is updated to output everything to the same output channel. A single Huffman_CbCr core combines the CbCr channels before the organizer core as they are likely easier to merge due to the chrominance quantization table's values being larger on average than the luminance quantization table's values, saving a passer core that would be needed if there was a second Huffman_CbCr core as every core can have only two inputs.

4.9 JPEG Encoder Version 8

Figure 4.5: JPEG Encoder Design 4

Version 8 was the first version to use JPEG Encoder Design 4 (Figure 4.5), the same as Design three but with RGB conversion built in. The RGB conversion cores each read three values of RGB, convert them into an s7.0 rounded color value of their respective channel, and pass it to the following core. Each of these cores uses fixed-point arithmetic in the conversion but still follows Equation 2.1, Equation 2.2, and Equation 2.3.

The RGB core that handles the Y channel conversion passes one block of output (3 x 64 words) downward to the Cb channel before processing that same block. The Cb channel does the same for the Cr channel, and the Cr channel processes only the block. Each RGB core needs to process 192 words to create an 8-by-8 block for its output.

4.10 JPEG Encoder Version 9

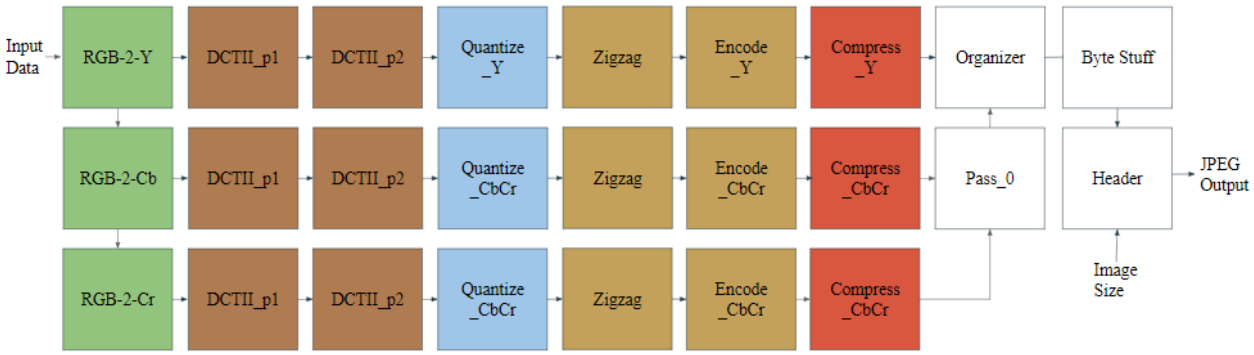


Figure 4.6: JPEG Encoder Design 5

Version 9 uses JPEG Encoder Design 5 (Figure 4.6), which replaces RLE with Entropy_Y and Entropy_CbCr. In addition, Compress replaces Huffman_Y and Huffman_CbCr. The entropy cores handle both run-length and Huffman encoding. These cores are designated by color channel to allow the entire Huffman encoding table of a respective channel to fit inside the data memory (similar to why Huffman_Y and Huffman_CbCr were separated). The entropy encoding cores dump Huffman codes, Huffman code sizes, values, and value sizes to the compression cores.

The compression cores compress the output of the entropy cores into variable-length bit streams—the bit streams output 16 bits at a time to the organizer core. When a new block arrives at the compressor cores, the compressor cores dump the rest of the bit stream along with the number of valid bits. Consequently, the organizer core knows when the next block is coming. At times it outputs an empty bit stream as its last output and a size of zero; this is required to avoid the case where the last bit stream is 16-bits long and therefore indistinguishable as the bitstream ending. The compressor cores are separated by channel to handle their individual channels to support exceptional Huffman cases (EOB and ZRL are covered in Section 2.9).

Finally, the organizer core covers 2 cases: the incoming bitstream has no offset and needs to be routed to the output, and the incoming bitstream has and offset and needs to combine the current word with the previous 16-bit word before rounding to the output. Future bottlenecks are avoided as much as possible; each case uses the fewest operations possible.

Case one, the no offset case:

1. Reads the input value.
2. Reads the input values size.
3. Checks if the size is still 16 (not the EOB).
4. Writes the value to the output.

Case two, the offset case:

1. Reads the input value.
2. Reads the input values size.
3. Checks if the size is still 16 (not the EOB).
4. Shifts the previous value over until only its unwritten valid bits are available.
5. Shifts the current value over to fill the rest of the previously shifted value.

6. Performs a bitwise OR to combine the values.
7. Writes the output value.
8. Saves the new output value as the old.

4.11 JPEG Encoder Version 10

Version 10 uses the same design as Version 9 (Figure 4.6), except the task of deciding the size of the Huffman AC code is passed to the compress cores instead of the entropy cores to help balance the pipelines to handle high amounts of data better.

4.12 JPEG Encoder Version 11

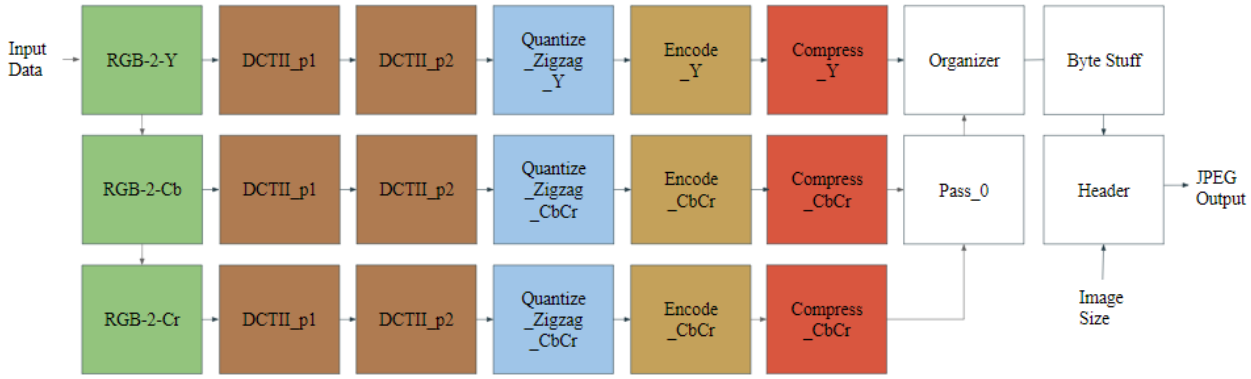


Figure 4.7: JPEG Encoder Design 6

Version 11 uses JPEG Encoder Design 6 (Figure 4.7). It scraps the changes of Version 10 and elects to combine zigzag and Quantization instead. Theoretically, this helps reduce the number of cores without affecting bandwidth, as zigzagging and quantization are relatively small pipeline operations.

4.13 JPEG Encoder Version 12

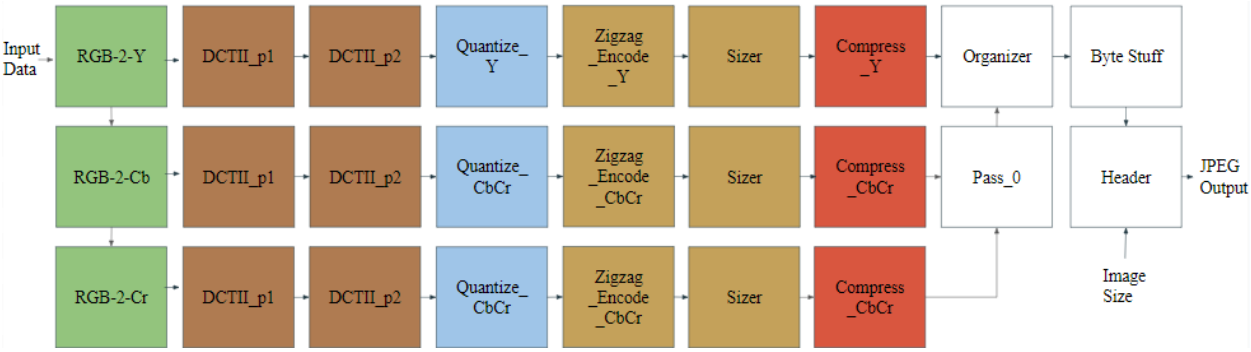


Figure 4.8: JPEG Encoder Design 7

Version 12 uses JPEG Encoder Design 7. It scraps the changes from Version 10 and 11 and combines zigzagging and entropy encoding into the same core. One of the most expensive operations in entropy encoding is getting the value size and code; that operation is now moved to its independent core to help balance the pipelines.

The new sizing algorithm has a time complexity of *O(log(n))* but had to do additional checks for edge cases, so the total amount of operations was closer to 18--dramatically down from the 32 operations that were possible in the worst case using the previously implemented *O(n)* algorithm.

4.14 JPEG Encoder Version 13

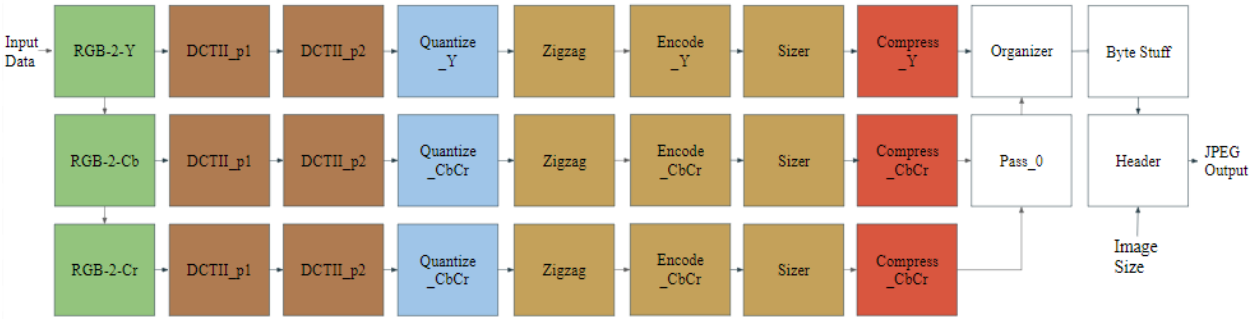


Figure 4.9: JPEG Encoder Design 8

Version 13 uses JPEG Encoder Design 8 (Figure 4.9). The main difference of this design is that it gives the zigzag operation its own core. Furthermore, Version 13 builds upon Version 12 by adding a more efficient size calculation algorithm to the sizer core. Luckily, there is a built-in leading zero-counting instruction in KiloCore's ISA. The LSDU instruction takes an unsigned input and returns how many leading zeros the number contains; however--due to a hardware oversight—it returns 0 when the input is 0 (instead of 16). Consequently, it takes one check, one LSDU instruction, and one subtraction to determine the size of a word. With this change, the size check algorithm is down from *O(log(n))* to *O(1)*.

Additionally, optimizations are made to the compressor core to help alleviate any pipeline imbalance from a particularly noisy image. The compiler can optimize the compressor core better since the central helper subroutine is now in-line.

4.15 JPEG Encoder Version 14

Version 14 returns to JPEG Encoder Design Version 5 (Figure 4.6), favoring the core savings over further balancing and the pipelines.

4.16 JPEG Encoder Version 15

Version 15 follows Version 14's lead by combining quantize and zigzag to form JPEG Encoder Design 6 (Figure 4.7). Preliminary performance numbers detailing the pipelines' most prolonged delay come from DCT-II, motivating the additional core savings to combine quantize and zigzag yet again.

4.17 JPEG Encoder Version 16

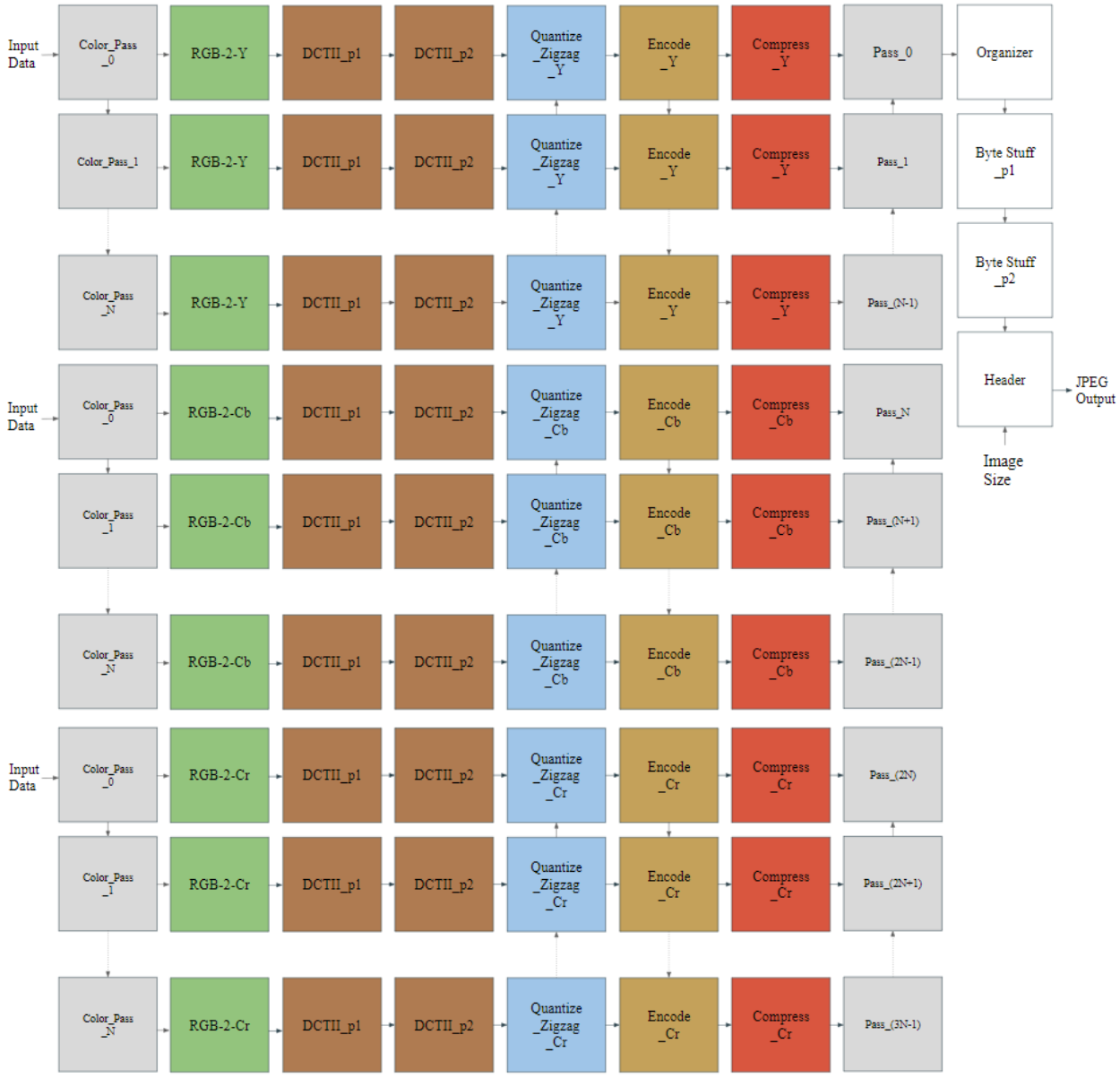


Figure 4.10: JPEG Encoder Design 6, N pipelines per channel

Version 16 uses JPEG Encoder Design 6, scaled to 2 pipelines per channel (Figure 4.10). The design has three inputs, each carrying the same data, to increase the bandwidth of the input port. This design marks the first to scale the number of pipelines per channel, and with that came algorithmic changes to the QZ and Entropy cores.

Since DC difference encoding depends on the previous block's value, the Entropy core now passes its DC coefficient downward to the following pipeline. This process recurses until the last pipeline passes the DC coefficient back to the QZ core and the first pipeline for the channel. The DC coefficient cannot pass back up through the Entropy cores as they already have two inputs.

A Python algorithm automatically generates the Color_Pass cores (Figure 4.10). The cores move data from the input and evenly distribute a JPEG processing block's worth of data (3 x 64 words, RGB input). Similarly, the Pass cores (Figure 4.10) collect the processed blocks and route them to the organizer core to ensure that bitstreams concatenate serially.

4.18 JPEG Encoder Version 17

Version 17 uses JPEG Encoder Design 6, scaled to 4 pipelines per channel (Figure 4.10). This design reflects no other notable changes.

4.19 JPEG Encoder Version 18

Version 18 uses JPEG Encoder Design 6, scaled to 8 pipelines per channel (Figure 4.10). This design reflects no other notable changes.

4.20 JPEG Encoder Version 19

Version 19 uses JPEG Encoder Design 6, scaled to 16 pipelines per channel (Figure 4.10). This design reflects no other notable changes.

4.21 JPEG Encoder Version 20

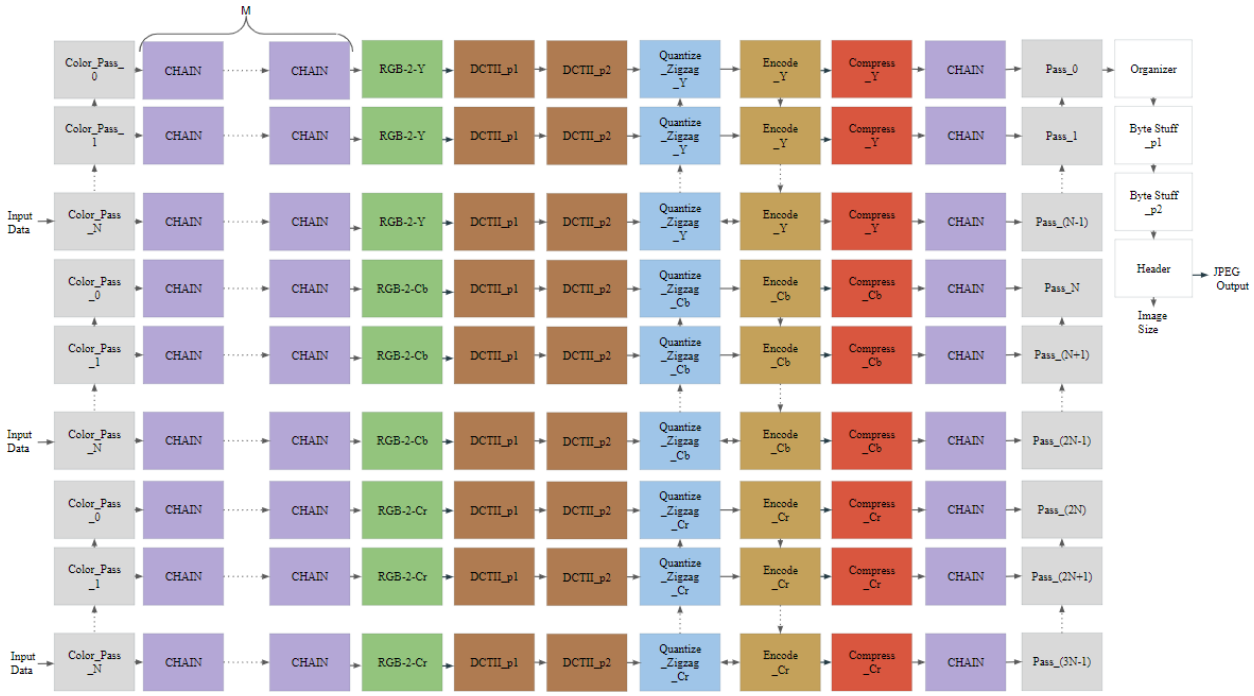


Figure 4.11: JPEG Encoder Design 6, N pipelines per channel, M input buffers

Version 20 uses JPEG Encoder Design 6, 8 pipelines per channel, with four input buffers (Figure 4.11). KiloCore FIFOs hold 32 words before forcing the output of the providing core to stall. Stalling FIFOs can become an issue in the Color_Pass cores as they need to distribute 192 inputs to the RGB converter cores. The RGB converter cores cannot process all 192 inputs fast enough not to stall and prevent the other pipelines from working. Since there are 192 inputs, 6 FIFOs are needed to completely store an entire block for processing without needing to stall. For this reason, Version 20 adds four input buffers in front of the RGB converter cores. Combined with the input FIFOs of the Color_Pass and RGB convert cores, this totals the needed 6 FIFOs.

Similarly, Version 20 adds an output buffer to the end of each pipeline to ensure that a pipeline would not stall due to the organizer not being able to process data fast enough. The

average 8-by-8 block does not produce more than 64 words to its output. Therefore, the input buffer of the Pass cores and the added output buffer adequately address the bandwidth limitation.

4.22 JPEG Encoder Version 21

Version 21 uses JPEG Encoder Design 6, with 16 pipelines per channel and four input buffers (Figure 4.11). This design reflects no other notable changes.

4.22 JPEG Encoder Version 22

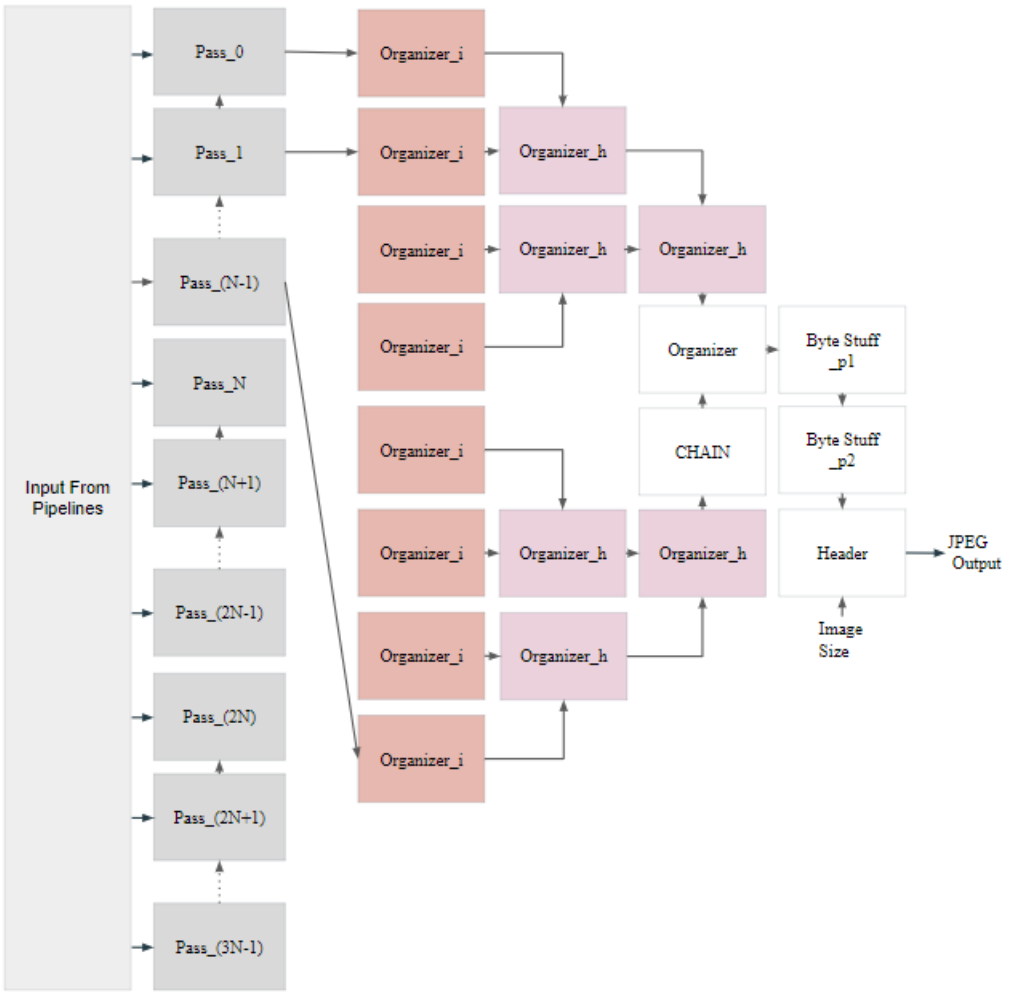


Figure 4.12: Pre-Organizer block diagram

Version 22 uses the same design as Version 20, with a new Pre-Organizer stage appended at the end (Figure 4.12). The algorithm that handles switching between two bitstreams takes

more operations than handling words within the same stream. The new pre-organizer stage allows 24 bitstream combinations to have the same operations as two. Each Organizer_i core combines three bitstreams of the same block (for each channel), and the Organizer_h blocks further consolidate these bitstreams into one.

Pass cores now target one block at a time, eliminating the need for buffer cores in the pre-organizer stage.

4.24 JPEG Encoder Version 23

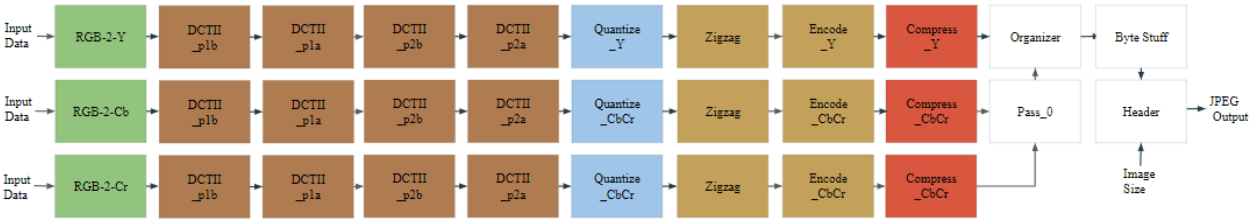


Figure 4.13: JPEG Encoder Design 9

Version 23 revisits the basic JPEG pipeline design for rebalancing (Figure 4.13). The DCT-II now uses four pipeline stages to complete the operations to minimize the bottleneck on lower-quality images. DCTII_p1 handles column-wise DCT-II operations, whereas DCTII_p2 handles row-wise DCT-II operations. DCT_p1a handles the first four columns, and DCT_p1b handles the last four. Similarly, DCT_p2a handles the first four rows, and DCT_p2b handles the last four rows. The cores that handle the first half of each block are farther in the pipeline to allow simultaneous processing with the last half of each block.

4.25 JPEG Encoder Version 24

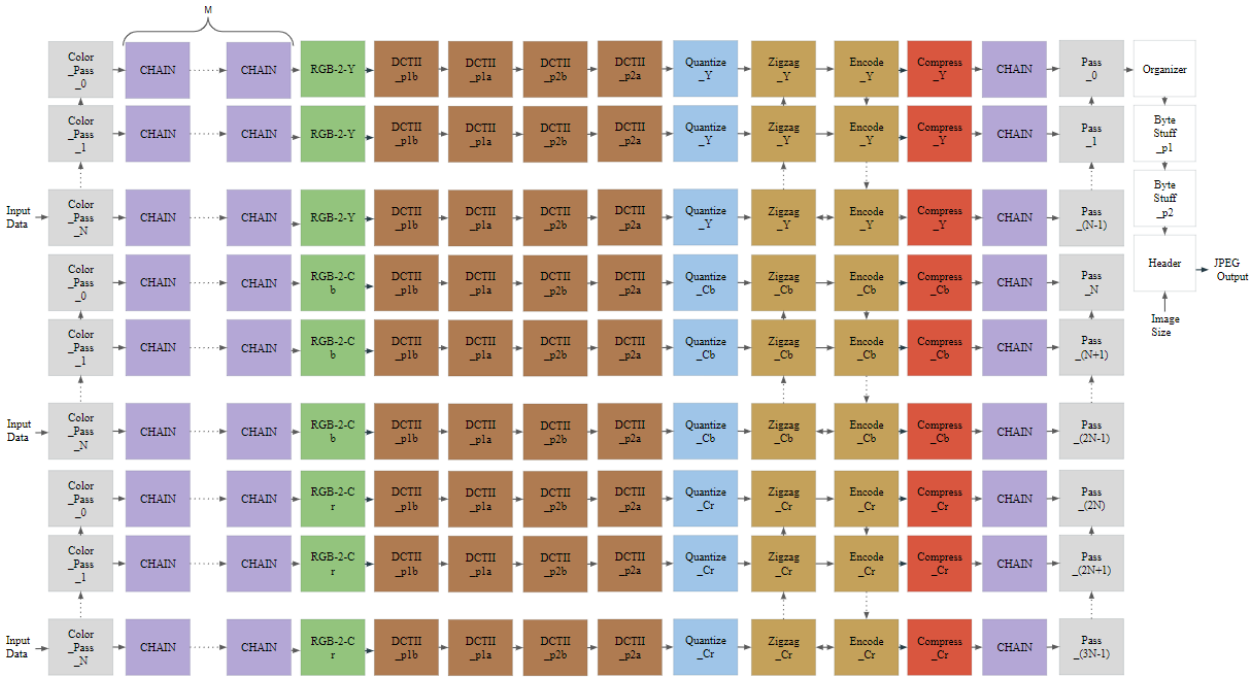


Figure 4.14: JPEG Encoder Design 9, N pipelines per channel, M input buffers

Version 24 uses JPEG Encoder Design 9 but scales up to 8 pipelines per channel with one input buffer (Figure 4.14). Although FIFOs can carry up to 32 words in KiloCore, they start to slow down after 24 to prevent overfilling. Color_Pass cores have been updated to write to their sequential core across both input buffers 24 words at a time to avoid this. When the core returns to the original input buffer to write, the data will be sufficiently removed to write another 24 words. Not only does this prevent stalling—the design never violates the reserve space in the FIFO—but we also effectively get 48 words per input buffer core.

RGB conversion is also updated to serve across both its input FIFOs. Consequently, in addition to performing the conversion operation, the core also serves as an additional buffer before the DCT cores. These three cores now allow for 144 words of buffer space. This design relies on the RGB converter being fast enough to process 48 words without stalling the previous

input buffers (as 192 words need to be processed to unblock the processing for the next pipeline in the chain.

4.26 JPEG Encoder Version 25

Version 25 uses the same JPEG Encoder Design 9, scaled to 8 pipelines per channel, with one input buffer. In addition, it innovates Version 24 by improving the DCT algorithm to use its two input FIFOs to help ensure an entire block can be stored in the FIFOs, avoiding any stalls.

4.27 JPEG Encoder Version 26

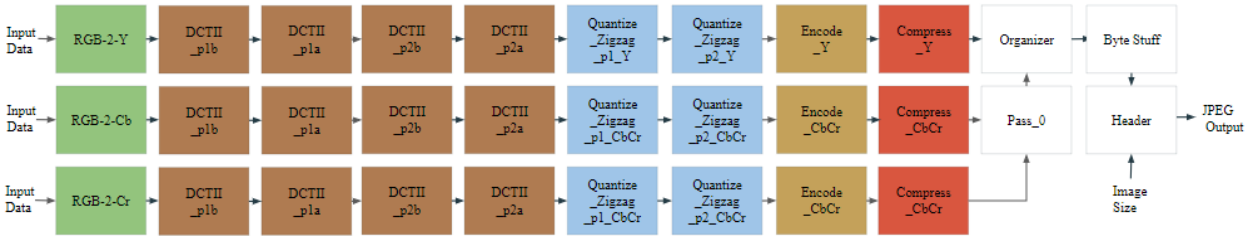


Figure 4.15: JPEG Encoder Design 10

Version 26 uses JPEG Encoder Design 10 (Figure 4.15). Almost every core in this design uses a new on-the-fly philosophy. Storing any data in a core takes additional unneeded processing, so every algorithm instead (where possible) processes all data without storing anything. Notable exceptions are the zigzag process where storing needs to take place.

All 4 DCT cores previously held 32 words before starting computation. During the column-wise DCT-II, this was particularly taxing due to the amount of uncoalesced reads and writes required in column-wise operations. Programming all data to be processed as it comes in leads to better compiler optimizations as it removes the loops used to store and save data. The DCT-II cores unroll all loops.

After initial performance numbers boomed from removing all loops in DCT-II cores, the quantization and zigzag cores unroll all loops with #pragma unroll. However, the quantize core

was still delayed, so it now shares its work with the previously named zigzag core. Now Quantize_Zigzag_p1 and Quantize_Zigzag_p2 accomplish a pipelined quantization and zigzagging of the block.

Entropy and Compression were both updated to remove all unneeded operations further. Consequently, the Entropy core partially unrolls the run-length encoding loop present in the Entropy core. A corner case exists where the Entropy core needs to check if it has counted more than 16 zeros in a row, as that would require inserting a 0xf0 byte. Still, that check only comes after passing through the 16 unrolled iterations and being sent to a nearly identical loop containing a said check. Consequently, blocks that do not have a run of 16 zeros can accelerate their performance by skipping 64 operations and blocks that can save at least 16 operations (and likely more).

Compression was initially written to append the entropy codes on the end of a 32-bit word until there were 16 valid bits to write to the output. Then, the 32-bit word would be shifted to find what bits to write and cleared of those bits. 32-bit operations are particularly taxing on KiloCore, a 16-bit data path architecture. Therefore, something as simple as a 32-bit shift can take multiple operations. Instead of using a 32-bit word to track the currently unwritten bits, a 16-bit value is now used. In an overflow, the bottom 16 bits are written to the output while the extra bits replace the “unwritten bits” value. Functions are also made inline to further increase compiler optimizations.

This pipeline is balanced to handle better the typical quality setting of the JPEG encoder, rather than the previous balance that served unrealistically high qualities. Finally, nearly all cores (besides the RGB converter core, as it has to light of a workload) have a similar delay with this design.

4.28 JPEG Encoder Version 27

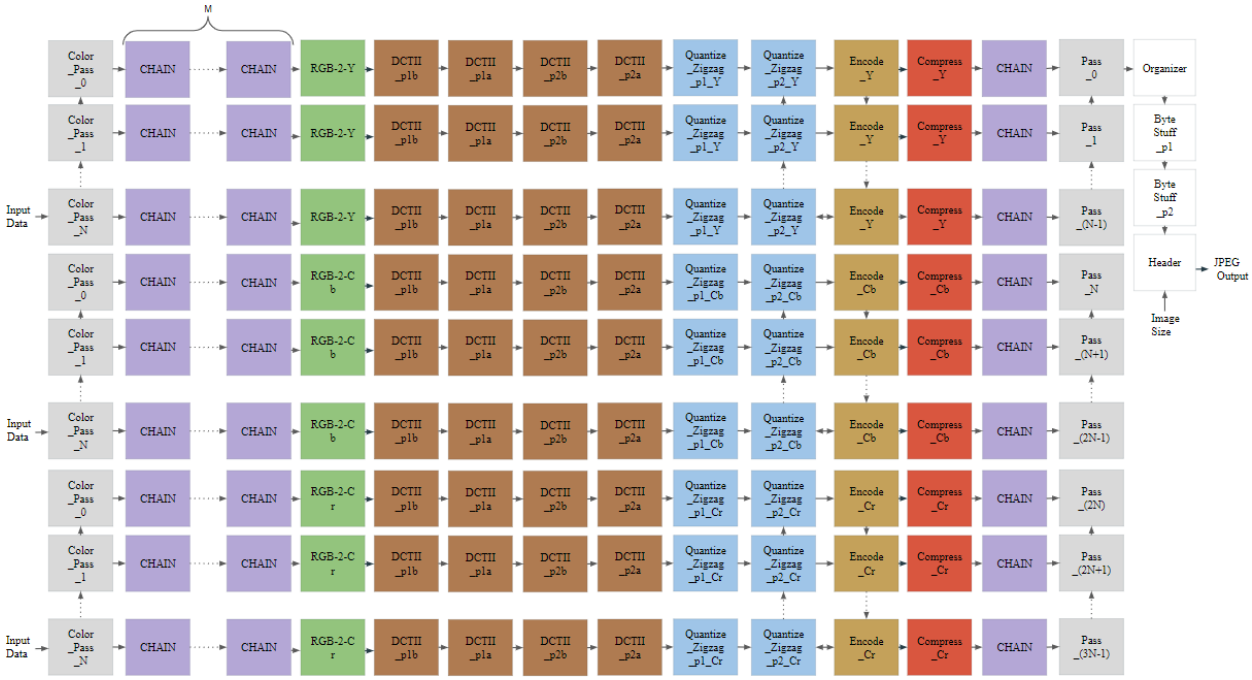


Figure 4.16: JPEG Encoder Design 10, N pipelines per channel, M input buffers

Version 27 scales Version 26 to 8 pipelines per channel, with two input buffers (Figure 4.16). Additionally, Version 27 adds input buffers to ensure an entire block can be stored without depending on the RGB converter's delay. With Color_Pass, two input buffers, and the RGB converter, there are now eighter 24-word FIFOs being used to store the entire 196-word block.

4.29 JPEG Encoder Version 28

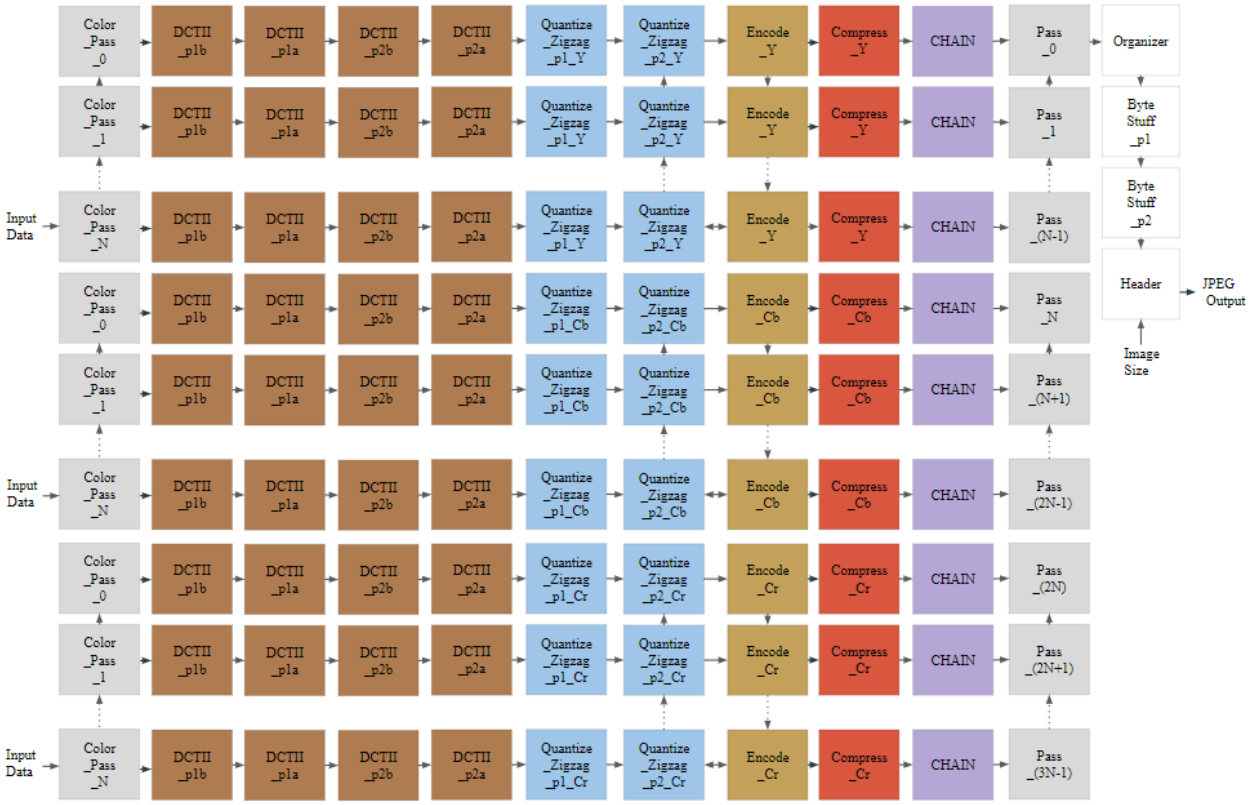


Figure 4.17: JPEG Encoder Design 10, N pipelines per channel, without RGB conversion

Version 28 removes RGB conversion and the necessary input buffers from Version 27 (Figure 4.17). Version 28 gives an alternative design to cases where input data is already in YCbCr format. Furthermore, it was impossible to simulate the complete performance characteristics of Version 27 due to a lack of input bandwidth (to be discussed in Chapter 5).

4.30 JPEG Encoder Version 29

Version 29 scales Version 28 to 16 pipelines per channel (Figure 4.17). This design reflects no other notable changes.

Chapter 5

Simulation Results of JPEG Implementations

5.1 Overview

Chapter 5 compares throughput, throughput per area, power, energy per megapixel encoded, and energy-delay product among the 29 implementations introduced in Chapter 4. These metrics show the benefits and disadvantages of different implementations.

All metrics vary between different quality factors and images due to the nature of JPEG encoding. For this reason, five different quality levels were tested to profile every version. Quality factor of zero is a worst-case scenario where the quantization table is wholly composed of ones. Furthermore, a quality factor of zero is not a realistic scenario as other encoding standards handle higher quality encoding better, whereas JPEG encoding trades image fidelity for compression. The higher the compression, the less load there is on the entropy, compression, and organizer cores, allowing higher throughput. Quality factor 0.1 gives a reasonable high-fidelity image benchmark of the design. While still demonstrating the performance penalty that comes with higher levels of quality, quality factor 0.1 better showcases JPEG encoders' strengths. Quality factors 0.1667, 0.5, and 1 represent a range of qualities to demonstrate how metrics change over different quality factors and how the relationship is not linear. Quality factor changes are not linearly proportional to compression changes; therefore, quality factor changes are not linearly proportional to the profiled metrics. Most designs industry designs are benchmarked at quality factor 0.1, and JPEG encoders are most used between quality factor 0.1 and 0.5.

Three test images were chosen (all 1240 x 960 pixels), and the metrics are averages between the three test images. libjpeg-turbo's testing also includes these test images and a quality factor of 0.1. These images (Figures 5.1 to 5.3) are more accessible to outside design comparisons and have below-average encoding times [8]. While the encoding times are below average, they are not corner cases; their results are relatively close to average picture results. Generally, images with higher frequency 8-by-8 pixel blocks have longer encoding times.

Throughput was calculated using Equation 5.1:

$$Throughput = \frac{(\#\ of\ Pixels) * 2^{20}}{Delay} \tag{5.1}$$

Average throughput was computed using Equation 5.2, where N is 3 for the three images:

$$Average\ Throughput = \frac{1}{N}\sum_{n=0}^{N} Throughput_n \tag{5.2}$$

Average power was calculated using Equation 5.3, where N is 3 for the three images:

$$Average\ Power = \frac{1}{N}\sum_{n=0}^{N} Power_n \tag{5.3}$$

Energy per megapixel encoded was calculated using Equation 5.4:

$$Energy\ per\ Megapixel\ Encoded = \frac{Average\ Power}{Average\ Throughput} \tag{5.4}$$

Throughput per area was calculated using Equation 5.5, where 0.05545 is the number of square millimeters per core on the KiloCore chip fabricated at 32nm [5]:

$$Throughput\ per\ Area = \frac{Average\ Throughput}{\#\ of\ Cores * 0.05545\ \frac{\mathrm{mm}^2}{\mathrm{core}}} \tag{5.5}$$

Energy-delay product was calculated using Equation 5.6:

$$Energy\text{-}delay\ Product = \frac{Energy\ per\ Megapixel}{Average\ Throughput} \tag{5.6}$$

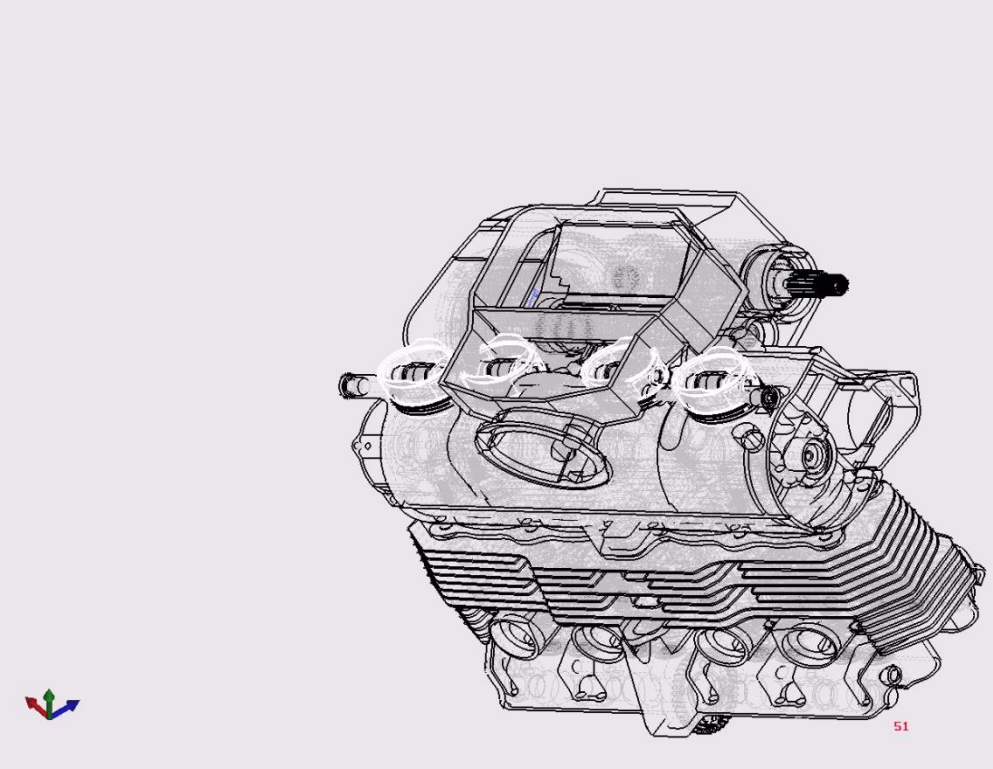


Figure 5.1: “vgl_6548_0026.ppm” encoded using quality factor 0 [8]

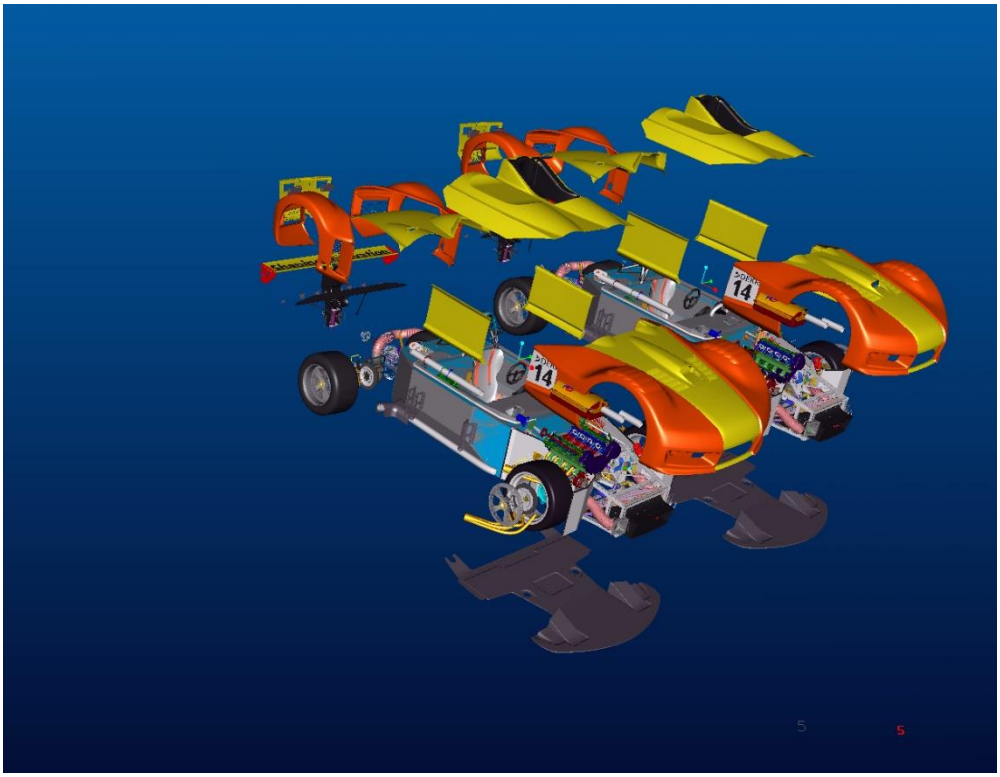


Figure 5.2: “vgl_6434_0018.jpeg” encoded using quality factor 0 [8]

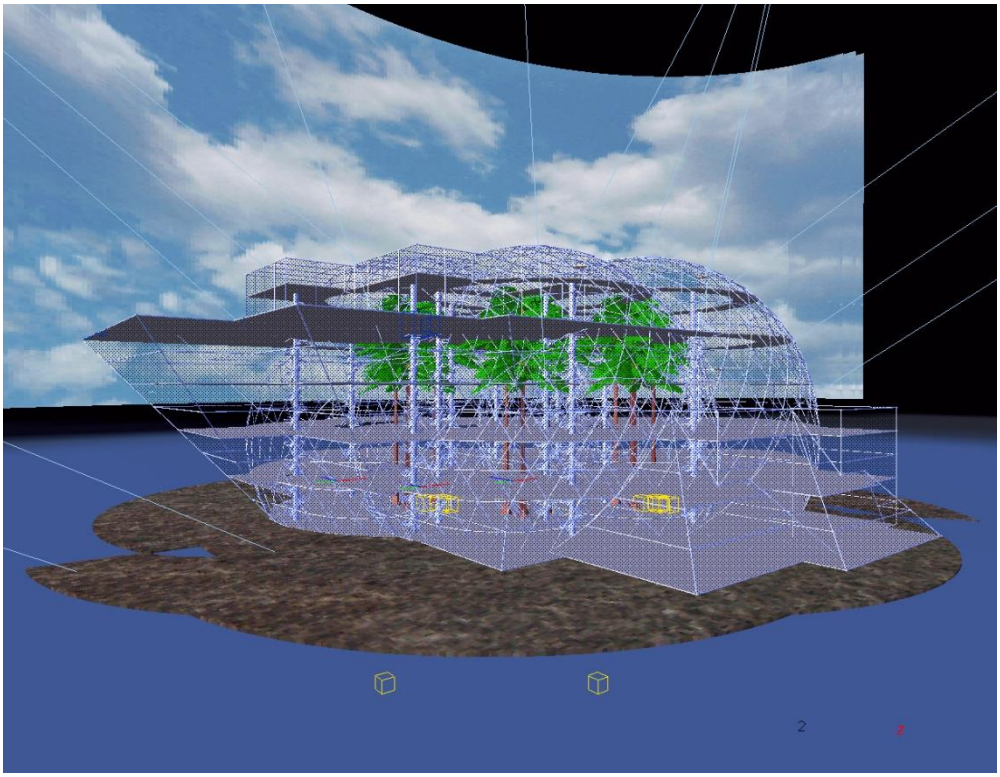


Figure 5.3: “vgl_5674_0098.jpeg” encoded using quality factor 0 [8]

5.2 Throughput Analysis

The following section discusses how throughput changes throughout the implementations and quality levels. As described in section 5.1, three images determine the average throughput of the design. However, these images have above-average complexity to encode; therefore, other more straightforward images yield higher than average throughput. Throughput values were taken directly from the KiloCore simulator's output.

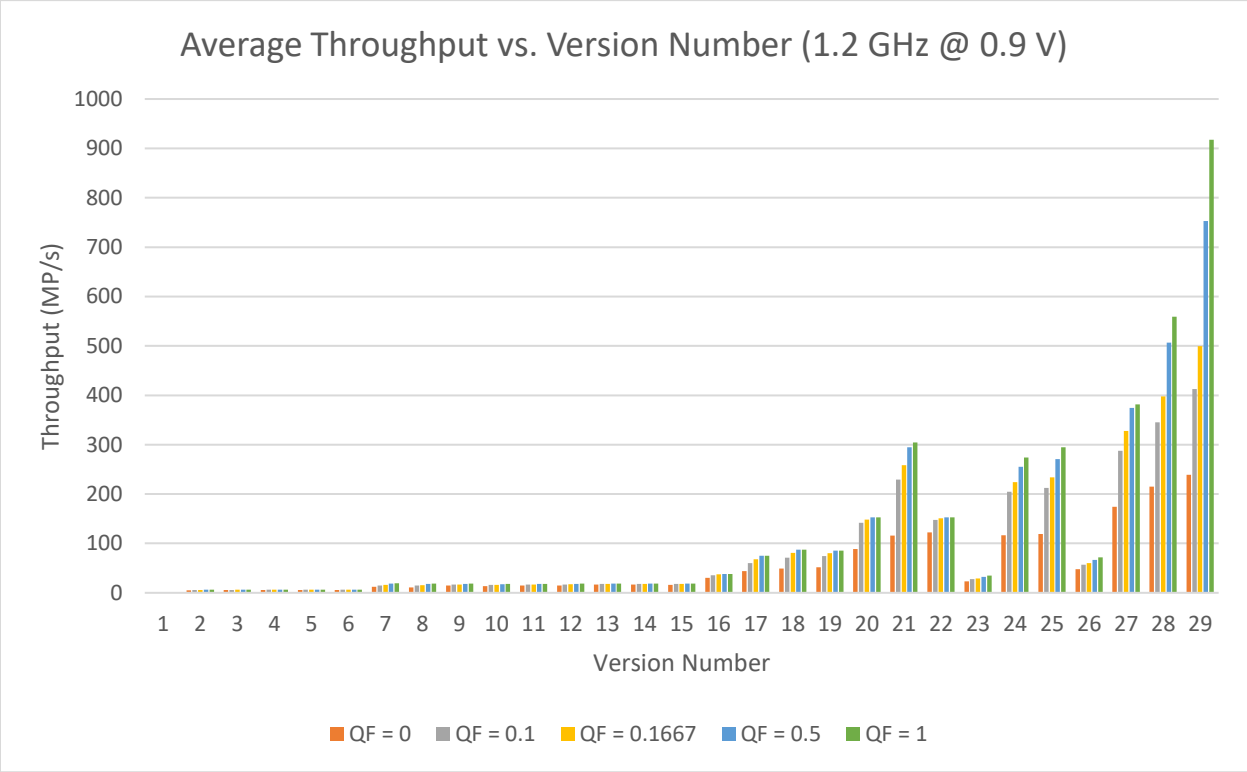


Figure 5.4: Throughput versus version number (1.2 GHz @ 0.9V)

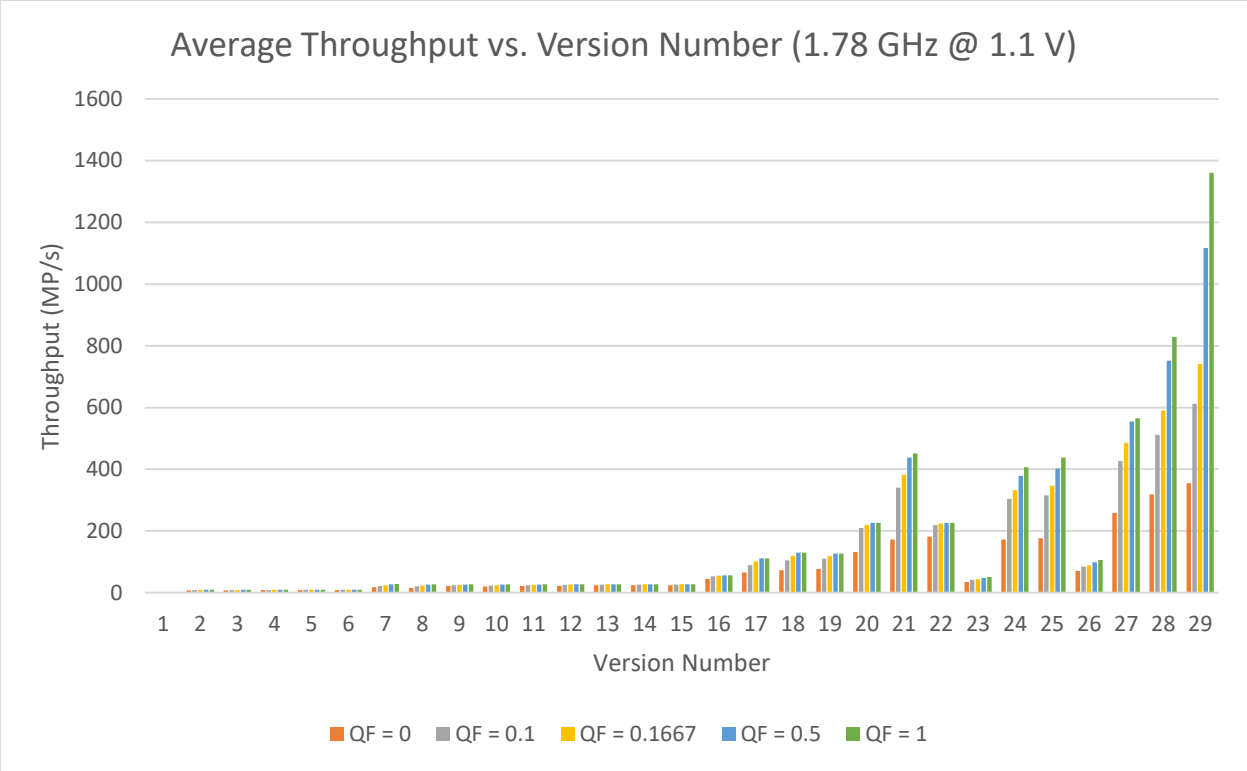


Figure 5.5: Throughput versus version number (1.78 GHz @ 1.1V)

Figure 5.4 and Figure 5.5 graph the throughput of the various implementations with different quality factors. As expected, the general trend shows upward throughput JPEG implementations innovate the encoding process. A notable exception is Version 26, as it was a single pipeline implementation that laid the groundwork for Versions 27, 28, and 29. Version 27 is faster than Version 28 due to the increased input bandwidth, not the exclusion of the RGB2YCbCr core. The workload of the RGB2YCbCr core is not the bottleneck of Version 27.

Increasing the clock speed has a proportional increase in throughput, showing the design scales with clock speed. This trend should hold indefinitely, so long as the clock speed increases are level across all cores. If not, increases to the bottleneck cores (DCT for low qualities, compress/entropy/organizer for higher qualities) result in the most significant speedups.

Higher quality factor values result in higher levels of throughput, and vice versa. This is due to the greater amount of non-zero data to encode, and more work for the entropy,

compression, and organizer cores when encoding with lower quality factor values. The best-case result for throughput was 286 MP/s for the quality factor set to 0, while 1.332 GP/s for the quality factor set to 1. Quality factor zero is an extreme case, so it is not representative of the average encoder experience, whereas quality factor 0.1 is a far more realistic test case. By slightly increasing the quality factor, throughput is increased to 549 MP/s, showing a non-linear relationship between the quality factor and throughput.

Version 1 sees no difference between quality factors as the bottleneck is the DCT-II operation. DCT-II's workload does not depend on quality factor.

5.3 Power Analysis

The following section discusses how power changes throughout the implementations and quality levels. As described in section 5.1, three images determine the average power of the design. Power values were taken directly from the KiloCore simulator's output.

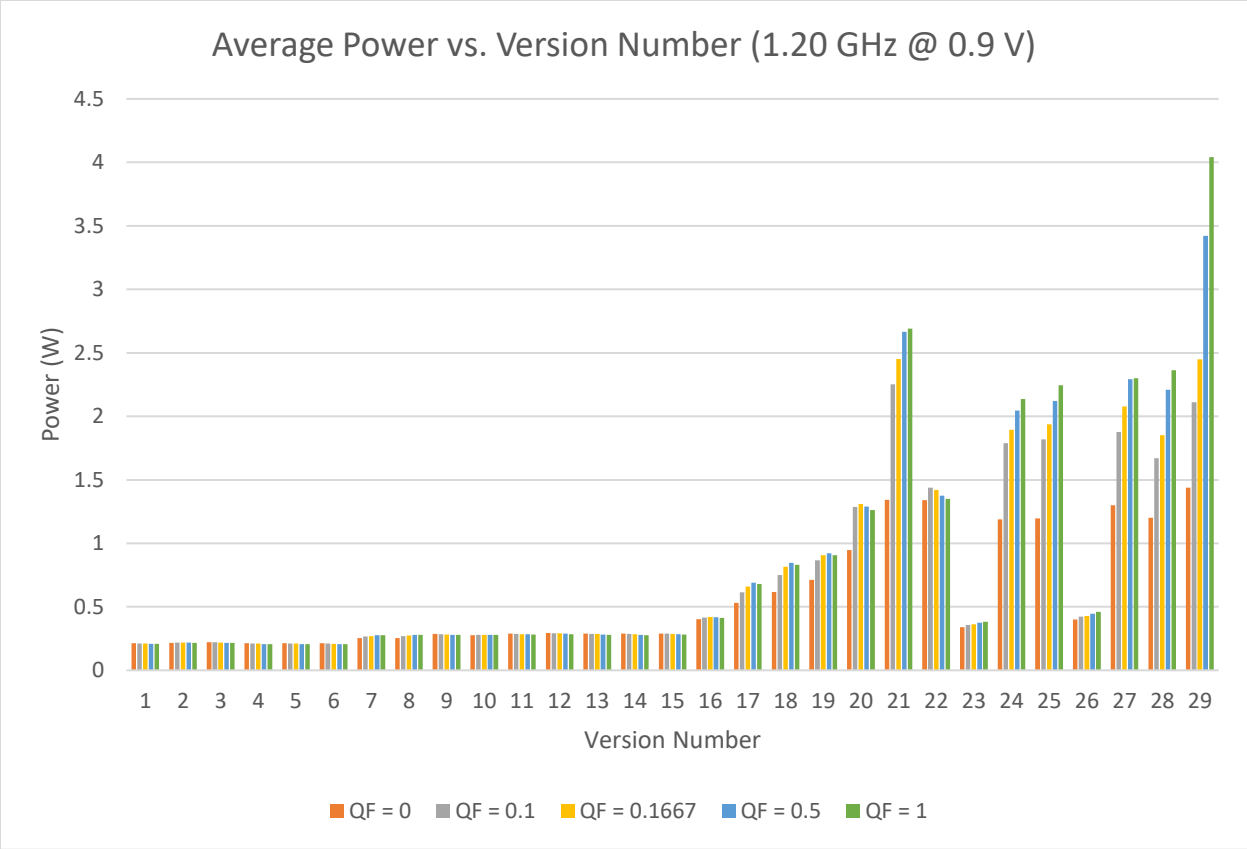


Figure 5.6: Average power versus version number (1.20 GHz @ 0.9 V)

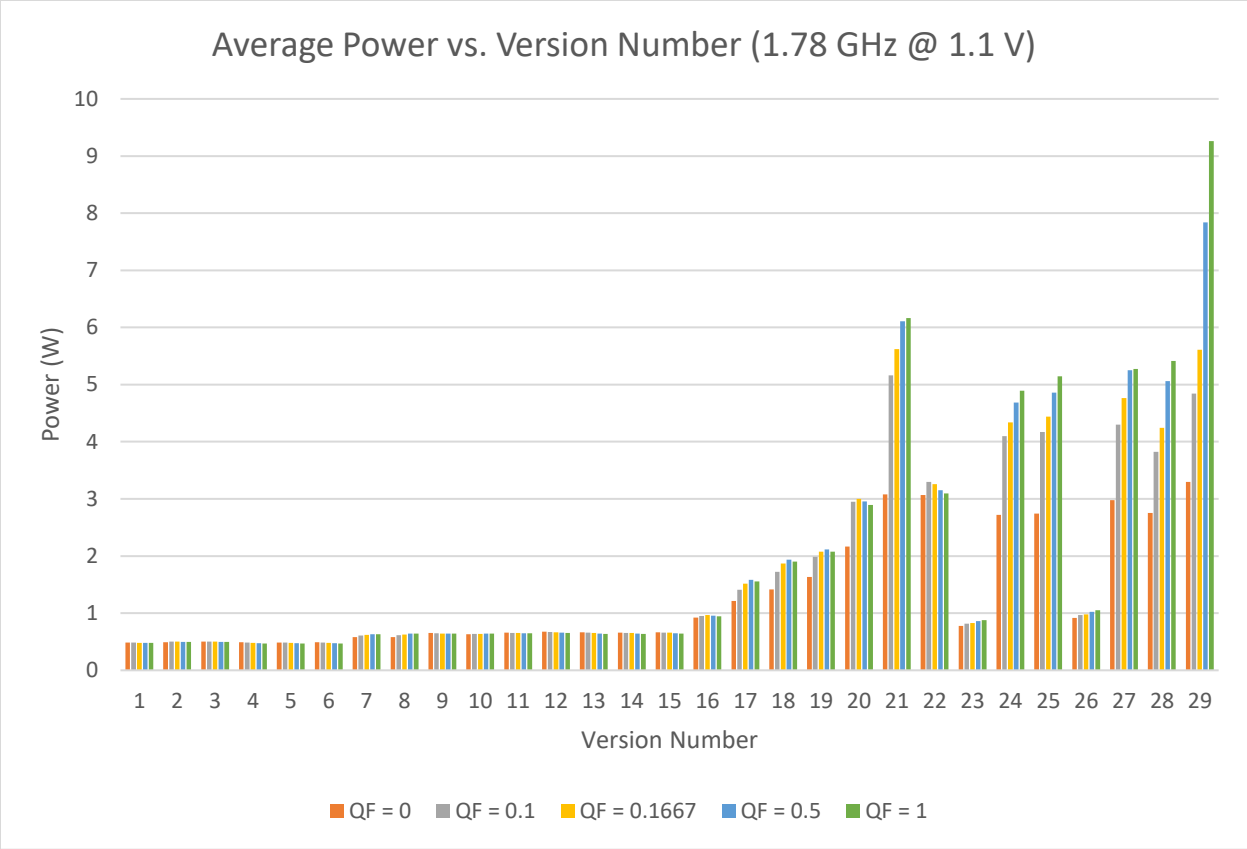


Figure 5.7: Average power versus version number (1.78 GHz @ 1.1 V)

The general trend between versions within the same quality factor is the same in Figures 5.6 and 5.7. An interesting side effect of increased throughput (and thus decreased delay) is increased power consumption as the quality factor increases. Therefore, the compression factor is getting larger, and so is the power.

Implementation 29 (1.78 GHz @ 1.1V) has the highest power of the group (9.26W when the quality factor is 1), likely due to it having the most throughput and high core usage. On the other hand, Version 1 (1.2 GHz @ .9V) has the lowest power usage at .2093 W (when the quality factor is 1). Lower power usage is not necessarily positive, as version one runs significantly longer than Version 29. In the next section, we explore energy per megapixel encoded to better understand the most energy-efficient design.

5.4 Energy per Megapixel Encoded Analysis

The following section discusses how energy per megapixel encoded changes throughout the implementations and quality levels. As described in section 5.1, three images determine the average energy per megapixel encoded of the design.

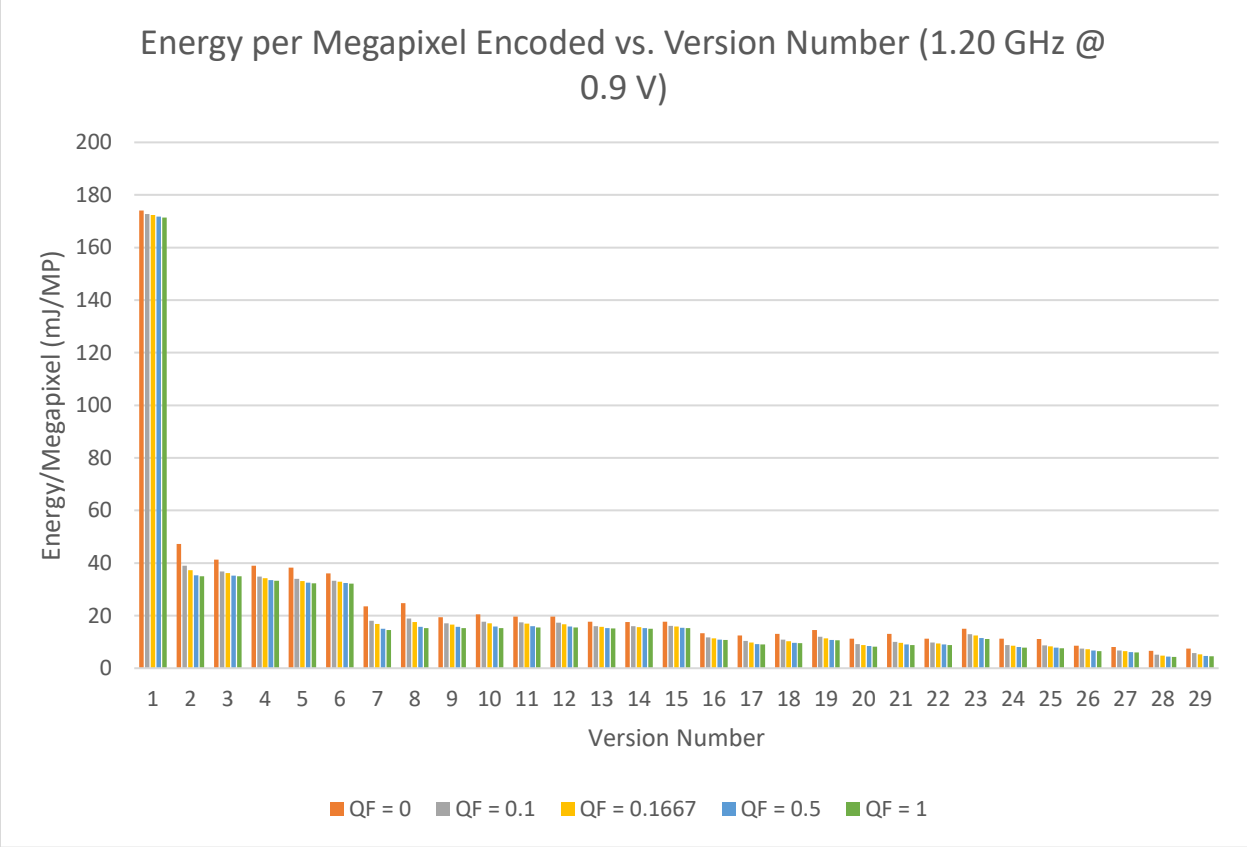


Figure 5.8: Energy per megapixel encoded versus version number (1.20 GHz @ 0.9 V)

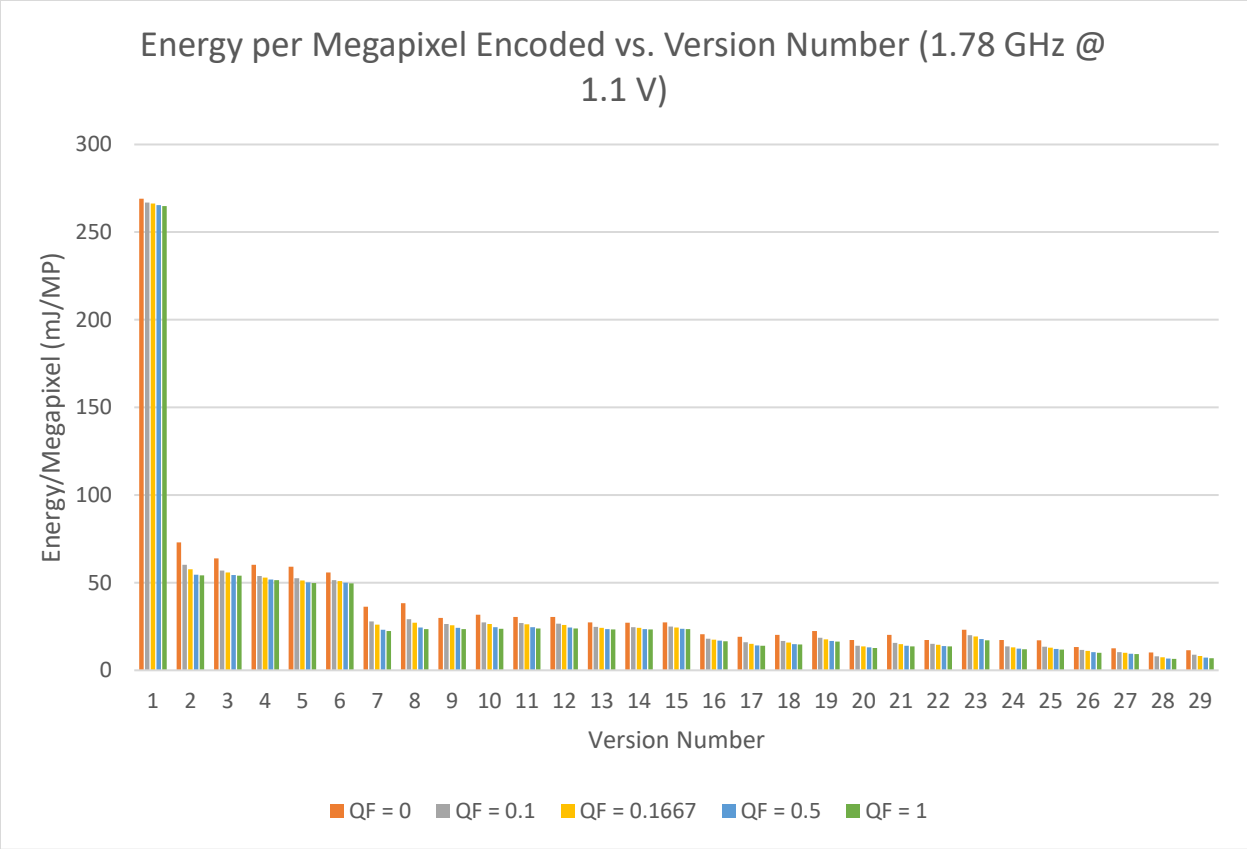


Figure 5.9: Energy per megapixel encoded versus version number (1.78 GHz @ 1.1 V)

Energy per megapixel encoded decreases over the various implementations (Figures 5.8 and 5.9), but notably, 1.2 GHz is consistently more energy efficient than 1.78 GHz. It was chosen precisely for this reason, while 1.78 GHz is the maximum clock speed. Energy per megapixel encoded also decreases with higher quality factors because lower-quality images have higher throughput and less work, making them a more energy-efficient option.

Version 28 (1.2 GHz @ 0.9 V) used just 4.24 mJ/MP (quality factor 1), making it the most energy-efficient design. On the other hand, Version 1 (1.78 GHz @ 1.1V) used 264.89 mJ/MP, making it the least energy efficient of all the designs at the same quality factor.

Version 1 uses more energy per megapixel encoded than the rest of the version due to its slower run time. Section 5.3 shows the average power of Versions 1 and 2 are about the same,

whereas the runtime of Version 1 is 4.2x longer than Version 2. Relatively constant power over a longer period of time equates to more energy per megapixel encoded. Version 2 goes from 512 to 40 multiplies and 448 to 232 additions, explaining the runtime and efficiency difference.

5.5 Throughput per Area Analysis

The following section discusses how throughput per area changes throughout implementation and quality levels. As described in section 5.1, three images determine the average throughput per area of the design.

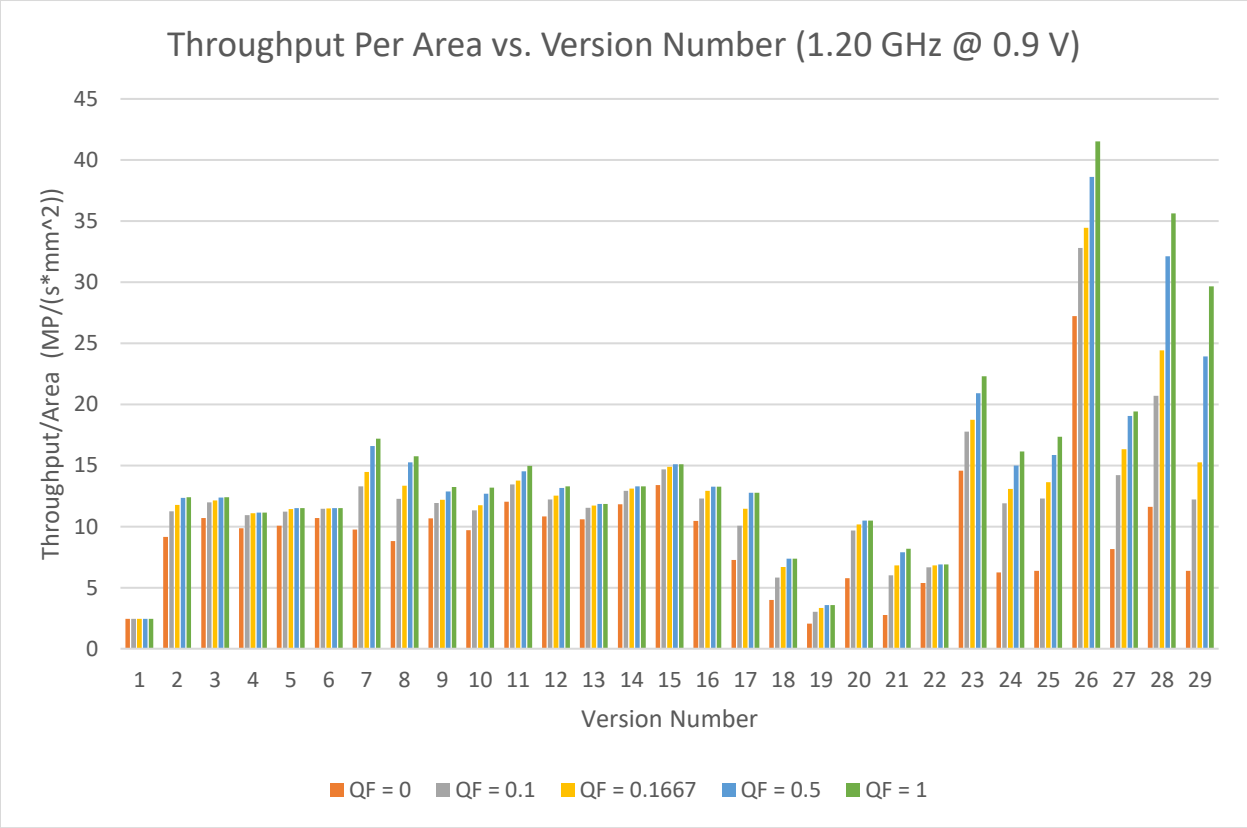


Figure 5.10: Throughput per area versus version number (1.20 GHz @ 0.9 V)

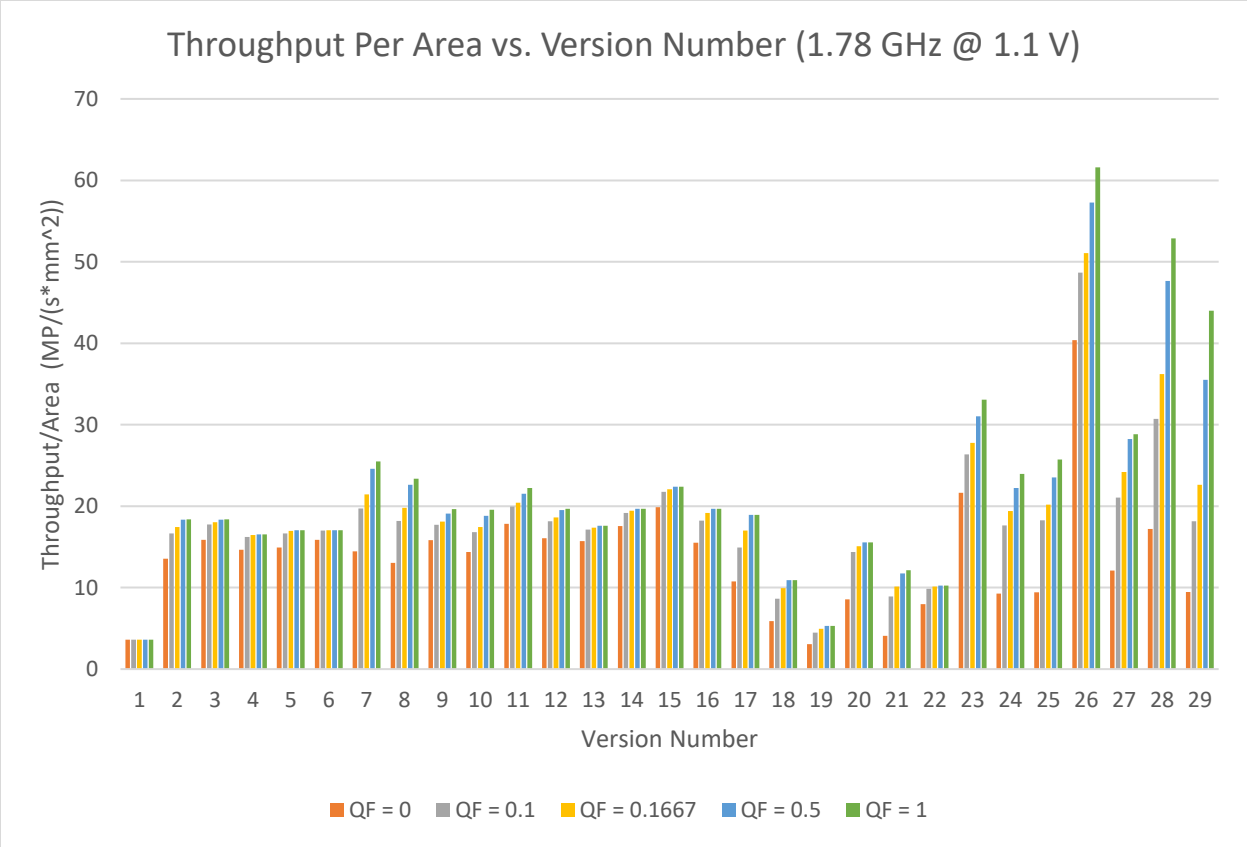


Figure 5.11: Throughput per area versus version number (1.78 GHz @ 1.1 V)

Throughput per Area (Figures 5.10 and 5.11) does not always increase with version number because implementations with only one pipeline have higher throughput per area than implementations with that pipeline scaled up. After all, while throughput scales well with pipeline count (assuming it is the same pipeline design), the area does not. Additional processing cores that organize the data coming into and out of the designs create this discrepancy. Notably, Version 28 outperforms Version 27 in throughput per area as they are the same design but with color conversion removed. Consequently, throughput increased, and area decreased between Version 27 and 28.

The highest throughput per area version was Version 26 (1.78 GHz @ 1.1V), the single pipeline version later scaled up in Version 26 through 29. Version 26 achieves 61.61

MP/(s*mm^2) using quality factor 1, whereas the least area-efficient design, Version 1 (1.20 GHz @ 0.9V), only achieves 2.45 MP/(s*mm^2) with the same quality factor.

Version 1 sees no difference between quality factors as the bottleneck is the DCT-II operation. DCT-II's workload does not depend on quality factor.

5.6 Energy-Delay Product

The following section discusses how energy-delay product changes throughout implementation and quality levels. As described in section 5.1, three images determine the average throughput per area of the design.

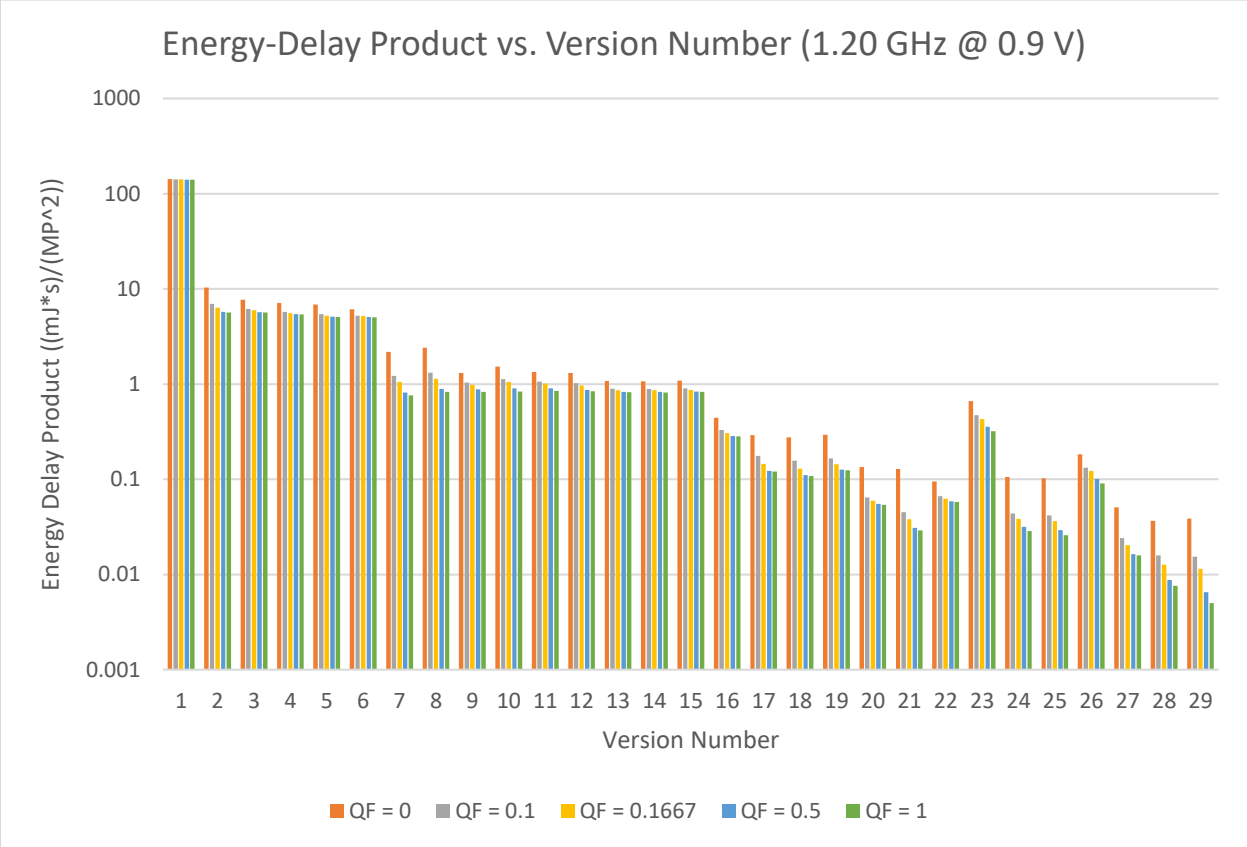


Figure 5.12: Energy-delay product versus version number (1.20 GHz @ 0.9 V)

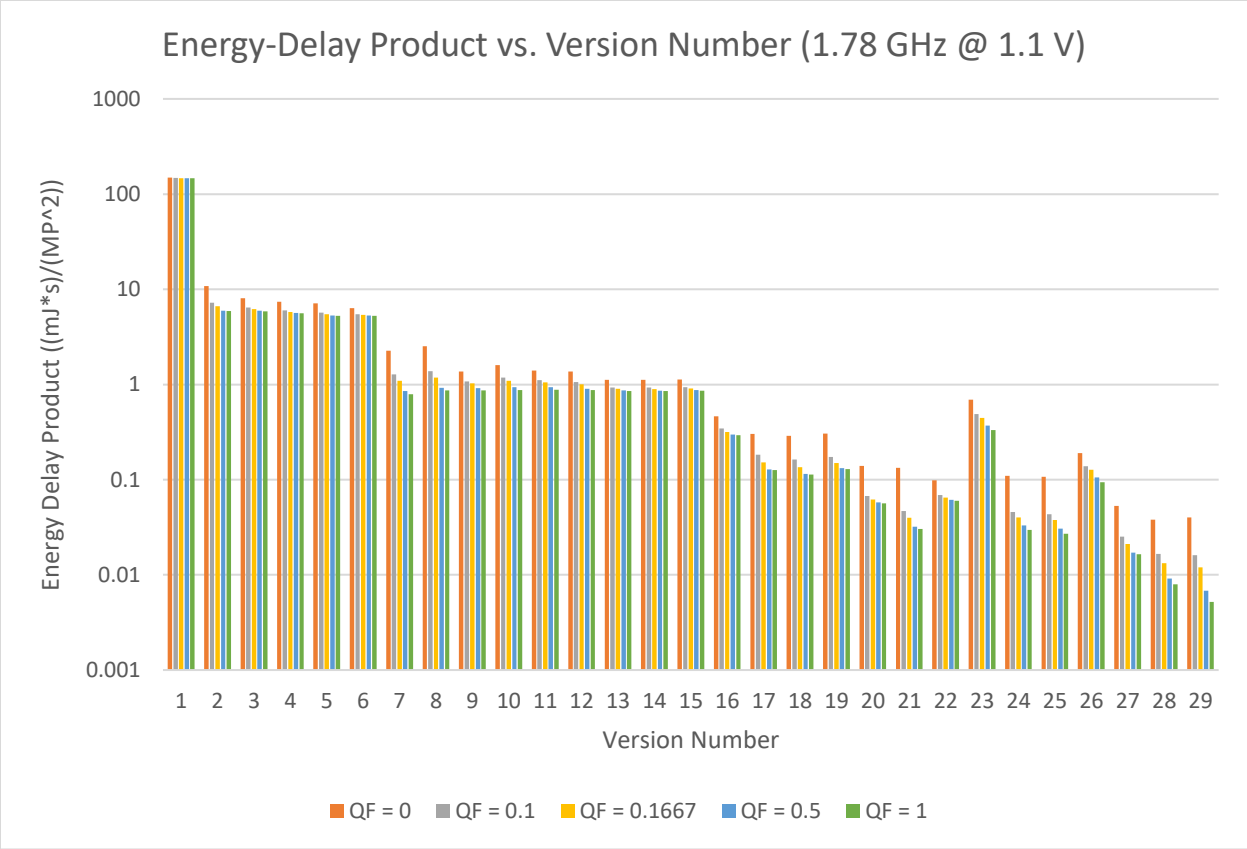


Figure 5.13: Energy-delay product versus version number (1.78 GHz @ 1.1 V)

Energy delay product across all versions and quality factors is plotted logarithmically in Figures 5.12 and 5.13. Energy delay product trends downward throughout the version-to-version innovations but is lowest at Version 29 (1.2 GHz @ 0.9 V) at 0.0050 mJ*s/(MP^2) for quality factor 1. The energy-delay product is worse for Version 1 (1.78 GHz @ 1.1V), coming in a 146.2549 uJ*s/(MP^2) using the same quality factor.

Energy delay product varies slightly between 1.2 GHz and 1.78 GHz implementations of the same version, but typically 1.2 GHz comes out slightly lower. In Version 29 specifically, the 1.2 GHz version (using quality factor 1) comes in .2 mJ*s/(MP^2) lower.

5.7 Energy per Megapixel Encoded vs. Area per Throughput Analysis

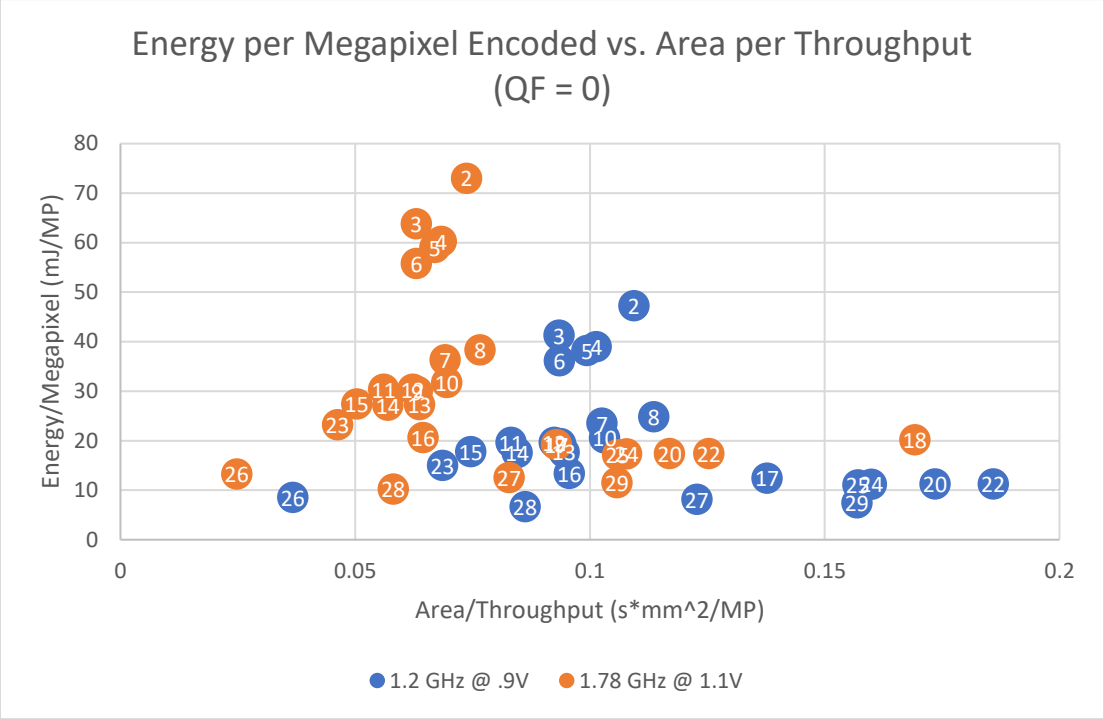


Figure 5.14: Energy per megapixel encoded versus area per throughput (QF = 0)

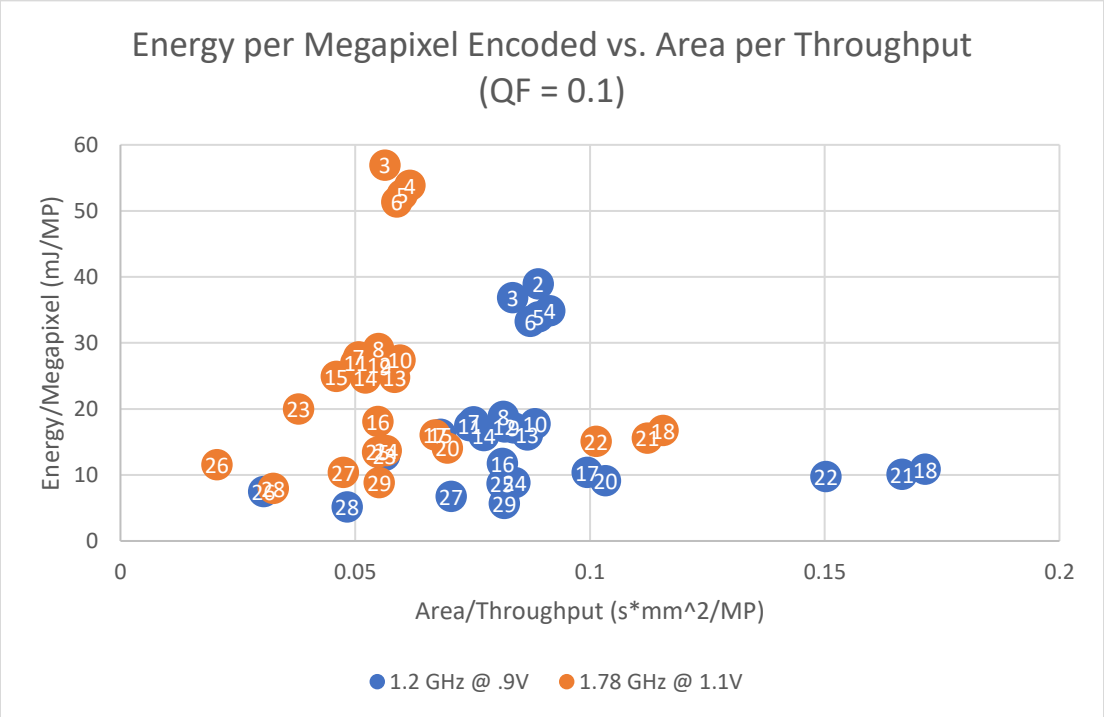


Figure 5.15: Energy per megapixel encoded versus area per throughput (QF = 0.1)

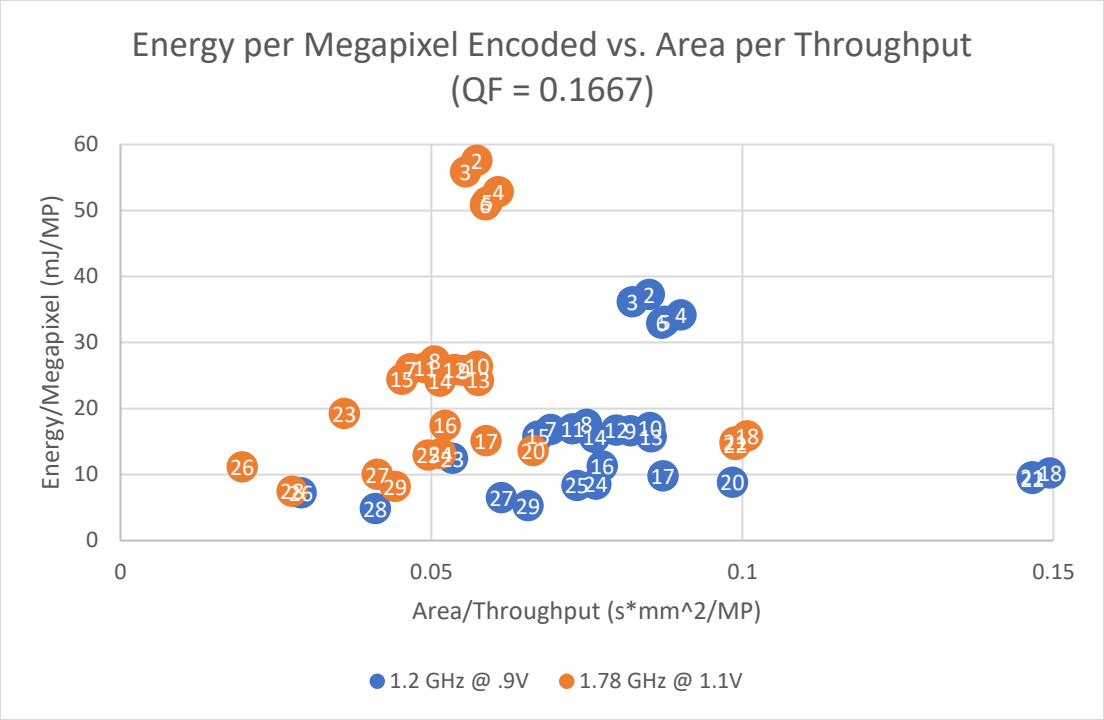


Figure 5.16: Energy per megapixel encoded versus area per throughput (QF = 0.1667)

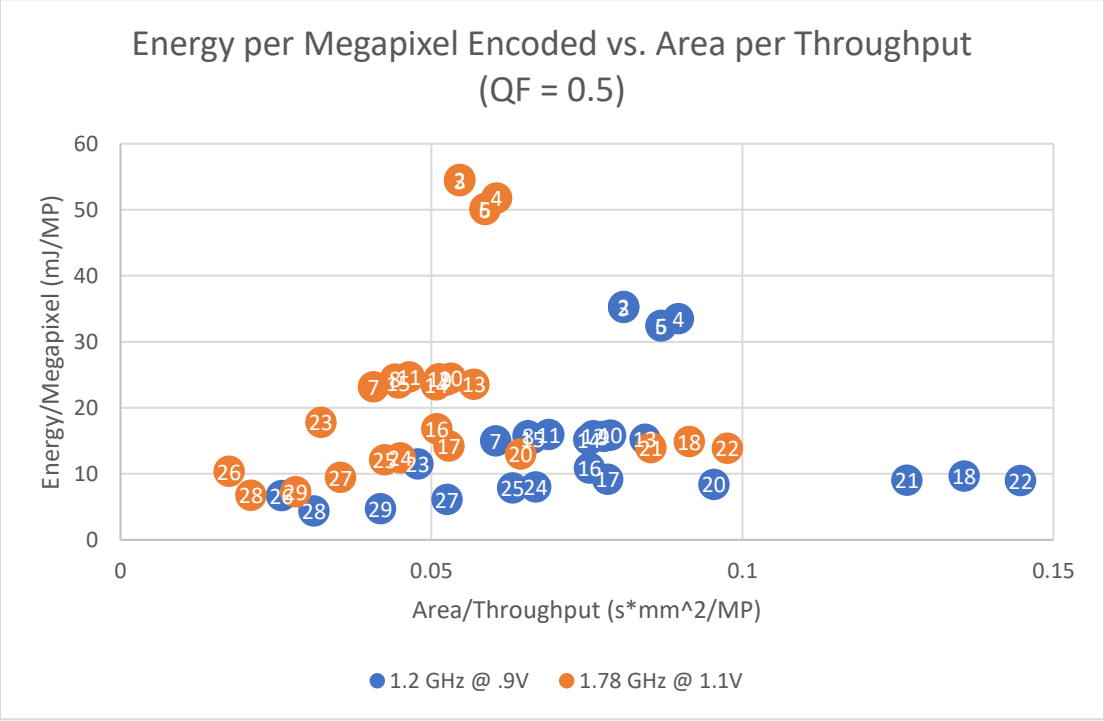


Figure 5.17: Energy per megapixel encoded versus area per throughput (QF = 0.5)

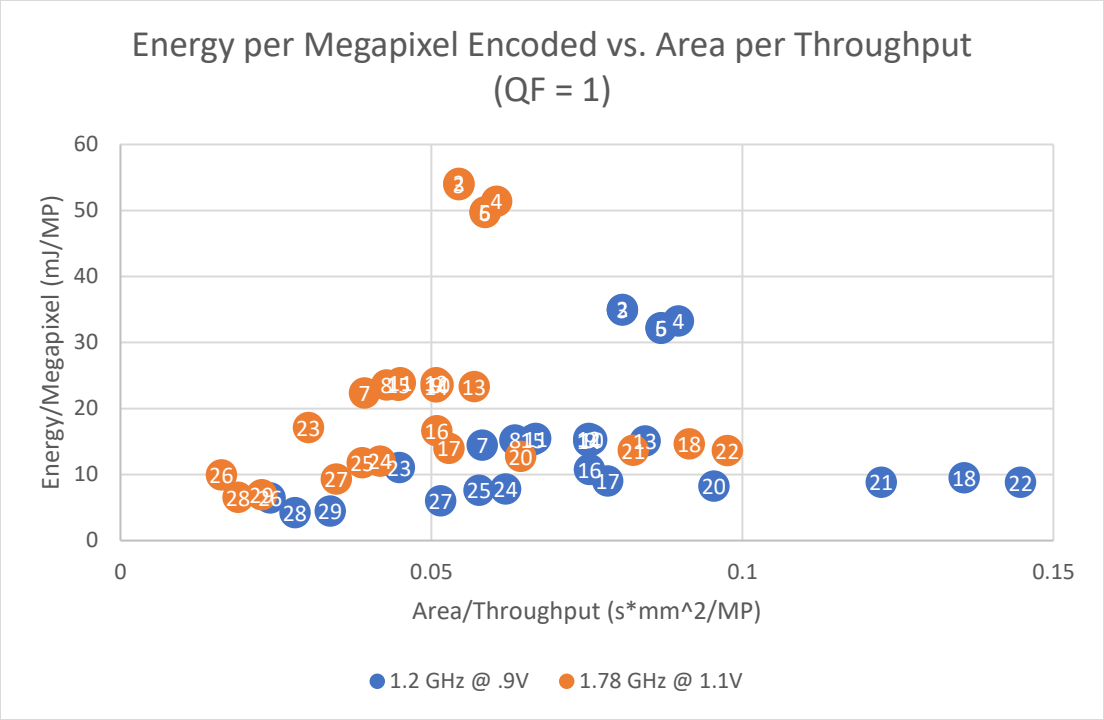


Figure 5.18: Energy per megapixel encoded versus area per throughput (QF = 1)

Energy per Megapixel Encoded vs. Area per Throughput (Figures 5.14 through 5.18) show the tradeoff between low energy and high throughput per area across the 29 versions of JPEG encoders. It is more desirable for a version (denoted by a labelled data point) to be closer to the origin as this denotes a low energy high throughput per area design. Versions 26 and 28 consistently fall closer to the origin, while Version 1 is too far to be plotted on all charts.

Versions at 1.78 GHz balance energy per megapixel encoded and area per throughput better (closer to the origin) at higher quality factors, while versions at 1.2 GHz have a better balance at lower quality factors but tend to not surpass 1.78 GHz versions.

Version Number	*Quality = 0*					
	Area (mm^2)	1.2 GHz @ 0.9V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.2210	0.2126	174.0958	2.4466	142.5849
2	0.4991	4.5644	0.2156	47.2447	9.1462	10.3507
3	0.4991	5.3416	0.2205	41.2739	10.7036	7.7268
4	0.5545	5.4715	0.2133	38.9787	9.8674	7.1240
5	0.5545	5.5802	0.2132	38.2053	10.0634	6.8466
6	0.5545	5.9318	0.2141	36.0976	10.6975	6.0855
7	1.1090	10.8123	0.2542	23.5080	9.7496	2.1742
8	1.1645	10.2504	0.2542	24.7954	8.8028	2.4190
9	1.3863	14.7857	0.2863	19.3625	10.6660	1.3095
10	1.3863	13.4430	0.2762	20.5463	9.6974	1.5284
11	1.2199	14.6638	0.2881	19.6471	12.0205	1.3398
12	1.3863	15.0060	0.2949	19.6502	10.8249	1.3095
13	1.5526	16.4349	0.2901	17.6497	10.5854	1.0739
14	1.3863	16.4039	0.2880	17.5548	11.8333	1.0702
15	1.2199	16.3470	0.2901	17.7435	13.4003	1.0854
16	2.8834	30.1579	0.4030	13.3622	10.4591	0.4431
17	5.8777	42.6734	0.5302	12.4253	7.2602	0.2912
18	11.8663	47.2878	0.6178	13.0636	3.9851	0.2763
19	23.8435	49.2760	0.7130	14.4692	2.0666	0.2936
20	14.5834	84.0618	0.9458	11.2508	5.7642	0.1338
21	37.2070	102.4844	1.3429	13.1031	2.7544	0.1279
22	22.1246	119.0326	1.3392	11.2510	5.3801	0.0945
23	1.5526	22.6370	0.3398	15.0108	14.5801	0.6631
24	16.9677	106.0952	1.1880	11.1976	6.2528	0.1055
25	16.9677	108.0474	1.1971	11.0796	6.3678	0.1025
26	1.7190	46.8120	0.4007	8.5604	27.2329	0.1829
27	19.6293	159.8874	1.2993	8.1262	8.1453	0.0508
28	15.6369	181.4568	1.2024	6.6262	11.6044	0.0365
29	30.2757	193.0376	1.4377	7.4479	6.3760	0.0386

Table 5.1: Data for 1.2 GHz @ 0.9V using quality factor 0

Version Number	*Quality = 0*					
	Area (mm^2)	1.78 GHz @ 1.1V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.8111	0.4872	268.9743	3.6292	148.5106
2	0.4991	6.7705	0.4942	72.9921	13.5668	10.7809
3	0.4991	7.9234	0.5053	63.7673	15.8770	8.0480
4	0.5545	8.1160	0.4888	60.2212	14.6367	7.4200
5	0.5545	8.2773	0.4886	59.0264	14.9274	7.1311
6	0.5545	8.7988	0.4907	55.7701	15.8679	6.3384
7	1.1090	16.0382	0.5825	36.3194	14.4619	2.2646
8	1.1645	15.2048	0.5825	38.3083	13.0575	2.5195
9	1.3863	21.9321	0.6561	29.9147	15.8212	1.3640
10	1.3863	19.9404	0.6330	31.7436	14.3844	1.5919
11	1.2199	21.7513	0.6602	30.3544	17.8304	1.3955
12	1.3863	22.2589	0.6758	30.3591	16.0569	1.3639
13	1.5526	24.3785	0.6648	27.2684	15.7017	1.1185
14	1.3863	24.3325	0.6599	27.1218	17.5527	1.1146
15	1.2199	24.2480	0.6647	27.4134	19.8771	1.1305
16	2.8834	44.7342	0.9235	20.6443	15.5144	0.4615
17	5.8777	63.2988	1.2151	19.1968	10.7693	0.3033
18	11.8663	70.1436	1.4157	20.1830	5.9112	0.2877
19	23.8435	73.0928	1.6340	22.3546	3.0655	0.3058
20	14.5834	124.6916	2.1674	17.3822	8.5503	0.1394
21	37.2070	152.0185	3.0775	20.2440	4.0858	0.1332
22	22.1246	176.5651	3.0691	17.3825	7.9805	0.0984
23	1.5526	33.5782	0.7787	23.1913	21.6271	0.6907
24	16.9677	157.3745	2.7226	17.3000	9.2749	0.1099
25	16.9677	160.2703	2.7435	17.1177	9.4456	0.1068
26	1.7190	69.4378	0.9184	13.2256	40.3955	0.1905
27	19.6293	237.1663	2.9776	12.5548	12.0823	0.0529
28	15.6369	269.1609	2.7555	10.2374	17.2132	0.0380
29	30.2757	286.3391	3.2949	11.5068	9.4577	0.0402

Table 5.2: Data for 1.78 GHz @ 1.1V using quality factor 0

Version Number	*Quality = 0.1*					
	Area (mm^2)	1.2 GHz @ 0.9V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.2210	0.2109	172.7282	2.4466	141.4648
2	0.4991	5.6092	0.2185	38.9454	11.2398	6.9431
3	0.4991	5.9730	0.2200	36.8392	11.9687	6.1677
4	0.5545	6.0626	0.2114	34.8742	10.9334	5.7524
5	0.5545	6.2264	0.2114	33.9562	11.2289	5.4536
6	0.5545	6.3516	0.2112	33.2437	11.4547	5.2339
7	1.1090	14.7420	0.2661	18.0509	13.2931	1.2245
8	1.1645	14.2785	0.2694	18.8658	12.2620	1.3213
9	1.3863	16.5482	0.2834	17.1258	11.9374	1.0349
10	1.3863	15.6990	0.2786	17.7455	11.3248	1.1304
11	1.2199	16.4022	0.2861	17.4408	13.4455	1.0633
12	1.3863	16.9454	0.2928	17.2790	12.2239	1.0197
13	1.5526	17.9213	0.2874	16.0382	11.5428	0.8949
14	1.3863	17.9110	0.2859	15.9640	12.9205	0.8913
15	1.2199	17.8930	0.2887	16.1356	14.6676	0.9018
16	2.8834	35.4394	0.4155	11.7233	12.2908	0.3308
17	5.8777	59.1340	0.6140	10.3832	10.0607	0.1756
18	11.8663	69.2270	0.7510	10.8487	5.8339	0.1567
19	23.8435	72.2233	0.8662	11.9939	3.0291	0.1661
20	14.5834	141.1890	1.2862	9.1100	9.6815	0.0645
21	37.2070	223.5339	2.2524	10.0763	6.0079	0.0451
22	22.1246	147.2507	1.4376	9.7629	6.6555	0.0663
23	1.5526	27.5962	0.3572	12.9450	17.7742	0.4691
24	16.9677	201.9905	1.7876	8.8498	11.9044	0.0438
25	16.9677	208.7887	1.8187	8.7109	12.3051	0.0417
26	1.7190	56.3868	0.4216	7.4769	32.8030	0.1326
27	19.6293	278.7267	1.8765	6.7325	14.1995	0.0242
28	15.6369	323.8956	1.6693	5.1537	20.7135	0.0159
29	30.2757	370.2318	2.1121	5.7048	12.2287	0.0154

Table 5.3: Data for 1.2 GHz @ 0.9V using quality factor 0.1

Version Number	*Quality = 0.1*					
	Area (mm^2)	1.78 GHz @ 1.1V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.8111	0.4833	266.8613	3.6292	147.3439
2	0.4991	8.3203	0.5006	60.1698	16.6723	7.2317
3	0.4991	8.8599	0.5043	56.9158	17.7535	6.4240
4	0.5545	8.9928	0.4845	53.8798	16.2179	5.9914
5	0.5545	9.2359	0.4845	52.4616	16.6562	5.6802
6	0.5545	9.4216	0.4839	51.3608	16.9911	5.4514
7	1.1090	21.8673	0.6098	27.8882	19.7181	1.2753
8	1.1645	21.1798	0.6173	29.1472	18.1886	1.3762
9	1.3863	24.5464	0.6495	26.4590	17.7071	1.0779
10	1.3863	23.2868	0.6384	27.4164	16.7984	1.1773
11	1.2199	24.3299	0.6556	26.9456	19.9442	1.1075
12	1.3863	25.1357	0.6710	26.6958	18.1322	1.0621
13	1.5526	26.5832	0.6587	24.7787	17.1218	0.9321
14	1.3863	26.5680	0.6553	24.6641	19.1654	0.9283
15	1.2199	26.5413	0.6617	24.9292	21.7570	0.9393
16	2.8834	52.5684	0.9521	18.1122	18.2314	0.3445
17	5.8777	87.7154	1.4071	16.0418	14.9234	0.1829
18	11.8663	102.6868	1.7211	16.7611	8.6536	0.1632
19	23.8435	107.1312	1.9852	18.5303	4.4931	0.1730
20	14.5834	209.4304	2.9477	14.0747	14.3609	0.0672
21	37.2070	331.5752	5.1618	15.5676	8.9116	0.0470
22	22.1246	218.4219	3.2946	15.0835	9.8724	0.0691
23	1.5526	40.9344	0.8187	19.9998	26.3651	0.4886
24	16.9677	299.6193	4.0966	13.6728	17.6582	0.0456
25	16.9677	309.7033	4.1680	13.4581	18.2525	0.0435
26	1.7190	83.6404	0.9662	11.5516	48.6578	0.1381
27	19.6293	413.4447	4.3005	10.4016	21.0626	0.0252
28	15.6369	480.4451	3.8255	7.9623	30.7251	0.0166
29	30.2757	549.1771	4.8404	8.8138	18.1392	0.0160

Table 5.4: Data for 1.78 GHz @ 1.1V using quality factor 0.1

Version Number	*Quality = 0.1667*					
	Area (mm^2)	1.2 GHz @ 0.9V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.2210	0.2105	172.3698	2.4466	141.1712
2	0.4991	5.8712	0.2189	37.2764	11.7648	6.3490
3	0.4991	6.0632	0.2193	36.1675	12.1496	5.9650
4	0.5545	6.1529	0.2104	34.1969	11.0963	5.5578
5	0.5545	6.3370	0.2104	33.2063	11.4283	5.2401
6	0.5545	6.3709	0.2098	32.9298	11.4894	5.1688
7	1.1090	16.0332	0.2701	16.8480	14.4573	1.0508
8	1.1645	15.5404	0.2739	17.6243	13.3457	1.1341
9	1.3863	16.9076	0.2816	16.6549	12.1966	0.9851
10	1.3863	16.2835	0.2786	17.1086	11.7464	1.0507
11	1.2199	16.7843	0.2845	16.9480	13.7588	1.0098
12	1.3863	17.3873	0.2913	16.7535	12.5427	0.9635
13	1.5526	18.1923	0.2859	15.7131	11.7173	0.8637
14	1.3863	18.1845	0.2845	15.6463	13.1178	0.8604
15	1.2199	18.1708	0.2874	15.8180	14.8954	0.8705
16	2.8834	37.2292	0.4215	11.3207	12.9116	0.3041
17	5.8777	67.3941	0.6608	9.8051	11.4661	0.1455
18	11.8663	79.3939	0.8152	10.2679	6.6907	0.1293
19	23.8435	79.3847	0.9054	11.4055	3.3294	0.1437
20	14.5834	148.2313	1.3103	8.8392	10.1644	0.0596
21	37.2070	253.8204	2.4525	9.6622	6.8219	0.0381
22	22.1246	150.8911	1.4206	9.4148	6.8201	0.0624
23	1.5526	29.0750	0.3622	12.4590	18.7267	0.4285
24	16.9677	222.0329	1.8938	8.5292	13.0856	0.0384
25	16.9677	231.1969	1.9364	8.3754	13.6257	0.0362
26	1.7190	59.2018	0.4278	7.2267	34.4407	0.1221
27	19.6293	320.4217	2.0794	6.4895	16.3236	0.0203
28	15.6369	381.7691	1.8506	4.8474	24.4146	0.0127
29	30.2757	461.9862	2.4489	5.3008	15.2593	0.0115

Table 5.5: Data for 1.2 GHz @ 0.9V using quality factor 0.1667

Version Number	*Quality = 0.1667*					
	Area (mm^2)	1.78 GHz @ 1.1V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.8111	0.4823	266.3077	3.6292	147.0381
2	0.4991	8.7090	0.5016	57.5913	17.4511	6.6129
3	0.4991	8.9938	0.5026	55.8781	18.0219	6.2129
4	0.5545	9.1268	0.4822	52.8335	16.4595	5.7888
5	0.5545	9.3999	0.4822	51.3031	16.9520	5.4578
6	0.5545	9.4502	0.4808	50.8758	17.0427	5.3836
7	1.1090	23.7825	0.6191	26.0298	21.4450	1.0945
8	1.1645	23.0516	0.6277	27.2292	19.7962	1.1812
9	1.3863	25.0796	0.6453	25.7315	18.0917	1.0260
10	1.3863	24.1539	0.6384	26.4324	17.4239	1.0943
11	1.2199	24.8968	0.6519	26.1844	20.4089	1.0517
12	1.3863	25.7912	0.6676	25.8839	18.6050	1.0036
13	1.5526	26.9852	0.6551	24.2765	17.3807	0.8996
14	1.3863	26.9737	0.6520	24.1733	19.4580	0.8962
15	1.2199	26.9534	0.6587	24.4384	22.0948	0.9067
16	2.8834	55.2233	0.9659	17.4902	19.1521	0.3167
17	5.8777	99.9679	1.5144	15.1487	17.0080	0.1515
18	11.8663	117.7677	1.8682	15.8636	9.9245	0.1347
19	23.8435	117.7540	2.0750	17.6212	4.9386	0.1496
20	14.5834	219.8764	3.0027	13.6564	15.0772	0.0621
21	37.2070	376.5002	5.6204	14.9279	10.1191	0.0396
22	22.1246	223.8217	3.2556	14.5456	10.1164	0.0650
23	1.5526	43.1280	0.8302	19.2488	27.7779	0.4463
24	16.9677	329.3487	4.3400	13.1775	19.4103	0.0400
25	16.9677	342.9421	4.4376	12.9398	20.2115	0.0377
26	1.7190	87.8160	0.9805	11.1652	51.0870	0.1271
27	19.6293	475.2922	4.7654	10.0262	24.2134	0.0211
28	15.6369	566.2908	4.2410	7.4891	36.2150	0.0132
29	30.2757	685.2795	5.6122	8.1896	22.6346	0.0120

Table 5.6: Data for 1.78 GHz @ 1.1V Using quality factor 0.1667

Version Number	*Quality = 0.5*					
	Area (mm^2)	1.2 GHz @ 0.9V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.2210	0.2097	171.7324	2.4466	140.6493
2	0.4991	6.1680	0.2180	35.3493	12.3596	5.7310
3	0.4991	6.1692	0.2172	35.2113	12.3620	5.7075
4	0.5545	6.1821	0.2073	33.5300	11.1490	5.4237
5	0.5545	6.3785	0.2073	32.4962	11.5032	5.0946
6	0.5545	6.3785	0.2067	32.4055	11.5032	5.0804
7	1.1090	18.3853	0.2762	15.0204	16.5782	0.8170
8	1.1645	17.7709	0.2803	15.7716	15.2612	0.8875
9	1.3863	17.8381	0.2798	15.6877	12.8679	0.8794
10	1.3863	17.5946	0.2793	15.8729	12.6922	0.9021
11	1.2199	17.7155	0.2830	15.9748	14.5221	0.9017
12	1.3863	18.2398	0.2883	15.8062	13.1577	0.8666
13	1.5526	18.4103	0.2811	15.2685	11.8577	0.8293
14	1.3863	18.4103	0.2801	15.2162	13.2807	0.8265
15	1.2199	18.4091	0.2832	15.3856	15.0907	0.8358
16	2.8834	38.2378	0.4171	10.9068	13.2613	0.2852
17	5.8777	75.0399	0.6908	9.2053	12.7669	0.1227
18	11.8663	87.4887	0.8453	9.6617	7.3729	0.1104
19	23.8435	85.3746	0.9223	10.8027	3.5806	0.1265
20	14.5834	152.8550	1.2903	8.4414	10.4815	0.0552
21	37.2070	294.2089	2.6646	9.0567	7.9074	0.0308
22	22.1246	152.8756	1.3746	8.9916	6.9098	0.0588
23	1.5526	32.4776	0.3747	11.5378	20.9182	0.3553
24	16.9677	254.1803	2.0455	8.0476	14.9802	0.0317
25	16.9677	269.1876	2.1209	7.8788	15.8647	0.0293
26	1.7190	66.3594	0.4460	6.7207	38.6046	0.1013
27	19.6293	373.9308	2.2927	6.1313	19.0496	0.0164
28	15.6369	502.3248	2.2089	4.3973	32.1243	0.0088
29	30.2757	724.4761	3.4215	4.7227	23.9293	0.0065

Table 5.7: Data for 1.2 GHz @ 0.9V using quality factor 0.5

Version Number	*Quality = 0.5*					
	Area (mm^2)	1.78 GHz @ 1.1V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.8111	0.4805	265.3229	3.6292	146.4945
2	0.4991	9.1493	0.4997	54.6139	18.3333	5.9692
3	0.4991	9.1510	0.4978	54.4006	18.3369	5.9447
4	0.5545	9.1701	0.4750	51.8032	16.5377	5.6491
5	0.5545	9.4615	0.4750	50.2059	17.0631	5.3063
6	0.5545	9.4615	0.4737	50.0659	17.0631	5.2915
7	1.1090	27.2715	0.6329	23.2062	24.5911	0.8509
8	1.1645	26.3602	0.6423	24.3668	22.6375	0.9244
9	1.3863	26.4599	0.6413	24.2372	19.0874	0.9160
10	1.3863	26.0986	0.6400	24.5233	18.8268	0.9396
11	1.2199	26.2779	0.6486	24.6808	21.5411	0.9392
12	1.3863	27.0558	0.6607	24.4202	19.5172	0.9026
13	1.5526	27.3086	0.6442	23.5896	17.5890	0.8638
14	1.3863	27.3086	0.6420	23.5088	19.6997	0.8609
15	1.2199	27.3069	0.6491	23.7705	22.3845	0.8705
16	2.8834	56.7194	0.9558	16.8507	19.6710	0.2971
17	5.8777	111.3092	1.5830	14.2220	18.9375	0.1278
18	11.8663	129.7750	1.9372	14.9272	10.9364	0.1150
19	23.8435	126.6389	2.1136	16.6899	5.3113	0.1318
20	14.5834	226.7349	2.9570	13.0417	15.5475	0.0575
21	37.2070	436.4098	6.1064	13.9924	11.7293	0.0321
22	22.1246	226.7654	3.1502	13.8918	10.2495	0.0613
23	1.5526	48.1751	0.8588	17.8257	31.0287	0.3700
24	16.9677	377.0341	4.6878	12.4333	22.2207	0.0330
25	16.9677	399.2950	4.8604	12.1725	23.5327	0.0305
26	1.7190	98.4331	1.0221	10.3834	57.2635	0.1055
27	19.6293	554.6641	5.2542	9.4727	28.2569	0.0171
28	15.6369	745.1151	5.0621	6.7937	47.6511	0.0091
29	30.2757	1,074.6395	7.8412	7.2965	35.4951	0.0068

Table 5.8: Data for 1.78 GHz @ 1.1V using quality factor 0.5

Version Number	*Quality = 1*					
	Area (mm^2)	1.2 GHz @ 0.9V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.2210	0.2093	171.4515	2.4466	140.4192
2	0.4991	6.1820	0.2166	35.0325	12.3876	5.6668
3	0.4991	6.1820	0.2158	34.9120	12.3876	5.6473
4	0.5545	6.1821	0.2057	33.2752	11.1490	5.3825
5	0.5545	6.3785	0.2056	32.2401	11.5032	5.0545
6	0.5545	6.3785	0.2052	32.1680	11.5032	5.0432
7	1.1090	19.0574	0.2760	14.4838	17.1843	0.7600
8	1.1645	18.3550	0.2799	15.2475	15.7628	0.8307
9	1.3863	18.3480	0.2793	15.2226	13.2357	0.8297
10	1.3863	18.2955	0.2797	15.2870	13.1978	0.8356
11	1.2199	18.2732	0.2828	15.4776	14.9793	0.8470
12	1.3863	18.4090	0.2850	15.4815	13.2797	0.8410
13	1.5526	18.4092	0.2780	15.1032	11.8570	0.8204
14	1.3863	18.4091	0.2773	15.0610	13.2798	0.8181
15	1.2199	18.4091	0.2806	15.2411	15.0907	0.8279
16	2.8834	38.2382	0.4115	10.7622	13.2615	0.2815
17	5.8777	75.0399	0.6793	9.0528	12.7669	0.1206
18	11.8663	87.4843	0.8310	9.4983	7.3725	0.1086
19	23.8435	85.3703	0.9055	10.6066	3.5804	0.1242
20	14.5834	152.8550	1.2620	8.2565	10.4815	0.0540
21	37.2070	304.2217	2.6906	8.8443	8.1765	0.0291
22	22.1246	152.8756	1.3500	8.8305	6.9098	0.0578
23	1.5526	34.6272	0.3834	11.0735	22.3027	0.3198
24	16.9677	273.9292	2.1363	7.7989	16.1442	0.0285
25	16.9677	294.3106	2.2449	7.6275	17.3453	0.0259
26	1.7190	71.3936	0.4600	6.4429	41.5333	0.0902
27	19.6293	381.3416	2.3005	6.0327	19.4272	0.0158
28	15.6369	557.2255	2.3632	4.2411	35.6353	0.0076
29	30.2757	898.1440	4.0407	4.4989	29.6655	0.0050

Table 5.9: Data for 1.2 GHz @ 0.9V using quality factor 1

Version Number	*Quality = 1*					
	Area (mm^2)	1.78 GHz @ 1.1V				
		Average Throughput (MP/s)	Average Power (W)	Energy per Megapixel Enc. (mJ/MP)	Throughput per Area (MP/(s*mm^2))	Energy-Delay Product (mJ*s/(MP^2))
1	0.4991	1.8111	0.4798	264.8889	3.6292	146.2549
2	0.4991	9.1700	0.4963	54.1245	18.3750	5.9023
3	0.4991	9.1700	0.4946	53.9383	18.3750	5.8820
4	0.5545	9.1701	0.4714	51.4095	16.5377	5.6062
5	0.5545	9.4615	0.4713	49.8103	17.0631	5.2645
6	0.5545	9.4615	0.4702	49.6989	17.0631	5.2527
7	1.1090	28.2684	0.6326	22.3772	25.4900	0.7916
8	1.1645	27.2265	0.6414	23.5571	23.3814	0.8652
9	1.3863	27.2162	0.6401	23.5186	19.6330	0.8641
10	1.3863	27.1383	0.6410	23.6181	19.5768	0.8703
11	1.2199	27.1053	0.6482	23.9126	22.2193	0.8822
12	1.3863	27.3067	0.6531	23.9186	19.6983	0.8759
13	1.5526	27.3070	0.6372	23.3342	17.5879	0.8545
14	1.3863	27.3069	0.6354	23.2689	19.6984	0.8521
15	1.2199	27.3069	0.6430	23.5471	22.3845	0.8623
16	2.8834	56.7200	0.9431	16.6274	19.6712	0.2931
17	5.8777	111.3092	1.5568	13.9864	18.9375	0.1257
18	11.8663	129.7683	1.9043	14.6747	10.9359	0.1131
19	23.8435	126.6326	2.0751	16.3870	5.3110	0.1294
20	14.5834	226.7349	2.8923	12.7561	15.5475	0.0563
21	37.2070	451.2622	6.1662	13.6642	12.1284	0.0303
22	22.1246	226.7654	3.0938	13.6430	10.2495	0.0602
23	1.5526	51.3637	0.8787	17.1083	33.0824	0.3331
24	16.9677	406.3283	4.8959	12.0491	23.9472	0.0297
25	16.9677	436.5607	5.1446	11.7843	25.7289	0.0270
26	1.7190	105.9006	1.0541	9.9541	61.6077	0.0940
27	19.6293	565.6567	5.2722	9.3205	28.8170	0.0165
28	15.6369	826.5511	5.4159	6.5524	52.8590	0.0079
29	30.2757	1,332.2469	9.2601	6.9507	44.0038	0.0052

Table 5.10: Data for 1.78 GHz @ 1.1V using quality factor 1

Chapter 6

Comparisons to Other Notable JPEG Encoders

6.1 Overview

Performance metrics reported in Chapter 5, throughput per area, energy, and energy-delay product, are compared to competitive industry designs in this chapter. These metrics represent low-power and high-throughput implementations.

6.2 Comparison of JPEG Encoder KiloCore Implementations with Competing Designs

6.2.1 Overview

To represent all platforms on a level playing field, all designs have been scaled to 32nm. If data is ambiguous for a given design, the narrowest range deducible is provided to keep the design in the comparisons. In addition, predictive polynomial models can scale CMOS device performance accurately between voltages and technology. The following two equations (6.1 and 6.2) are required to scale both delay (used in throughput) and energy [9]:

$$DelayFactor = a_{d3}V^3 + a_{d2}V^2 + a_{d1}V + a_{d0} \quad (6.1)$$

$$EnergyFactor = a_{e2}V^2 + a_{e1}V + a_{e0} \quad (6.2)$$

Equations 6.1 and 6.2 are used with the following equations to scale Delay and Energy [9]:

$$Delay_x = \frac{DelayFactor_x}{DelayFactor_y} * Delay_y \quad (6.3)$$

$$Energy_x = \frac{EnergyFactor_x}{EnergyFactor_y} * Energy_y \tag{6.4}$$

The following table (6.1) and equation (6.5) are provided for area scaling calculations [9]:

Technology Node	*Scale Factor*
32 nm	1
45 nm	0.46
20 nm	2.2
14 nm	2.7
7 nm	7.8

Table 6.1: Area scaling factors

$$Area_x = AreaFactor_y * Area_y \tag{6.5}$$

After reviewing the results of Chapter 5, Versions 26 and 29 of the JPEG encoders stood out as stronger contenders than the rest of the implementations. Version 26 lacks high throughput but has great throughput per area, and Version 29 has the most competitive throughput and energy metrics. Alongside KiloCore implementations are four competing designs. First, Texas Instruments provides a low-power JPEG encoder implemented on the C66x digital signal processor [10,11]. Second, VISENGI offers an FPGA JPEG implementation profiled on the Xilinx Zynq 7020 FPGA [12,13]. Third, libjpeg-turbo (v2.1.5.1) is considered the standard implementation on x86-based platforms and is profiled on an Intel i9 9900 [8,14]. The Intel i9 9900 was not chosen for any particular reason. Finally, Nvidia's nvJPEG library is profiled on the A100 architecture with an Intel Xeon Platinum 8168 [15,16,7,18]. The A100 GPU and Intel Xeon Platinum were chosen as Nvidia uses them in their promotional material for nvJPEG. The GPU computes DCT-II, quantization, and run-length and Huffman encoding, while the CPU handles difference encoding and concatenating the bit streams. The following table includes all found metrics and ranges:

Vendor	*Architecture*	*Technology Node (nm)*	*Area (mm^2)*	*Clock Frequency (GHz)*	*Throughput (MP/s)*	*Energy per Megapixel Enc. (uJ/MP)*
TI	Embedded C	40	6.48	1.25	66.66	20,252
VISENGI	FPGA	28	16.261	0.200	533	375
libjpeg-turbo1.5	CPU	14	175.780	5.00	152.790	98,170
Nvidia	GPU+CPU	7 + 14	826 - 5120	1.41	5438	73,600 – 111,300
KiloCore v29 0.9V	CPU	32	30.276	1.20	370.232	5,705
KiloCore v29 1.1V	CPU	32	30.276	1.78	549.177	8,814
KiloCore v26 0.9V	CPU	32	**1.719**	1.20	56.387	7,477
KiloCore v26 1.1V	CPU	32	**1.719**	1.78	83.640	11,552

Table 6.2: Unscaled comparison data for various JPEG encoder implementations

TI's implementation assumes one DSP core is used and operating at average power consumption [10]. Nvidia's implementation has a scale for area and energy per pixel, as it is unclear how much the CPU assists during the JPEG encoding process. The lower bound demonstrates just the GPU, whereas the upper bound demonstrates the GPU with 80% of the CPU. VISENGI does not have public performance information, but after reaching out, they agreed to provide the information listed in Table 6.2; they warn that power and energy information can vary between which FPGA is chosen for their IP core. There is no public data for the size of a register slice or a LUT slice for the Xilinx FPGA, so an unweighted average was taken of the memory, registers slices, and LUT slices used to make a crude approximation of area utilized (5.63%). Since only one of eight cores is used in the libjpeg-turbo implementation, area and energy metrics have been scaled down by 8x. All performance data represents 4:4:4 subsampling and a 10:1 image compression factor (quality factor 0.1). Unfortunately, Nvidia did not post what compression factor or quality they used to achieve the represented numbers. They could have used a lower quality to drive a higher throughput.

Delay and energy factors were calculated for each unique technology node using Equations 6.1 and 6.2 and are tabulated below:

Technology Node	*Delay Coefficients*				*Delay Factor*
	ad3	ad2	ad1	ad0	
32 nm HP @ 0.9V	-1047	2982	-2797	873.5	8.357
45 nm HP @ 1.0V	-501.6	1567	-1619	566.1	12.5
20 nm HP @ 1.0V	0	34.63	-66.37	41.15	9.41
14 nm HP @ 1.0V	-40.66	109.2	-100.6	35.92	3.86
7 nm HP @ 1.0V	-28.58	76.6	-70.26	24.69	2.45

Table 6.3: Delay factor calculations using Equation 6.1

Technology Node	*Energy Coefficients*			*Energy Factor*
	ae2	ae1	ae0	
32 nm HP @ 0.9V	0.8367	-0.4341	0.1701	0.457
45 nm HP @ 1.0V	1.018	-0.3107	0.1539	0.861
20 nm HP @ 1.0V	0.373	-0.1582	0.04104	0.256
14 nm HP @ 1.0V	0.2363	-0.09675	0.02239	0.162
7 nm HP @ 1.0V	0.1776	-0.09097	0.02447	0.111

Table 6.4: Energy factor calculations using Equation 6.2

Table 6.2 represents scaled performance metrics to 32 nm HP, using Tables 6.1,6.3, and 6.4 along with Equations 6.3 through 6.5.

Vendor	*Scaled Area (mm^2)*	*Clock Frequency (GHz)*	*Quality Factor*	*Throughput (MP/s)*	*Energy per Megapixel Enc. (uJ/MP)*	*Throughput per Area (MP / (s*mm^2))*	*Energy-Delay Product (uJ*s / (MP^2))*
TI	2.98 - 6.48	1.25 - 1.870	~0.1	66.66 - 99.71	10,749 – 38,155	10.287 – 33.460	304 – 383
VISENGI	16.261 - 35.774	0.200 - 0.225	~0.1	533-600	**375 - 670**	16.772 - 32.778	**0.704 – 1.117**
libjpeg-turbo1.5	474.61	**2.30**	0.1	70.57	277,036	0.149	3,926
Nvidia	6442.8 - 18036.6	0.41	N/A	**1594.25**	302,842 – 409,265	0.088 - 0.247	190 - 257
KiloCore v29 0.9V	30.276	1.20	0.1	370.23	5,705	12.228	15
KiloCore v29 1.1V	30.276	1.78	0.1	549.18	8,814	18.139	16
KiloCore v26 0.9V	**1.719**	1.20	0.1	56.39	7,477	32.804	133
KiloCore v26 1.1V	**1.719**	1.78	0.1	83.64	11,552	**48.656**	138

Table 6.5: Scaled comparison data for various JPEG encoder implementations

6.2.2 Area Analysis

KiloCore Version 26 has the lowest area by at least 1.73 times (Table 6.5). KiloCore can run multiple programs simultaneously; thus, a lower-area version of the JPEG encoding algorithm can prove helpful in specific applications. Besides TI's DSP and VISENGI's FPGA design, most designs are multiple magnitudes larger than both KiloCore versions showcased.

KiloCore implementations have higher yields due to smaller area; therefore, KiloCore designs are more cost-effective solutions. Due to KiloCore's smaller silicon footprint, SOC designers will find it easier to add KiloCore as an integrated IP component on their SOC.

6.2.3 Throughput Analysis

Table 6.5 shows Nvidia has the greatest throughput of every implementation, with 1.594 GP/s. However, Nvidia did not provide compression ratio information, and a lower quality ratio was likely used to highlight the speed of using the A100 (as higher qualities would stress the CPU since it handles the end of the Huffman encoding algorithm). If that is the case, it should be compared to Version 29 (1.78 GHz @ 1.1V) using quality factor 1. In this case, Version 29 has

1.332 GP/s throughput, much closer to Nvidia's implementation. VISENGI has the same throughput as Version 29 (1.78 GHz @ 1.1V). Still, due to scaling information being unavailable for the 20nm technology node, it is impossible to know what design has a higher throughput.

KiloCore Version 29's throughput makes it a more competitive option in situations that demand large amounts of photos to be processed at once, such as data centers. Computers used for video or photo editing will also benefit from KiloCore Version 29's throughput as encoding times will be reduced over standard general-purpose CPUs like the compared i9 9900.

6.2.4 Energy per Megapixel Encoded Analysis

VISENGI's implementation has the lowest energy per megapixel encoded; at most, it is 670 (uJ)/MP. Although KiloCore (1.2 GHz @ 0.9V) is at least 8.5x less energy efficient than VISENGI's implementation, it handily beats the DSP, x86, and GPU implementations by 1.88x (at least), 48x, and 53x (at least) respectively.

FPGA's have lower power consumption than general-purpose processors; therefore, it VISENGI's performance metrics really serve as a best-case scenario for power consumption as it would be rare for a general-purpose processor to match the power consumption of application specific hardware. KiloCore JPEG encoders' low energy consumption relative to the other general-purpose processors makes it a more compelling option in data center environments where the system would be running consistently. In these data centers, saving money on power usage is essential.

6.2.5 Throughput per Area Analysis

KiloCore Version 26 (1.78 GHz @ 1.1V) has the highest throughput per area of all the designs profiled. It beats the runner-up (VISENGI's FPGA implementation) by at least 1.45x and the other designs by at least 100x.

KiloCore Version 26 and KiloCore Version 29 throughput per area makes a compelling case for SOC designs to include KiloCore as a digital signal processor on their designs. SOC's have limited area to incorporate a variety of processors, but a design that achieves higher throughput per area is a more compelling candidate to include over a design that costs too much area to justify its improved throughput.

6.2.6 Energy-Delay Product Analysis

VISENGI boasts an impressive energy delay products at approximately 1 (uJ*s)/(MP^2), and KiloCore Version 29 (1.78 GHz @ 0.9V) lags by a factor of 15x. KiloCore Version 29 (1.78 GHz @ 0.9V) does beat Nvidia's design by 12.66x and Intel's design by 261.73x.

Energy-delay product highlights the design that can simultaneously save the most energy while having the highest throughput. KiloCore Version 29's performance in this category is a testimony to its ability to achieve the lowest energy per megapixel encoded of all general-purpose designs and the second highest throughput. Nvidia's solution does have higher throughput but uses significantly more energy per megapixel encoded than KiloCore Version 29 to achieve this. Data centers looking to balance the price of operating a data center solution and throughput of said solution will find KiloCore is a stronger competitor than Nvidia's solution.

6.2.6 Energy Per Megapixel Encoded vs. Area per Throughput Analysis

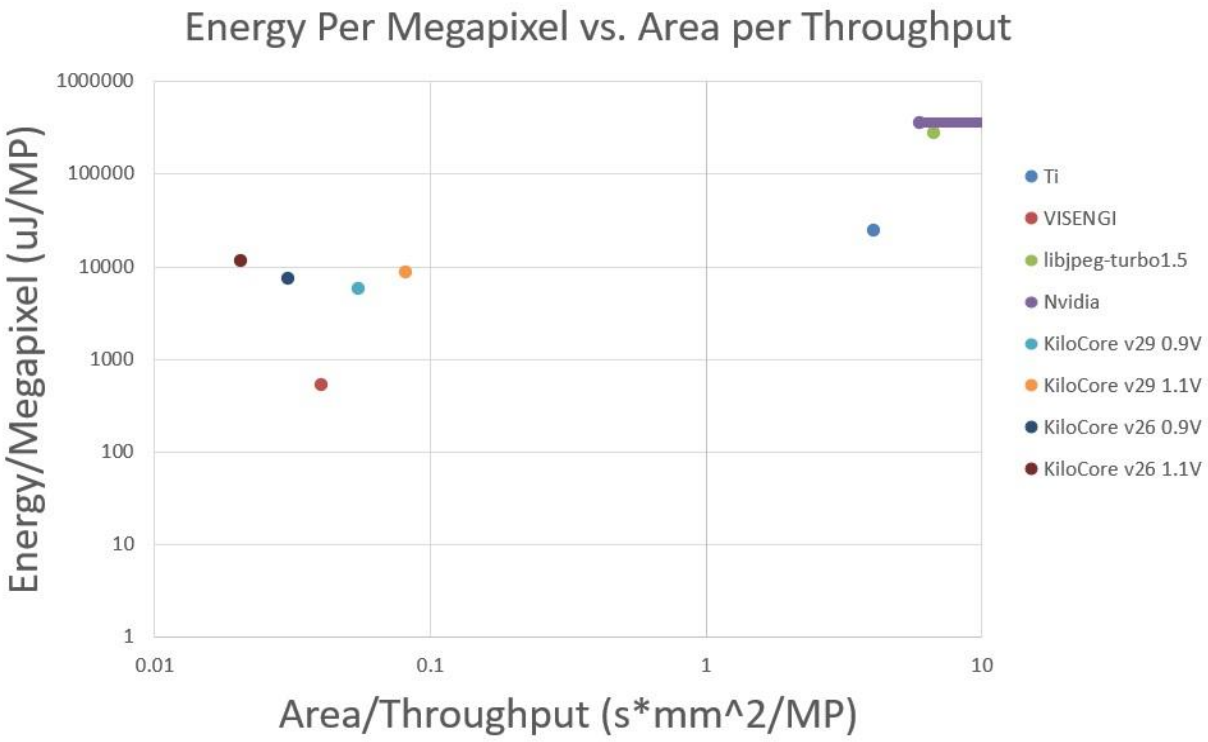


Figure 6.1: Energy per megapixel encoded versus area per throughput analysis, KiloCore implementations and competing vendors (QF = 0.1, all processes scaled to 32nm)

Figure 6.1 shows which designs balance low power usage, high throughput, and low area usage. The data was graphed on a logarithmic scale to allow all vendors to fit on the same graph, and it is more desirable for a design to be closer to the origin. Nvidia's implementation has a line to emphasize that the compression ratio of their metrics is unknown. KiloCore implementations have higher Energy per megapixel encoded than VISENGI's implementation (as discussed in 6.2.4) and KiloCore Version 26 beats every vendor in Area per Throughput, while Version 29 lags behind VISENGI's implementation (as discussed in 6.2.5).

Figure 6.1 shows the KiloCore designs are orders of magnitudes ahead of most industry general-purpose competition, only challenged by an FPGA implementation (VISENGI). Comparing general-purpose algorithms with application specific hardware is generally not done

as application specific hardware has inherent advantages in energy consumption; however, KiloCore implementations sit between general-purpose and application specific designs in energy consumption, making a compelling argument for their usage. It is more economical to invest in general-purpose hardware over application specific hardware on an SOC, but this usually comes at the cost of energy or performance. KiloCore JPEG implementations do reflect this energy cost, but significantly less than other industry competitors making it the most compelling option amount general-purpose hardware.

6.3 Conclusion

KiloCore designs were able to beat out all competition in area and throughput per area. KiloCore designs lagged behind VISENGI's FPGA implementation in energy per megapixel encoded and energy-delay product. KiloCore features inefficient FF memories in its design that could be the reason the FPGA implementation was much more energy efficient. Also, the FPGA implementation is written in a hardware description language, whereas KiloCore's JPEG encoders were written in C++. Hardware description languages require more specificity that could contribute to more efficiency in their design. Finally, KiloCore's 2.29 GHz feature was never implemented and used in the KiloCore JPEG encoders due to the simulation lacking support. With this simulation improvement (and likely a few code changes), the gap between the two designs may have been much closer.

Furthermore, KiloCore designs fail to surpass Nvidia's nvJPEG implementation in throughput, only reaching .34x the scaled performance. However, Nvidia is the only vendor that did not publish their compression ratio, and it could be that Nvidia's nvJPEG numbers are for a larger compression ratio (larger quality factor), in which case KiloCore designs may be much closer to nvJPEG's throughput. KiloCore JPEG encoders' ability to boast competitive

performance numbers while beating every general-purpose implementation in energy usage makes it a desirable alternative to other implementations.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis demonstrates the flexibility of the KiloCore platform by providing 29 working implementations of JPEG encoders. Of these implementations, there is the flexibility to allow for different input data types, core counts, and energy efficiencies.

Chapter 1 explains why the JPEG encoding algorithm suits the KiloCore platform. Chapter 2 discusses the JPEG encoding algorithm step-by-step, summarizing the JPEG standard. Chapter 3 gives background on the KiloCore platform, and the relevant architectural features used in JPEG encoding. Chapter 4 introduces 29 Versions of JPEG Encoders on the KiloCore platform. Chapter 5 details the simulation results of all 29 Versions, comparing each one to the other. Finally, Chapter 6 compares the most competitive versions with competitive industry JPEG encoders to determine how KiloCore implementations fair against various architectures.

7.2 Future Work

7.2.1 C++ and Assembly Discrepancies

Currently, C++ code written for KiloCore is passed through a Clang frontend and a custom KiloCore compiler backend to generate KiloCore assembly. Due to this, multiple architectural features are not adequately taken advantage of in the demonstrated implementations. For example, sophisticated looping with no overhead, address generation, and proper memory management is a hit or miss as the compiler mainly verifies preliminary work.

Furthermore, future designers should write the most competitive KiloCore algorithms in assembly to get even more performance out of the designs.

Unfortunately, the compiler also struggles to minimize the number of instructions per core. KiloCore is limited to 128 instructions per core, and the compiler rarely compiles C++ to abide by this rule, even with the C++ code being perfectly capable of it. For example, the quantization cores are similar; they multiply element by element with 64 incoming elements with a set quantization table in data memory. However, Quantization_Y does an additional check on only the first value to see if there is an end-of-image flag before looping 63 times to handle the rest of the values, whereas Quantization_CbCr directly loops 64 times. This additional check, which is no more than four lines, leads to Quantization_Y having 36 instructions and Quantization_CbCr having 107 instructions. It may make sense if Quantization_Y has more instructions as it does an additional check, but the compiler having almost 3x as many instructions for the simpler code is far from optimal. Furthermore, Quantization_CbCr does not leverage the RPT() instruction that allows for zero overhead looping and instead unrolls the loop partially.

In Version 26 and after, there was an explicit effort to unroll and inline every function possible to increase performance, leading to an increase of about 2x. Consequently, inlining functions affects the instruction count, increasing it by nearly 4x in some cases. While this means that many of the compiled assembly functions cannot be realized in KiloCore, this was a tradeoff necessary to understand better what the KiloCore platform could do without writing the assembly directly. However, writing assembly and properly managing instruction count and memory is the next step for this project.

7.2.2 Additional JPEG Encoding Features

Various JPEG encoding features have yet to be implemented on KiloCore, including color subsampling, progressive JPEGs, RGB-only JPEGs, 12-bit color depth JPEGs, and arithmetic encoding. Additionally, KiloCore could implement on-the-fly quality changes. Currently, quantization tables load in compile time, but KiloCore could recalculate them after detecting the already present end-of-image tag. Combined with an additional signal specifying a quality value, this could make for a very efficient encoding design on various quality levels.

7.2.3 Future KiloCore Improvements

Although KiloCore has a clever 256 x 16-bit memory algorithm, it is sometimes very limiting. It makes it challenging to code Huffman encoding, where a Huffman table is around 160 words. Consequently, it is difficult to fit all the words in the same memory, leading to the need to break up the Huffman table into 16 chunks. If the 256-word memory were truly contiguous, the Huffman encoding would be far more straightforward. Furthermore, the quantization cores could be one core, as there would be plenty of room to fit both quantization tables (64 words each) and intermediates needed in calculations without the compiler throwing an error. Fitting both quantization tables would be possible if a future designer writes an assembly version of the encoders.

Longer FIFOs would help clear up input buffer cores, explicitly reducing the reserve space in the FIFOs to allow 32 words to be written without further delay as the FIFO becomes full. Juggling 24 words to each input FIFO and back creates additional instructions and input buffer cores. Although the original work that introduced KiloCore argues that 32 entries are enough (with eight being in the reserve space), it is relatively myopic to believe that there would not be algorithms that would struggle with the limited space. Especially, JPEG can use FIFOs

that can accommodate 64 words, allowing no input buffers and an entire block to fit in the FIFO without stalling the other pipelines (without buffers). Admittedly, the flexibility of KiloCore in allowing buffers to be programmed and added is arguably a sound solution, but not entirely ideal.

The C++ compiler needs additional work to realize its full potential. Not only does it fail to take advantage of crucial KiloCore features, but it also fails to relay critical error and warning information and instead opts for vague error messages. Although out of the scope of JPEG encoding, it would be advantageous to all future KiloCore development if the C++ compiler was built from the ground up in a low-level language, possibly skipping Clang entirely. The C++ compiler also lacks algorithmically, containing $O(n^2)$ algorithms that create painfully long loading times.

KiloCore has limited bit streaming functionality, making combining variable bit streams especially slow. Adding instructions that allow for managing variable length bitstreams would help with most popular encoding algorithms, as variable length encoding is a powerful data compression tool.

There is no simulation support for the 2.29 GHz operation mode on KiloCore. This speed is only possible when replaced with two-cycle instructions (like multiplies and a specific branching condition). The entropy, compressor, and organizer cores would likely benefit from this functionality as they do not have any multiplies. In addition, improvements in the serial-operation cores would help bring the low-quality factor values closer to the high-quality ones.

7.2.4 JPEG Decoding

JPEG decoding is the next logical step after JPEG encoding on KiloCore. Most of the JPEG decoding algorithm is a relatively simple change from JPEG encoding (DCT, quantization,

and run-length encoding); however, reading from a bit stream bit-by-bit and Huffman decoding is a non-trivial task. Furthermore, decoding has additional features that mimic the encoding features discussed in section 7.2.2.

Bibliography

[1] W3C. "ITU-T Recommendation T.81: Digital Compression and Coding of Continuous-Tone Still Images – Requirements and Guidelines." Available online: https://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[2] W3C. "JPEG File Interchange Format Version 1.02." Available online: https://www.w3.org/Graphics/JPEG/jfif3.pdf.

[3] MathWorks. "Discrete Cosine Transform (DCT)." Available online: https://www.mathworks.com/help/images/discrete-cosine-transform.html.

[4] Huang, Y., Yang, S., & Tsai, P. "The Fast Computation of DCT in JPEG Algorithm." In Proceedings of the European Signal Processing Conference (EUSIPCO), Vol. 2, pp. 1293-1296, 1998. Available online: https://www.eurasip.org/Proceedings/Eusipco/Eusipco1998/sessions/T%20M/TM%20P-7/149/paper568.pdf.

[5] Bohnenstiehl, B. "Design and Programming of KiloCore Processor Arrays." Ph.D. Dissertation, University of California, Davis, 2020. Available online: http://vcl.ece.ucdavis.edu/pubs/theses/2020-1.bbohnenstiehl/Brent_Dissertation_Final.pdf.

[6] "Pillow Documentation: Image File Formats." Available online: https://pillow.readthedocs.io/en/stable/handbook/image-file-formats.html.

[7] Baas, B. "Rounding Techniques." University of California, Davis. Available online: https://www.ece.ucdavis.edu/~bbaas/281/notes/Handout.rounding.pdf.

[8] "About libjpeg-turbo: Performance." Available online: https://libjpeg-turbo.org/About/Performance.

[9] Stillmaker, A., & Baas, B. "Scaling Equations for the Accurate Prediction of CMOS Device Performance from 180 nm to 7 nm." Available online: http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/VLSI-Scaling-Stillmaker.pdf.

[10] R. Damodaran et al., "A 1.25GHz 0.8W C66x DSP Core in 40nm CMOS," 2012 25th International Conference on VLSI Design, Hyderabad, India, 2012, pp. 286-291, doi: 10.1109/VLSID.2012.85.

[11] Texas Instruments. "JPEG Encoder C6678 Datasheet." Available online: https://software-dl.ti.com/dsps/dsps_public_sw/codecs/C6678/JPEG_E/latest/exports/JPEG_BL_Encoder_C6678_Datasheet.pdf.

[12] Visengi. "JPEG Extended Encoder." Available online: https://www.visengi.com/products/jpeg_extended_encoder.

[13] Xilinx. "Zynq-7000 Product Selection Guide." Available online: https://docs.xilinx.com/v/u/en-US/zynq-7000-product-selection-guide.

[14] Intel. "Intel Core i9-9900 Processor." Available online: https://www.intel.com/content/www/us/en/products/sku/191789/intel-core-i99900-processor-16m-cache-up-to-5-00-ghz/specifications.html.

[15] NVIDIA. "Leveraging the Hardware JPEG Decoder and nvJPEG on A100." Available online: https://developer.nvidia.com/blog/leveraging-hardware-jpeg-decoder-and-nvjpeg-on-a100/.

[16] NVIDIA. "NVIDIA A100 Data Sheet." Available online: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf.

[17] NVIDIA. "NVIDIA Ampere Architecture Whitepaper." Available online: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[18] Intel. "Intel Xeon Platinum 8168 Processor." Available online: https://www.intel.com/content/www/us/en/products/sku/120504/intel-xeon-platinum-8168-processor-33m-cache-2-70-ghz/specifications.html.

[19] A. Stillmaker, B. Bohnenstiehl, and B. Baas, "The Design of the KiloCore Chip," in ACM/IEEE Design Automation Conference (DAC), Austin, TX, June 2017.

[20] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A 32 nm 1000-Processor Computational Array," IEEE Journal of Solid-State Circuits (JSSC), vol. 52, no. 4, pp. 891–902, April 2017.

[21] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A Fine-Grained 1000 Processor Array for Task Parallel Applications," IEEE Micro, vol. 37, no. 2, pp. 63-69, March-April 2017.

[22] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A 32 nm 1000-Processor Array," in Proceedings of the IEEE HotChips Symposium on High-Performance Chips (HotChips 2016), Cupertino, CA, August 2016.

[23] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "A 5.8 pJ/Op 115 Billion Ops/sec, to 1.78 Trillion Ops/sec 32 nm 1000-Processor Array," in Symposium on VLSI Circuits, Honolulu, HI, June 2016.

[24] Z. Xiao and B. Baas, "A High-Performance Parallel CAVLC Encoder on a Fine-Grained Many-Core System," 2008 IEEE International Conference on Computer Design, Lake Tahoe, CA, USA, 2008, pp. 248-254, doi: 10.1109/ICCD.2008.4751869.

[25] Z. Xiao, S. Le and B. Baas, "A Fine-Grained Parallel Implementation of a H.264/AVC Encoder on a 167-processor Computational Platform," 2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), Pacific Grove, CA, USA, 2011, pp. 2067-2071, doi: 10.1109/ACSSC.2011.6190391.

[26] Z. Xiao and B. M. Baas, "A 1080p H.264/AVC Baseline Residual Encoder for a Fine-Grained Many-Core System," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 21, no. 7, pp. 890-902, July 2011, doi: 10.1109/TCSVT.2011.2133290.

[27] B. M. Baas, "A Parallel Programmable Energy-Efficient Architecture for Computationally-Intensive DSP Systems," The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003, Pacific Grove, CA, USA, 2003, pp. 2185-2189 Vol.2, doi: 10.1109/ACSSC.2003.1292368.

[28] A. Stillmaker, Z. Xiao, and B. M. Baas, "Toward More Accurate Scaling Estimates of CMOS Circuits from 180 nm to 22 nm," Technical Report ECE-VCL-2011-4 VLSI Computation Lab, ECE Department, University of California, Davis, Dec. 2011.