Title

Computation of the Discrete Cosine Transform on Many-core Platform

Author

Wang, Michael

Publication Date

2024

Peer reviewed|Thesis/dissertation

Computation of the Discrete Cosine Transform on Many-core Platform

By

MICHAEL WANG

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:

Bevan M. Baas, Chair

Venaktesh Akella

Hussain Al-Asaad

Committee in Charge

2024

Abstract

The discrete cosine transform and inverse discrete cosine transform are crucial components in many forms of image processing and multimedia. In particular, the discrete cosine transform is an important step for lossy compression algorithms due to its ability to commonly enable high compression ratios while retaining key information [1]. Real-time image processing extends to all manner of devices, where high throughput, energy-efficient, and area-efficient implementations of the discrete cosine transform are a necessity in smaller devices. This work looks at 20 different implementations of the discrete cosine transform on the novel fine-grain many-core architecture using the KiloCore processor array. The best-performing implementations are considered against the NVIDIA Tesla T4 (GPU), NVIDIA GT-720m (GPU), Altera Stratix III (FPGA), and Intel i5-4200M (CPU). All devices are profiled and evaluated based on throughput, area, power, energy efficiency, area efficiency, and energy-delay product.

As devices are developed in different technologies, data is looked at at each device's original technology and scaled to 32nm. Fine-grain many-core discrete cosine transform implementations showcase the highest throughput per area and lowest area among devices profiled in this work. These designs on the novel platform yield at least 8.35× and up to 5966× higher throughput per area compared to FPGA, CPU, and GPU implementations.

Acknowledgements

I'd like to express deep gratitude to my advisor Bevan M. Baas, who served as my major professor and instructor in VLSI courses. His guidance and expertise have not only greatly assisted the completion of my thesis but have also helped further develop my skills in research and as an electrical engineer.

I'd like to thank both Professor Venkatesh Akella and Hussain Al-Asaad for their support in developing my thesis, from whom I've learned much in their digital systems and computer architecture courses.

I thank all past and present members of the VCL, whose work and insight have helped tremendously in my research. A special thanks to Dr. Brent Bohnenstiehl for his invaluable work in the development of special software tools. A special thanks to Derek Li and Yechengnuo Zhang for whom I had the privilege to work alongside with on many different projects.

Finally, I'd like to especially thank my friends and family who have supported me throughout my time as both an undergraduate and graduate student at UC Davis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The discrete cosine transform (DCT) and inverse discrete cosine transform (IDCT) are transformation techniques that have wide applications in many digital signal processing workloads, particularly in image processing and other forms of digital multimedia. This is due to the discrete cosine transform's property of storing the most important information (low-frequency elements) in only the first few components of a transformation, which can be seen in Figure 1.1, where most of the information is present in the upper left corner of the input. This serves as the backbone for several lossy compression algorithms as this allows less important, high-frequency information to be discarded, alleviating high memory and bandwidth requirements for uncompressed media and lossless compression.

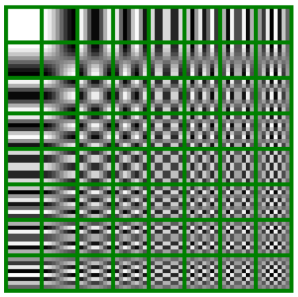


Figure 1.1: Visualization of a 2D DCT on an 8x8 input [2]

With the current state of technology, the only effective means of multimedia communication is to compress the data before storage or transmission [3]. This extends not only to larger desktop computers but also smaller, handheld devices such as phones or watches where the amount of available silicon is much smaller and energy constraints play a much larger role in both silicon and algorithm design. While many architectural and algorithmic solutions have been explored that yield high throughput discrete cosine transforms, implementations found on GPUs and CPUs are wasteful in energy and area. As a result, it is necessary to implement the discrete cosine transform to achieve high performance and maintain good energy and area efficiency.

This thesis explores implementations of the discrete cosine transform on the novel fine-grain many-core platform. Designs explore both single-engine and parallel architectures that yield both energy-efficient and area-efficient discrete cosine transforms while maintaining high throughput.

1.2 Thesis Organization

The remainder of the thesis is organized as follows:

- Chapter 2 gives background to the discrete cosine transform and discusses certain methods for computation.
- Chapter 3 explores and defines a high-level description for a fine-grain many-core platform.
- Chapter 4 discusses general methods of computing the discrete cosine transform on the novel fine-grain many-core architecture.
- Chapter 5 showcases 20 different implementations explored of the discrete cosine transform on the KiloCore fine-grain many-core processor array.

- Chapter 6 presents and compares results in several different metrics between KiloCore implementations
- Chapter 7 presents the most competitive implementations from Chapter 6 and compares them with other devices on the market.
- Chapter 8 gives a summary of this work and provides details on future work regarding the discrete cosine transform.

Chapter 2

Background and Calculation of the Discrete Cosine Transform

2.1 Definition of the Discrete Cosine Transform

The DCT is a mathematical transform that converts signals from a spatial domain to a frequency domain. The opposite is true for the IDCT, which operates in the reverse direction. They are similar to the Fourier transform functionally, but operate only on real numbers being an expression of cosine functions. The transformation for both functions is expressed as a series of coefficients that result from the summation of these cosine functions that operate at different frequencies.

Both transforms are important tools often used in signal processing, and especially in several lossy compression and decompression algorithms. The forward DCT specifically is often used for compression, mainly in different types of media such as audio or images. This is due to the DCT's ability to store most of a signal's relevant information in the first few coefficients that it produces whereas later coefficients can be discarded as they contain less information, making the DCT very powerful for these particular algorithms [1]. The inverse is used in the opposite direction to reconstruct compressed data. While this work specifically explores the IDCT, both the forward and inverse implementations are practically the same

mathematically; thus, results in one case generally closely apply to the other.

The DCT can be applied in multiple dimensions, but for the sake of this work, it is targeted for use in image processing algorithms that only operate on two-dimensional data, such as JPEG decompression. Only implementations of the 1D and 2D DCT are discussed.

2.2 Standard 1D Discrete Cosine Transform

The standard 1D IDCT, or DCT-III, for a vector of length 8 is shown in Equation 2.1 [4]. As cosine coefficients are known a priori, the implementation of a length 8 IDCT requires 16 multiplies and 8 adds, or 24 total operations to fully compute.

$$f(x) = \sum_{u=0}^{7} \alpha(u)C(u)cos[\frac{(2x+1)u\pi}{16}] \tag{2.1}$$

where

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

The standard definition of the 1D DCT (DCT-II) for a vector of length 8 is shown in Equation 2.2 [4]. Aside from the transform now being a function of u, the actual computation of the 1D DCT and IDCT are identical, also naively requiring 16 multiplies and 8 adds, totaling 24 operations to fully compute.

$$f(u) = \sum_{u=0}^{7} \alpha(u)C(u)cos[\frac{(2x+1)u\pi}{16}] \tag{2.2}$$

where

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

2.3 Standard 2D Discrete Cosine Transform

The standard definition of the 2D IDCT (or DCT III) is shown in Equation 2.3. The 2D IDCT is an extension of the 1D IDCT, where as opposed to a single vector, the 2D IDCT is utilized when one performs a 1D IDCT on both the rows and columns of a matrix or vice versa. The use of DCT coefficients and the cosine function remain the same between the standard definitions of both versions. As shown in Equation 2.3 [5], the implementation of the IDCT requires 16448 multiplies and 4096 adds to compute fully (ignoring computation of the cosine).

$$f_{x,y} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7}\alpha(u)\alpha(v)F_{u,v}cos[\frac{(2x+1)u\pi}{16}]cos[\frac{(2y+1)v\pi}{16}] \tag{2.3}$$

where

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

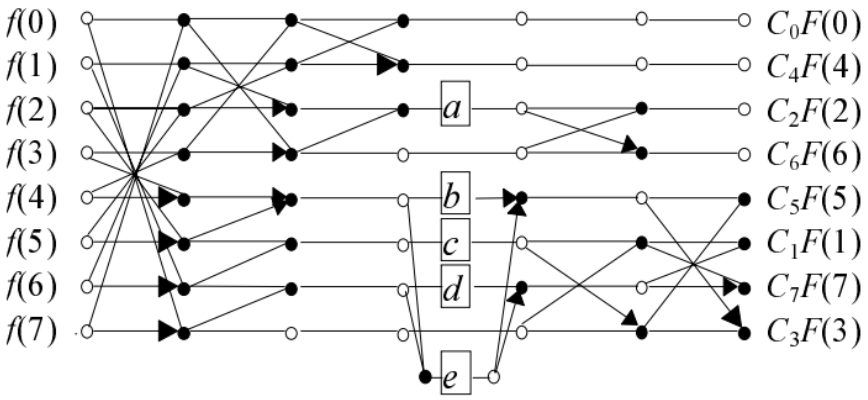
Equation 2.4 [5] shows the series definition of the 2D DCT where there is a difference in the placement of two coefficients. As discussed in later sections, however, the placement of these coefficients should not affect the number of operations required to compute the DCT.

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7}g_{x,y}cos[\frac{(2x+1)u\pi}{16}]cos[\frac{(2y+1)v\pi}{16}] \tag{2.4}$$

where

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

2.4 AA&N Discrete Cosine Transform Algorithm



$$a = c = 0.707106718,\ b = 0.541196100,$$
$$d = 1.306562963,\ e = 0.382683432,$$
$$C_0 = 8,\ C_1 = C_2 = C_3 = C_4 = C_5 = C_6 = C_7 = 16.$$

Figure 2.1: AA&N DCT algorithm [6]

The second DCT algorithm to consider is the Arai, Agui, and Nakajima's (AA&N) Algorithm as shown in Figure 2.1 [6]. The AAN algorithm is an optimization of the standard 2D DCT, where the total numbers of operations are collapsed to only 144 multiplies and 464 additions. Because of the vast reduction in the total number of operations between this implementation and the standard series definition, most compression and decompression algorithms favor this implementation due to its reduced computational intensity. The inverse of the AAN DCT is just the same computation but in reverse, requiring the same number of operations as the forward implementation.

Shaded dots denote an add, with additional arrows specifying a subtract. Boxes represent a multiplication with the constant within the box. The output also has an additional scaling constant. However, in algorithms such as JPEG compression and decompression, this can be combined with the quantization stage.

It should be noted that the full algorithm is a 1D DCT performed on either a single

row or column of an input matrix and must be performed 8 separate times for both rows and columns.

2.5 Row and Column Discrete Cosine Transform Algorithm

The final algorithm that is explored in this work is the explicit Row and Column DCT. Whereas the standard series definition of the 2D DCT combines the operation into a single expression performed on a single matrix, the Row and Column DCT performs the explicit 1D DCT along each row of a matrix, and then again on each column of a matrix (or vice versa), as discussed in a previous section. The computational intensity of this implementation is lowered by reducing the inherent looping required to perform the DCT; where the series definition requires 64x64 loops, the Row and Column DCT requires 64x8 + 64x8, resulting in 2048 multiplies and 1024 adds (when ignoring the cosine computation). Equation 2.1 [4] shows the 1D IDCT for a vector of length 8. Much like the AAN algorithm, this equation only performs the 1D DCT on a single row or column of a matrix, meaning this must be applied 8 separate times for both rows and columns.

While having a large reduction in the number of operations compared to the standard 2D DCT, the explicit Row and Column algorithm are less ideal in general-purpose hardware, whereas the AAN DCT has both fewer operations and less of a strain on memory capacity due to larger look-up tables to store coefficients.

2.6 Fixed-point Representation

All implementations of DCT are done in 16-bit fixed-point notation. The fixed-point data type is simply an integer value that contains an implicit scaling factor denoted by an imaginary "decimal point" that is interpreted by the user and otherwise does not inherently affect computation. The decimal point placement determines the number of integer and fractional bits that are present for a number. This work presents fixed-point notation in an

X.Y format, where X represents the number of integer bits and Y represents the number of fractional bits.

Chapter 3

Fine-Grain Many-Core Processor Array

3.1 Overview

This work explores the implementation of the DCT on the novel fine-grain many-core processor architecture. This section is a brief discussion of high-level characteristics and architectural features of this particular processor style.

3.2 High-Level Architecture

The fine-grain many-core processor architecture is a novel architecture defined as an array of simple processors that are independently clocked and individually programmed [7]. Each processor is clocked utilizing their own clock oscillators, allowing the processor to operate globally asynchronously, and locally synchronously (GALS). This additionally allows idle processors to be independently turned off when they are no longer busy. As processors must each be individually programmed, each core in a fine-grain many-core architecture operates on its instructions.

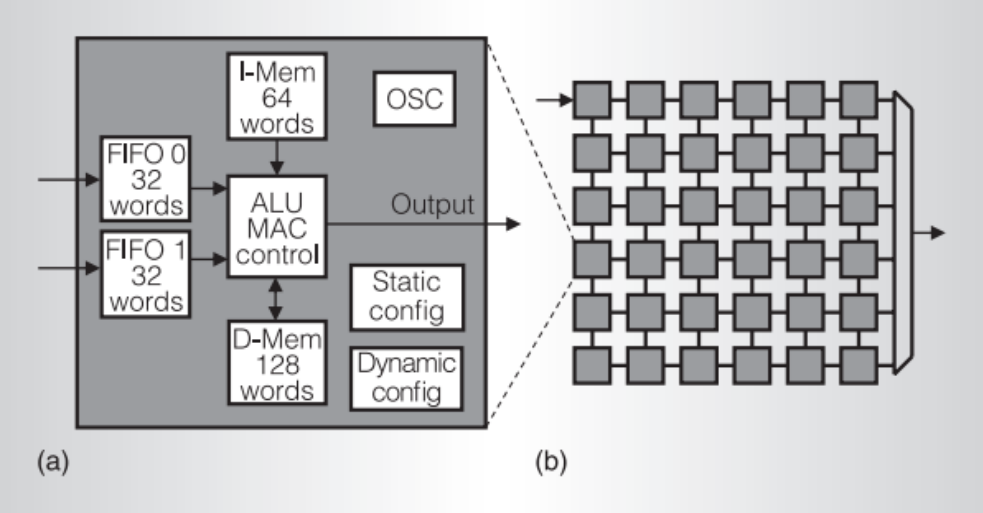


Figure 3.1: AsAP 1 Processor and Layout [7]

Processors are able to transfer data between each other using a 2D mesh network that allows for near and distant inter-processor communication between each other. This allows for a number of serial and parallel architectures to be developed on a fine-grain many-core system.

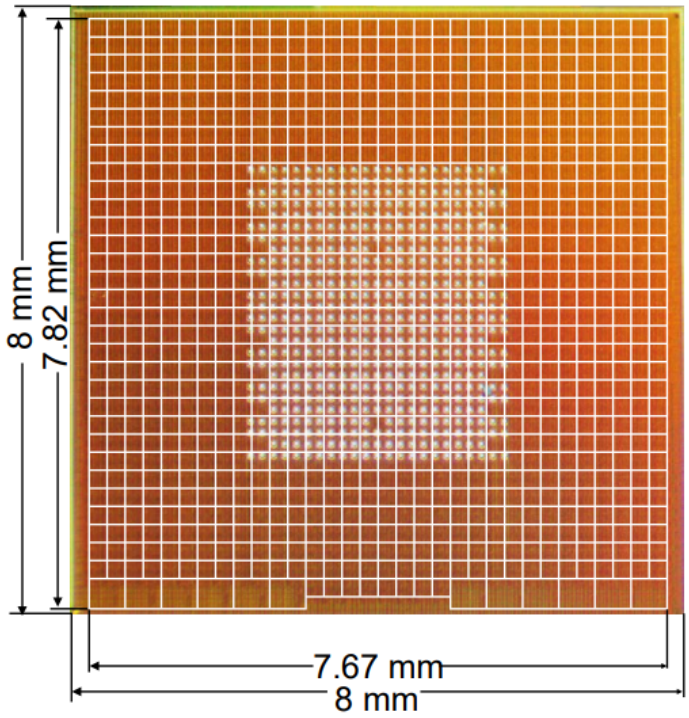
3.3 Memory and Processor Architectures

As the previous section states, processors operate on their own data and instructions. Other parallel architectures, such as GPUs, have larger, centralized memory from which data is pulled. In fine-grain many-core, each processor has its own instruction and data memory. While there may be a larger, distributed memory available in this novel architecture, in many applications, these are seldom used.

Each processor executes RISC-type instructions and are non-algorithmic specific. Although not specialized in any particular workload, this allows for a wide range of different computations and algorithms to be done on this particular architecture; the performance of these workloads can be further boosted as the large number of cores enables high-level parallelization of these workloads, striking a balance between programmability and high

performance.

3.4 KiloCore Processor



Technology	32nm IBM PDSOI CMOS
Num. Procs.	1000
Num. Mems.	12
Die Area	64 mm^2
Array Area	60 mm^2
Transistors	621 Million
C4 Bumps	564 (162 I/O)
Package	676 Pad Flip-Chip BGA

Figure 3.2: KiloCore die photo and specific information [8]

This work explores implementations of the DCT on the KiloCore processor. KiloCore is a fine-grain many-core programmable processor and is the third generation of Asynchronous Array of Simple Processors (AsAP). The KiloCore processor contains exactly 1000 individually programmable processors for efficient MIMD workloads. Processors are single-issue, RISC-type, and operate globally asynchronous, and locally synchronous (GALS) with each processor containing its clock oscillator. Among 1000 programmable processors, the KiloCore processor also includes an FFT accelerator and 12 64KB SRAM modules. The KiloCore processor is built using IBM's 32nm technology node [8].

KiloCore connects processors via a simple two-dimensional mesh network. On-chip

communication operates in two domains, the faster, primary one of the two comprised of a circuit switch network, and the secondary, slower network comprised of a packet router more suited for high fan-out and fan-in. Both protocols end with a size 32 16-bit word dual clock FIFO which helps ensure synchronization between independently clocked processors. Processors can map to both adjacent and distant neighbors.

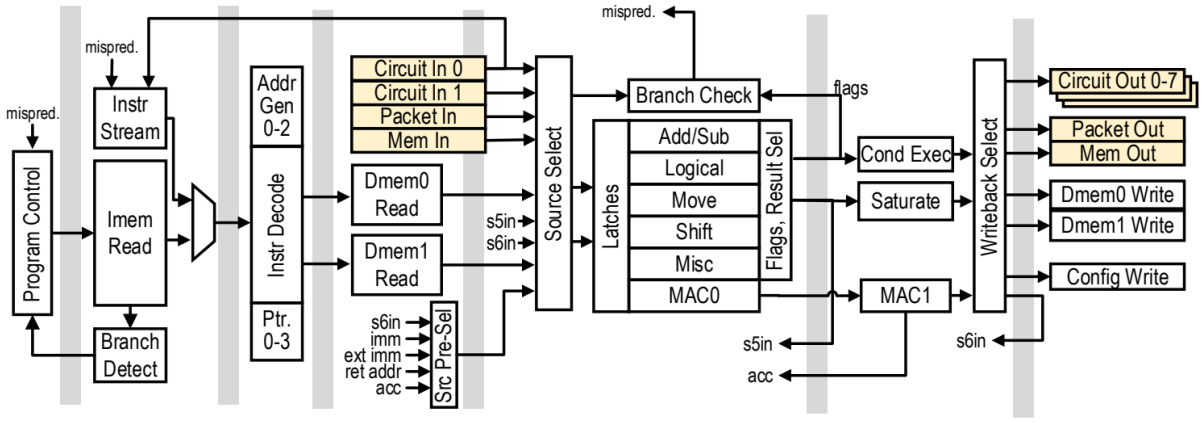


Figure 3.3: KiloCore 7-stage processor pipeline [9]

KiloCore processors contain both 128 40-bit instruction memory and 256 16-bit data memory, both of which are comprised of registers. The data memory itself is split into two segments, both containing 128 16-bit words. To the programmer, the split between memories is generally invisible as both segments are interfaced with one write port and two read ports. The write port is shorted between both memory segments in cases where data of the same array, or the same data itself, would like to be accessed twice.

KiloCore operates on a custom instruction set architecture comprised of 88 different RISC-type instructions which are non-algorithm specific. These instructions are executed on a 7-stage pipeline. Operands for instructions are pulled from three sources; the processor's two input FIFOs, data memory, or in some cases the bypass registers within the pipeline itself if operands have very high temporal locality. Output data can be forwarded to 8 different output ports connected to KiloCore's circuit switch network, 2 on each side of the processor, or through the packet router. The KiloCore pipeline is shown in Figure 3.3.

Chapter 4

Discrete Cosine Transform on Many-Core Architecture

4.1 Overview

This chapter discusses general strategies and methods to compute the 2D DCT on a fine-grain many-core platform. This takes into consideration many of the architectural features discussed in the previous chapter and presents numerous considerations when designing an implementation of the 2D DCT. This includes general strategies that can be applied on to other architectures as well.

4.2 Lookup Table of Coefficients

The first and most generally applicable consideration for implementing the DCT is the handling of the DCT coefficients used to compute the transform. This primarily discusses the coefficients for the standard 1D and 2D DCT algorithms discussed in Chapter 2.

The most straightforward implementation is to generate the cosine coefficients at runtime. This involves computing the term inside of the cosine and then performing the cosine itself. While the most straightforward design, this largely increases the computational complexity required to compute the DCT, requiring 3 multiplies, 1 add, and 1 shift (assuming the denominator is a power of 2) to compute the inside term before performing the cosine

(based on Equations 2.1-2.4). The cosine itself must be done in software without any dedicated hardware. However, a fast cosine can be implemented with the use of a lookup table and linear interpolation.

The second and more favored design is to pre-compute the cosine coefficients and write them into a 2D lookup table. This is possible as the inputs to the cosines are both discrete and known a priori depending on the size of the input matrix. In the case of an 8x8 matrix, the cosine function generate only 64 different coefficients. These 64 coefficients can be directly mapped into a lookup table based on the inputs that produced them. For example, in Equation 2.1, a cosine term with the input $x = 1$ and $u = 3$ would be written to row 1, column three within the lookup table, or vice versa.

1.000	0.9807	0.9239	0.8315	0.7071	0.5555	0.3827	0.1951
1.000	0.8315	0.3827	-0.1951	-0.7071	-0.9807	-0.9239	-0.5555
1.000	0.5555	-0.3827	-0.9807	-0.7071	0.1951	0.9239	0.8315
1.000	0.1951	-0.9239	-0.5555	0.7071	0.8315	-0.3827	-0.9807
1.000	-0.1951	-0.9239	0.5555	0.7071	0.8315	-0.3827	-0.9807
1.000	-0.5555	-0.3827	0.9807	-0.7071	-0.1951	0.9239	-0.8315
1.000	-0.8315	0.3827	0.1951	-0.7071	0.9807	-0.9239	0.5555
1.000	-0.9807	0.9239	-0.8315	0.7071	-0.5555	0.3827	-0.1951

Table 4.1: Discrete Cosine Transform Coefficients (8x8)

The DCT can be further simplified by implementing the α terms into the same lookup table as the cosine values as these are also known a priori. As shown in Equations 2.1-2.4 α can only be one of two values, with it only having any significance when the input term is

0. Additionally, when the input to α is 0, the cosine always evaluates to 1. This means that the only time α has significance in the DCT are instances where a lookup table coefficient is equal to 1. Knowing this, α can be inserted into the lookup table by expressing all 1s inside of the lookup table as $\frac{1}{\sqrt{2}}$.

0.7071	0.9807	0.9239	0.8315	0.7071	0.5555	0.3827	0.1951
0.7071	0.8315	0.3827	-0.1951	-0.7071	-0.9807	-0.9239	-0.5555
0.7071	0.5555	-0.3827	-0.9807	-0.7071	0.1951	0.9239	0.8315
0.7071	0.1951	-0.9239	-0.5555	0.7071	0.8315	-0.3827	-0.9807
0.7071	-0.1951	-0.9239	0.5555	0.7071	0.8315	-0.3827	-0.9807
0.7071	-0.5555	-0.3827	0.9807	-0.7071	-0.1951	0.9239	-0.8315
0.7071	-0.8315	0.3827	0.1951	-0.7071	0.9807	-0.9239	0.5555
0.7071	-0.9807	0.9239	-0.8315	0.7071	-0.5555	0.3827	-0.1951

Table 4.2: Discrete Cosine Transform Coefficients including α (8x8)

The use of the lookup table applies for both the standard 2D DCT algorithm and the Row and Column DCT algorithm.

4.3 Single-Core Discrete Cosine Transform

The first method to implement the 2D DCT is to perform the full computation within a single core. This requires that for a given matrix size, the single DCT core must fully process both dimensions of the matrix before writing to the output of the core.

For implementations that compute the 1D DCT separately for rows and columns, this

method tends to have a strain on memory capacity, requiring intermediate data to be stored between both the 1D row DCT and 1D column DCT. Depending on memory constraints, a single-core implementation on a many-core platform may only support the straightforward 2D DCT.

Data flow into the core can operate in two ways: a full matrix can be read in before computing, or an "on-the-fly" approach can be done where only parts of the matrix (i.e. a full row) is read and processed before reading the next values. Typically, the on-the-fly philosophy is a faster approach; however, depending on upstream processors and available FIFO space between processors, this may incur stalling at the upstream processor. Performance gains with an on-the-fly philosophy can be negated given this fact.

4.4 Multi-core Discrete Cosine Transform

Chapter 2 showed two methods that had the 2D DCT computation broken into two stages, a horizontal DCT stage and a vertical DCT stage. In a fine-grain, many-core platform, this kind of multi-stage implementation of the DCT as DCT can be split between several cores to compute the horizontal and vertical DCT separately. This method is much more favored over the single-core implementation due to the smaller computational complexity per core and additional pipelining benefits as each core can operate on a different matrix at any given time.

The simplest implementation is to break up the DCT into two cores which compute the horizontal and vertical DCT separately. The first core either computes the full row or column and then send the intermediate data downstream to the second DCT core. Data flow considerations are also an important consideration between DCT cores and outside with any additional upstream processor as well. Just like in the single-core DCT, while an on-the-fly philosophy is generally yields faster processing, stalling upstream may occur.

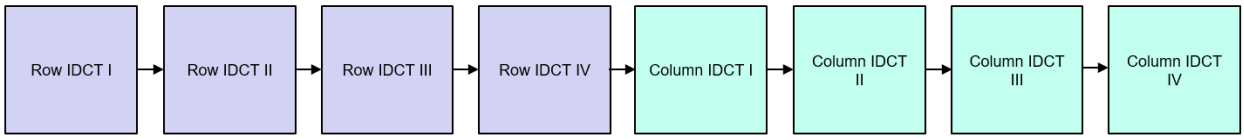


Figure 4.1: Example of a four-stage DCT

The multi-core DCT can be further split into multiple stages for both the row and column DCT. That is, both the row and column DCTs can be split to into smaller stages that can operate on smaller sections of an input matrix. For an 8x8 matrix, the minimum number of elements required to perform the DCT is 8, meaning that both the row and column stages can be split into 8 separate cores which each compute 1 row or 1 column of the matrix. This allows for more simultaneous computation of a full matrix and has greater benefits due to pipelining at the expense of higher communication complexity and area.

4.5 Parallel Architectures

Both single and multi-core DCTs aim to compute a single input matrix at a time. While multi-core DCTs have some pipelining which allows for early processing of new DCTs, this still incurs stalling upstream as it still requires a few cycles in order to process the data before moving to a new matrix.

To alleviate this, parallel instantiations of DCT engines can be utilized to processes multiple input matrices at a given time. Matrices that are processed by the DCT have no dependencies between each other, allowing for multiple DCT engines to operate on different parts of the input data, only requiring a FIFO data flow to be maintained as data is scattered and gathered between these multiple engines.

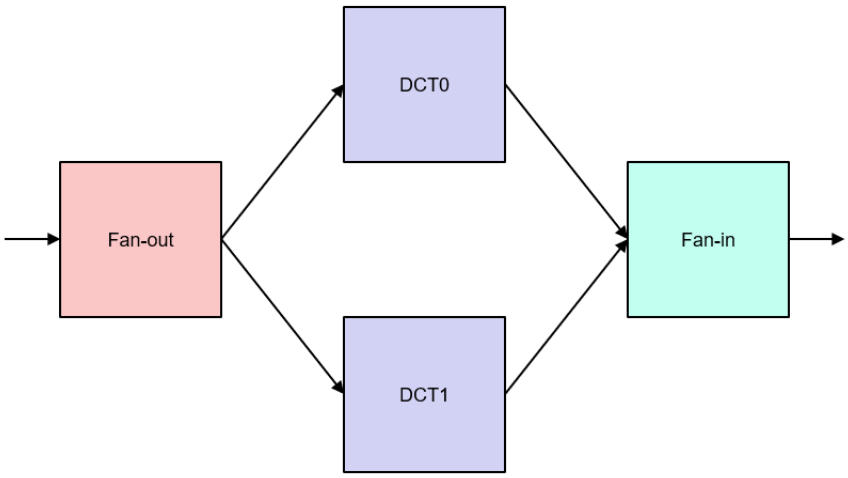


Figure 4.2: Example of a 2-engine DCT

On a many-core platform, the scatter and gather of data can be done by using fan-out and fan-in cores. The number of I/O ports on each core determines the number of fan-out and fan-in cores required to properly distribute and collect data within the whole system. For example, if a particular many-core architecture has 2 input ports and 4 output ports, then for a 4-engine DCT system, at least one fan-out core and 3 fan-in cores would be required.

Similar to the multi-core and single-core DCTs, inter-processor communication also plays a large role in realizing high performance. While on-the-fly computation is typically faster in isolation, depending on the architecture of the many-core platform, stalling upstream negates potential performance gains in a parallel architecture.

Chapter 5

Implementation of the Discrete Cosine Transform on a Many-Core Platform

5.1 Overview

This chapter showcases 20 different DCT implementations on the KiloCore many-core platform. Exploration of the DCT on this novel architecture is broken up into two different sections: single-engine and multi-engine implementations. 16-bit fixed point representation was used for all versions discussed in this chapter. The single-engine implementations explore the different DCT algorithms discussed in previous chapters and other hardware and algorithmic changes to increase overall throughput. The multi-engine implementations have multiple, parallel instantiation of DCT engines. Each section discusses the overall architectural details, changes between different implementations, inherent drawbacks, and bottlenecks.

5.2 Testing Overview

All versions are simulated using the KiloCore Project Manager as discussed in chapter 3. Measurements such as throughput and power are taken from the simulation results of the Project Manager simulation environment. The DCT was tested in an isolated environment,

where payload data was fed directly to the input of the DCT, and the throughput was measured directly at the output of the DCT. The payload data was assumed to be in an 11-bit format and fed into the simulation environment as a text file containing numbers in a 16-bit format. Both random data and 4k gray-scale images were used.

The output of the DCT implementation was written to an output file and was compared directly to SciPy's [10] 32-bit floating point DCT using a Python script to measure the SNR of the KiloCore implementation.

5.3 Single Engine DCT

This subsection focuses on the different single-engine DCT implementations. Changes in the single-engine implementations primarily focus on exploring different DCT algorithms, hardware usage, and code optimizations to achieve higher performance.

5.3.1 Single Engine DCT Version 1.0

This is the first implementation of the DCT on the KiloCore platform that was explored. This version is a direct, naive implementation of the DCT's series definition as discussed in Chapter 2, with the goal of both functionality (rather than performance) and getting baseline numbers for the DCT on a many-core platform.

This direct implementation of the DCT, as shown later on, uses the least amount of cores to implement, being a one-core that computes the full 2D DCT. This implementation operates on one 8x8 data block at any given time, meaning that 64 samples are read in at a time and are all processed together before operating on the next 8x8 block. The cosine values of the DCT are known a priori and are precomputed and stored in the cores DMem in a 5.11 format, creating a size 64 LUT that is accessed based on either x and u or y and v from Equation 2.3. The α coefficient is programmed as a function call, where for any values other than 0, the value equals 1, and for 0, is $\frac{\sqrt{2}}{2}$. To compute the DCT for all 64 values, four nested for loops of size 8 are used, the two innermost reflecting the need for the 64

summations for each element of an 8x8 block, and the two outermost loops representing all 64 elements for each block. A final multiply is done for the scaling constant to compute each element fully.

As a direct implementation of the 2D DCT definition, each 8x8 block requires a total of 4096 loops to fully compute as there are 64 loops per element and 64 elements per 8x8 block. Each multiply done in KiloCore must also be shifted based on the format of the LUT (in this case since it's 5.11, a shift to the right by 11) to maximize precision while maintaining a 16-bit format. Additionally, before each shift, a rounding constant is added to each operation to better maintain accuracy after each multiply. With this in consideration, each loop requires 4 multiplies, 4 shifts, and 5 adds to compute, and thus a full 8x8 block requires 53440 operations. This does not include considerations of other sources of stalling. This is one of three implementations where α affects the overall computation.

5.3.2 Single Engine DCT Version 1.1

Version 1.1 implements a code optimization where at least one of the multiplies is not required to be in the innermost loop and is instead placed in the loop just above it. With this change, the number of operations is reduced by a substantial amount, with the innermost loop only requiring 3 multiplies, 3 shifts, and 4 adds, thus resulting in the total number of operations being reduced to 42688 operations, a 20% reduction from the previous implementation. α is still considered in the total computational complexity in this implementation.

While other optimizations can be explored, it's clear that for direct implementation, the main issue occurs with the number of loops required to fully compute an 8x8 block. Following this version, other DCT algorithms are explored to reduce that number.

5.3.3 Single Engine DCT Version 2.0

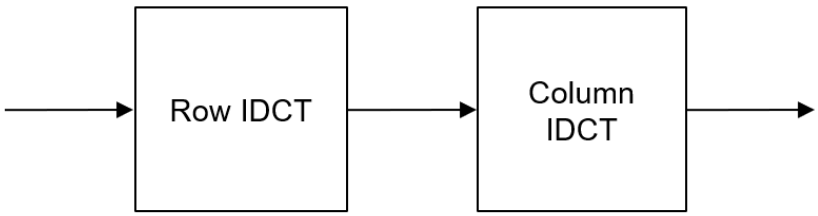


Figure 5.1: Version 2.0 Processor Layout

Version 2.0 directly implements the AAN Algorithm discussed in Chapter 2. The 2D AAN DCT implementation seeks to reduce looping and the number of operations through algorithmic manipulations and breaking the computation into two different stages, a 1D DCT for the rows, and another 1D DCT for the columns.

This version is a 2-core implementation where each core computes on 1D DCT on either the rows or columns of the input matrix. Much like in the standard implementation, each DCT core reads all elements in an 8x8 block and processes the payload entirely before continuing to another 8x8 block. It should be noted that because the DCT is now split between two cores, there is a degree of pipelining that is exploited in the design where after the first DCT core finishes its computation, it can immediately start computing the next payload even before the second DCT core has finished.

As opposed to the 4096 loops required in the direct implementation of the 2D DCT, 1 DCT core in the AAN implementation requires only 8 loops to compute all 64 elements, wherein each loop, a series of operations are performed on either a row or a column of the input matrix. Each loop requires roughly 13 multiplies, 46 adds (including both adds and subtracts), and 13 shifts (for rounding per each multiply). For 8 loops, this results in 552 operations per core, and for two cores is 1104 operations per 8x8 matrix, a 97% reduction in the number of operations than version 1.1.

5.3.4 Single Engine DCT Version 2.0

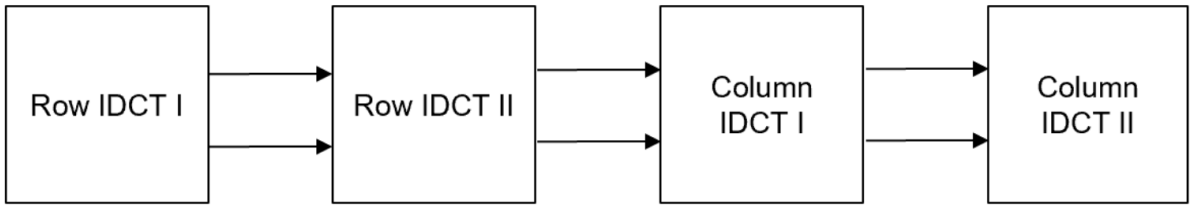


Figure 5.2: Version 2.1 Processor Layout

Version 2.1 is an optimized implementation of the standard AAN algorithm which sees a large increase in performance over version 2.0. This implementation is derived from a JPEG encoder implementation on the same KiloCore processor [11], which utilizes a 4-core engine rather than 2 from the previous version. The row and column DCT cores are further split into two cores, where one core computes the DCT along the first half of an 8x8 block (either row or column) and another core computes the last half. This introduces a better pipelining effect, where cores now operate on smaller portions of data, reducing the time between different reads.

Additionally, this implementation unrolls all loops for each core. While there is zero overhead hardware looping, the KiloCore compiler may not take full advantage of many architectural features for this particular platform. As a result, the use of hardware looping can sometimes be missed in certain designs, where standard looping requires additional overhead: loop counters and checks must be done through software, which incurs extra cycle time in execution. This, however, results in a significant increase in the number of instructions, over 500 per core, which far exceeds the 128 instruction memory.

5.3.5 Row and Column DCT Motivation

In general-purpose hardware, it has been shown that the AAN implementation, and algorithms based on AAN, are favored over the Row and Column DCT due to performance as well as memory and bandwidth requirements not suited to smaller devices [12]. However, we explore the R&C DCT as while the AAN implementation works well to reduce the overall number of operations, it does not exploit well the features present in the KiloCore architecture as the R&C does. Thus, with both algorithmic and hardware optimizations, it may be possible to increase the performance of the R&C DCT such that it outperforms the AAN DCT.

5.3.6 Single Engine DCT Version 3.0

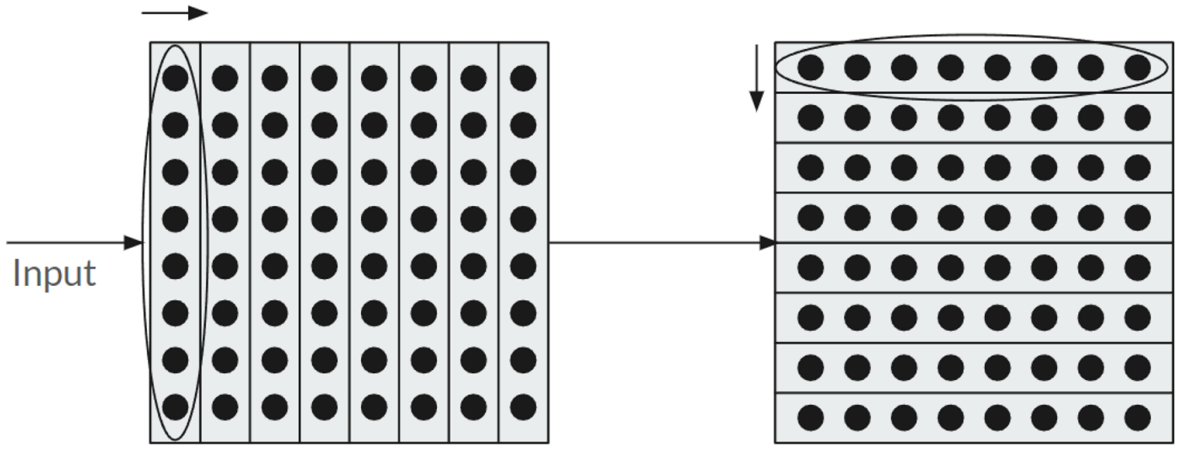


Figure 5.3: Visualization of R&C computation. The first core computes from left to right in column-major. The second core computes from top to bottom in row-major.

The R&C DCT, as the name suggests, splits the computation into two stages of 1D DCTs, much like the AAN algorithm. However, unlike the AAN implementation, the computation follows the standard 1D DCT computation shown in Chapter 2, requiring at least 8 loops to compute each element in an 8x8 block (16 including both stages).

Version 3.0 is a direct implementation of the 1D DCT computed for the rows and columns of an 8x8 matrix. This is a 2-core implementation, with the first core computing the column DCT and the second core computing the row DCT. Much like in the previous versions, the entire 8x8 matrix is read and stored in memory before processing. The next payload is only read after processing is done. This implementation benefits from pipelining as the first core can process new data even before the second DCT core has finished processing its data.

As a direct implementation of the 1D DCT in Equation 2.1, each element requires 8 loops to compute, and for 64 elements, requires a total of 512 loops per DCT core. As in standard 2D DCT, cosine values are known a priori and are stored in LUTs. Each loop has two multiplies, two shifts (for rounding), and three adds, plus an additional multiply, add, and shift for the outside scaling constant. This results in 3776 operations to compute the 1D DCT, and 7552 operations accounting both cores. α is still considered in the total computational complexity in this implementation.

While a considerable improvement in the number of operations compared to the standard 2D DCT (around 82.3% improvement) it falls short of the AAN implementation as previously stated. Additionally, this implementation suffers from additional overhead from using the LUT, which is be explored in a later section. As a result, the throughput generated by this version is significantly slower than the AAN algorithm by a pretty considerable margin.

5.3.7 Single Engine DCT Version 3.1

In version 3.1, the α coefficient was precomputed and stored along with the cosine values as these values are also known a priori. The scaling constant generally done after the innermost loop is also now accounted for and placed inside the LUT of coefficients. With this, the innermost loop is reduced by 3 operations, as well as the final operation per element being completed is eliminated, resulting in only 1 multiply, 2 adds, and 1 shift per loop or

2048 operations per DCT core (4096 for 2 cores).

5.3.8 Single Engine DCT Version 3.2

The KiloCore datapath contains a MAC and 40-bit accumulator that can be used in a separate pipeline for the standard ALU. From version 3.1, the total number of operations was reduced to 1 multiply, 2 adds, and 1 shift, where the first add is for rounding per each multiply and the second add is the summation. In reality, there is only 1 add and 1 multiply required, and thus version 3.2 uses the MAC to reduce the two operations into 1 single operation that can be completed with 1 cycle throughput. Additionally, computations done using the MAC are stored inside the 40-bit accumulator as opposed to a 16-bit variable in memory or bypass registers.

As a result of the MAC, the number of operations is further reduced to just 1 MAC per innermost loop. However, the 40-bit accumulator only allows reads from either bit [39:32], [31:16], and [15:0]. The final result for each computation of an element lands somewhere in between bits [31:16] and [15:0], requiring two reads, two shifts, and an OR operation to extract the correct value which is explored in a later section. As a result, the overall number of operations required for a single core is 832, and 1664 for both cores. Compared to the standard AAN implementation, the gap is now reduced to a 560 operation difference per 8x8 matrix.

It should be noted that implementations with the inclusion of the MAC no longer employ any rounding techniques, resulting in the removal of an add and shift in the innermost loop. Previous rounding techniques involved adding one half (based on the fixed-point notation) to a product and then performing a bit shift (also based on the fixed-point notation) to further reduce the number of operations to perform. This is due to the presence of the 40-bit accumulator; in previous versions, when a 16x16 multiplication is done, only the top 32 bits of the product can be used where rounding attempts to compensate for the lost information. The 40-bit accumulator helps preserve all information across each multiply-accumulate that

is performed until the final result is needed, hence the removal of rounding techniques.

5.3.9 Single Engine DCT Version 3.3

Version 3.3 utilizes the KiloCore address generators for accessing LUT values stored in the DMem. Memory accesses in KiloCore also play a significant role in affecting the overall performance of a design. In previous implementations of the DCT, loops were utilized for two reasons: to repeat certain operations and to index through the LUT of coefficients. While the AAN implementation only has 8 loops per core and very rarely uses the looping value to index through an array, the R&C DCT uses the LUT for each MAC operation that is executed. This means that the pointer of the looping value must first be dereferenced before it can be used to index through the LUT table, requiring extra cycles to retrieve a coefficient before computing a MAC operation.

With the KiloCore address generator, pointers to the LUT can automatically be generated without the need to first dereference the pointer from memory. These pointers are also automatically incremented upon use based on a striding value that is predetermined by the programmer; the R&C DCT benefits greatly as the DCT indexes either by row or column with a set striding amount.

5.3.10 Single Engine DCT Version 3.4

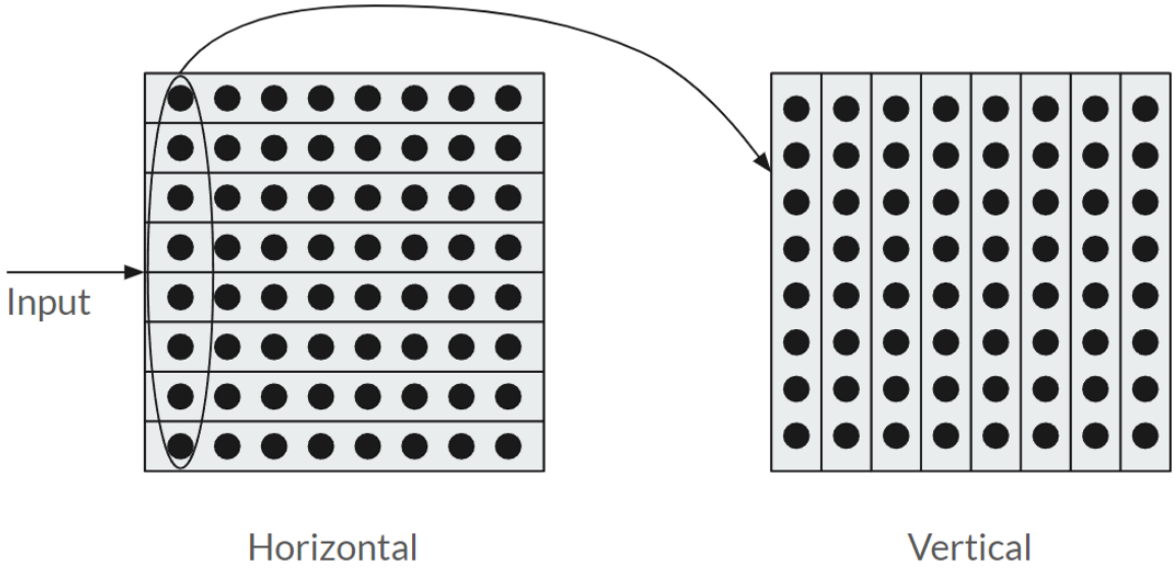


Figure 5.4: Visualization of communication change between DCT cores. The Row DCT is the upstream processor. The first column elements of the Row DCT are sent as soon as they are done computing.

Version 3.4 implements a change in how data is communicated into and between cores. In all previous versions, all 64 elements of an 8x8 matrix were read first before computing the DCT. This would be true for both the upstream and downstream processors. Version 3.4 instead adopts an "on-the-fly" method, where each core only reads enough data from the 8x8 matrix to start computing the DCT, the minimum being 8 elements. Additionally, since data more naturally comes in as row-major, the Row DCT is now switched to be the upstream processor and the Column DCT to be the downstream processor.

After reading the first 8 elements (for the upstream processor, this is the first 8 row elements) the first computation is done to produce the first 8 row elements to be sent to the downstream processor, in which only the first element of this row is immediately sent to column DCT. This is because the column DCT reads data in column-major, and thus requires the first 8 columns before starting its computation. This repeats for the next 7 rows

where the first element of each row is immediately sent to the column DCT, allowing the column DCT to get a head start in computing before all 64 elements are fully computed. Once the row DCT finishes computing all 64 elements, it bursts the rest of the 56 elements in column-major to the downstream processor to complete the 2D DCT.

This change only reflects how DCT cores read and write data; the computation itself remains unchanged from the previous iteration.

5.3.11 Single Engine DCT Version 3.5

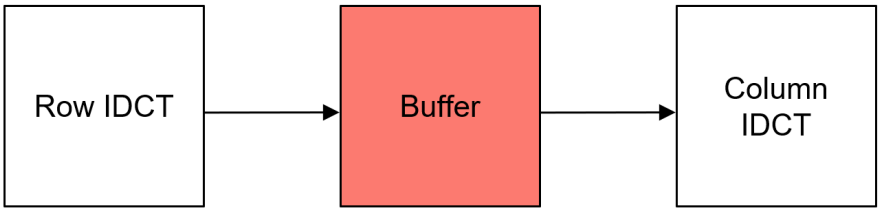


Figure 5.5: Processor layout of version 3.5

Version 3.5 is a three-core implementation that now adds a buffer core between the row and column DCT. Data bursts between the row and column DCT now became the bottleneck as the row DCT would burst 56 elements to the column DCT. Because the FIFO space between each core is only of size 32 words, this meant that the row DCT would stall as a result of the column DCT only reading 8 elements at any given time. This means that at most, the row DCT could write 40 elements to the FIFO before needing to stall for the last 16 elements.

The addition of the buffer core doubles the effective FIFO space between cores as there is now a size 32-word FIFO between the row DCT and buffer core and another size 32-word FIFO between the buffer core and the column DCT, allowing the row DCT to perform a complete burst without stalling. The buffer core is programmed to read first and then immediately write out the value.

5.3.12 Single Engine DCT Version 3.6

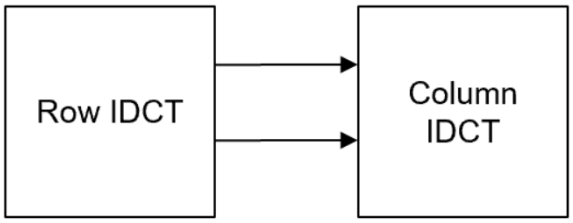


Figure 5.6: Processor layout of version 3.6

While the buffer core alleviates stalling between both DCT cores, the additional hardware cost is undesirable, increasing the core count by 1.5 times. Version 3.6 removes the buffer core and instead utilizes two input ports available on each KiloCore processor, reverting back to a 2-core design. This allows the same effect of doubling the FIFO size without an additional core.

This change is only reflected in the manner in which DCT cores read and write data. Only the column DCT utilizes the two input ports, where the input and output of the 2D DCT are still interfaced through only one port. In this implementation, the row DCT alternates between two output ports, sending 8 elements through one port before switching.

5.3.13 Single Engine DCT Version 3.7

As discussed in version 3.2 when implementing the MAC, the 40-bit accumulator has slower reads when they are made between the high and low 16 bits. As this is the case with all previous implementations using the MAC and accumulator, this required two reads (for the high and low 16 bits), two shifts, and an OR operation to perform the most correct and precise read from the accumulator.

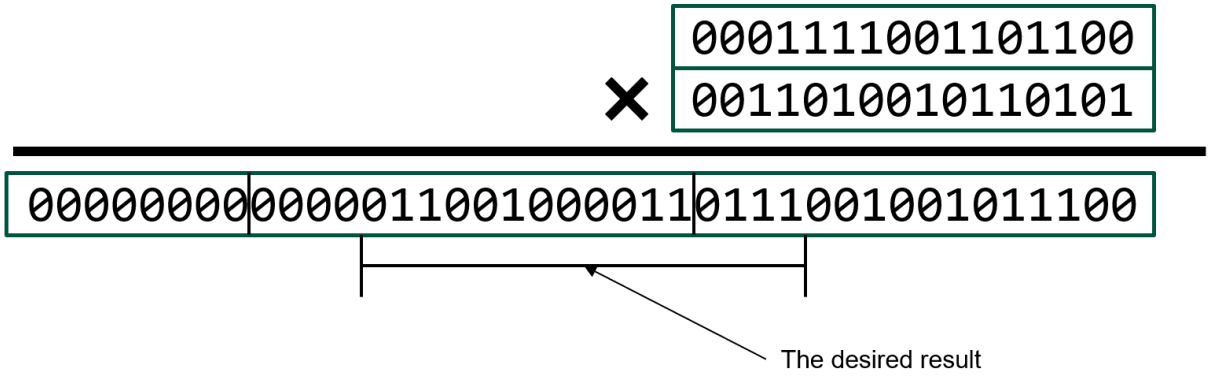


Figure 5.7: Example of a MAC computation

An example can be shown in Figure 4.7. In this example, two 16-bit numbers are multiplied and accumulated by the MAC, where the initial accumulator value is 0. The desired result is in the range of [27:12], which is in between the high and low 16-bits of the accumulator. To get this result, the top 16 bits need to be read and shifted to the left by 4, and the bottom 16 bits need to be read and shifted to the right by 12. The resulting two values then need to be OR'ed to get the desired value from the accumulator.

For the DCT, the need for 4 additional operations is primarily due to the LUT's fixed-point representation, where DCT coefficients are within the range of [-1, 1]. However, the LUT was currently stored in a 5.11 notation. While fixed-point representation is somewhat arbitrary, this resulted in the final value in the accumulator being only partially aligned with bits [31:16], requiring 4 extra operations to do a proper read.

Version 3.7 adjusts the LUT of coefficients to be in a 1.15 fixed-point representation, which now has a range of [-1, 1). While 1 is a possible value in the LUT, the 1.15 format provides a very close approximation to 1 and thus does not need to be hard-coded in the scenario where a look-up of the value one is performed.

5.4 Multi-Engine DCT

The sections explore multi-engine DCT architectures that instantiate multiple single-engine DCT blocks to compute the DCT. The purpose and subsequent changes following the different multi-engine implementations are purely for throughput improvement, with minimal changes in the DCT algorithm. It should be noted that individual engines are taken from single-engine implementations discussed in the previous section.

5.4.1 Multi-engine DCT Version 4.0

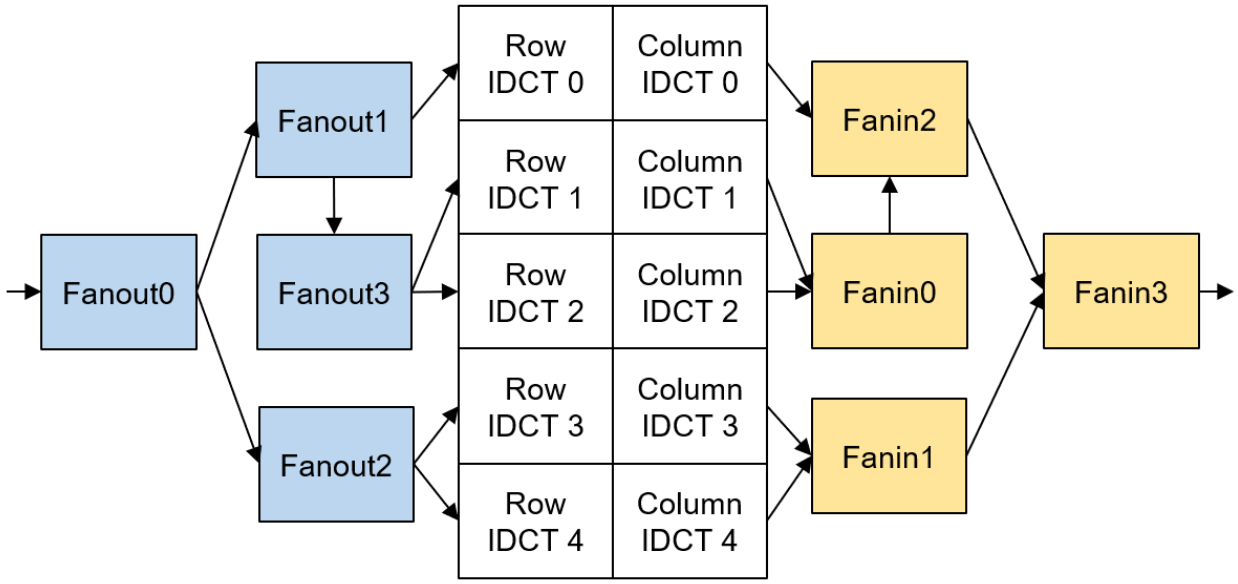


Figure 5.8: Processor Layout of version 4.0

Version 4.0 of the multi-engine DCT is an initial implementation focused more on functionality and baseline performance. This implementation used 5 engines (R&C version 3.6), requiring 10 total DCT cores, 4 fan-out, and 4 fan-in cores for an 18-core implementation. Fan-out cores are required to scatter multiple packets to multiple different engines and fan-in cores are required to fan the data back in. These cores also maintain a FIFO data

flow for the DCT computation.

Figure 4.8 shows the processor layout of how core data is distributed and gathered between the 5 DCT engines. The initial fan-out core (Fan-out 0) takes incoming data and splits it into two different paths. Because there is an odd number of cores, it distributes 3 8x8 matrices to the Fan-out 1 core and 2 8x8 matrices to the Fan-out 2 core. The Fan-out 1 core further splits the data such that after sending an initial matrix to Row DCT 0, it sends the other two matrices to the Fan-out3 core to send to Row DCT 1 and 2 respectively. The Fan-out 2 core sends its two matrices to Row DCT 3 and 4.

Fan-in of data works in the opposite direction. Fan-in 0 takes output data from Column DCT 1 and 2 and Fan-in 1 takes output data from Column DCT 3 and 4. Fan-in 0 sends its data to Fan-in 2 before the data is sent to the final fan-in core while Fan-in 1 sends it directly. Fan-in 3 reads data from Fan-in 1 and Fan-in 2 cores in the correct FIFO orientation.

5.4.2 Multi-engine DCT Version 4.1

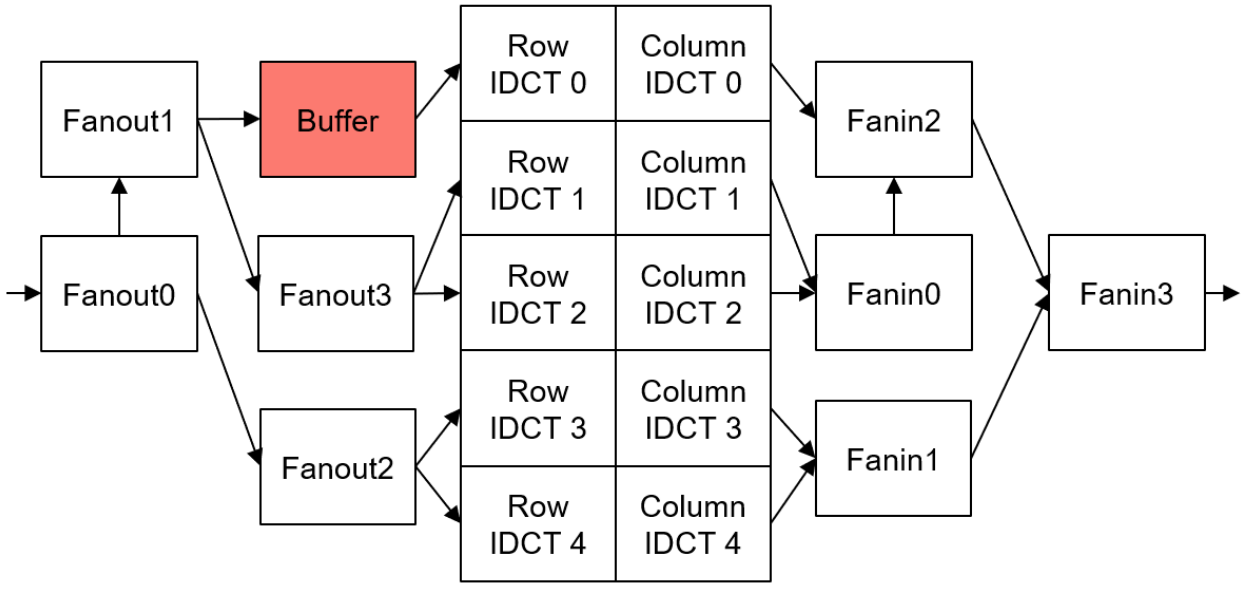


Figure 5.9: Processor Layout of version 4.1

Version 4.1 introduces a buffer core between the Fan-out 1 core and Row DCT 0, becoming a 19-core implementation. The introduction of a buffer plays the same role as it did in version 3.5, where due to large burst lengths between the fan-out cores and the DCT engines, stalling would incur on the upstream fan-in processors. This implementation uses version R&C version 3.6 DCT engine.

Only a single buffer core was used for 1 DCT engine as this version presents itself as a way to illustrate the communication behavior of the full system. As shown in a later section, the theorized performance of a 5-engine DCT was not met with the previous implementation. This occurred for two reasons: data would stall at the Row DCT cores as the buffers would become full before all the data could be sent, causing the upstream fan-out cores to stall at their outputs, and since a FIFO dataflow needed to be preserved, DCT engines 1 and 2 would have to wait for DCT engine 0 to process enough values before starting their computations, which affects the overall system performance.

The addition of the buffer core alleviates stalling between the Fan-out 1 core and the Row DCT 0 core by doubling the effective FIFO size, increasing the throughput to near its theoretical maximum. Data transfers to the first engine became unhindered which allowed processed data to appear at the output earlier while also allowing subsequent engines to start processing earlier.

5.4.3 Multi-engine DCT Version 4.2

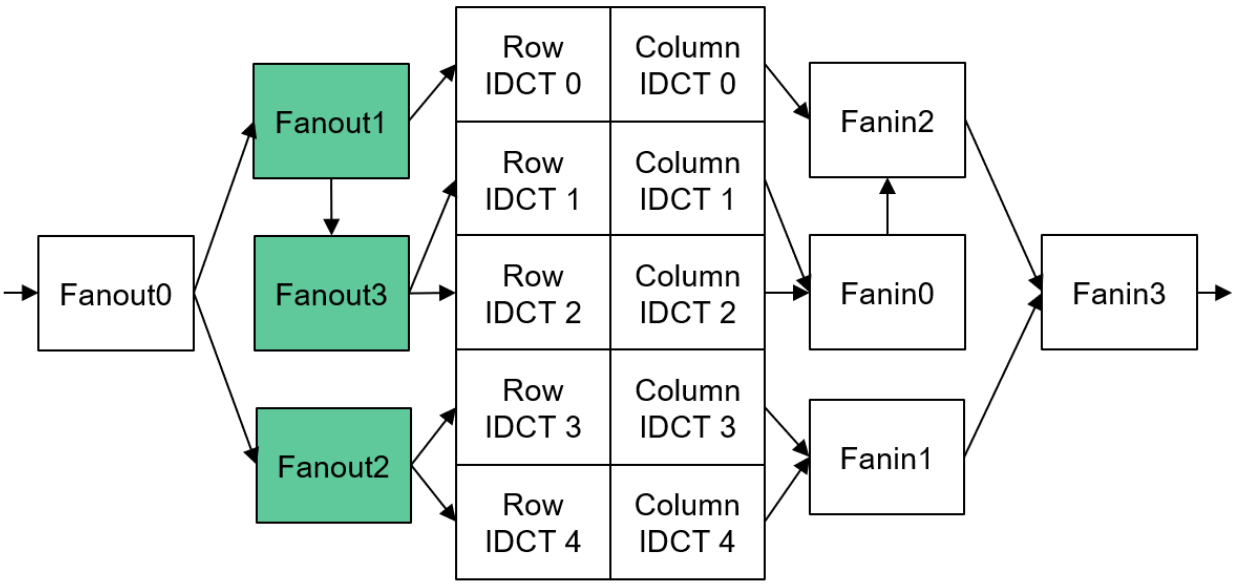


Figure 5.10: Processor Layout of version 4.2

Version 4.2 removes the buffer core, and in place to account for stalling of processors due to large burst lengths instead uses a scheduling approach. In previous implementations, a full 8x8 matrix would be sent before the next 8x8 block was even looked at. This version instead schedules smaller bursts of data to each DCT engine and sends only half of the full payload data at first. The second half of the data is stored in the processor's local memory and is later accessed and sent once all engines receive the first half of their data. This implementation uses version R&C version 3.6 DCT engine.

The Fan-out 0 core is unaffected by this change and still does a full-sized burst to the Fan-out 1 and Fan-out 2 cores. The Fan-out 1 core first bursts 32 elements to the Row DCT 0 core and then bursts the first 32 elements for the Row DCT 1 and 2 cores to the Fan-out 3 core. Fan-out cores 2 and 3 burst the first 32 elements to their corresponding DCT engines. This pattern repeats for the next burst of 32 elements which instead is read from each of the highlighted fan-out cores' local memory in Figure 4.9.

This helps alleviate nearly all stalling between the fan-out cores and DCT engines as the burst size is now manageable relative to the FIFO depth. DCT engines read the first 8 values immediately, meaning that each FIFO only contains 24 elements at any given time.

5.4.4 Multi-engine DCT Version 4.3

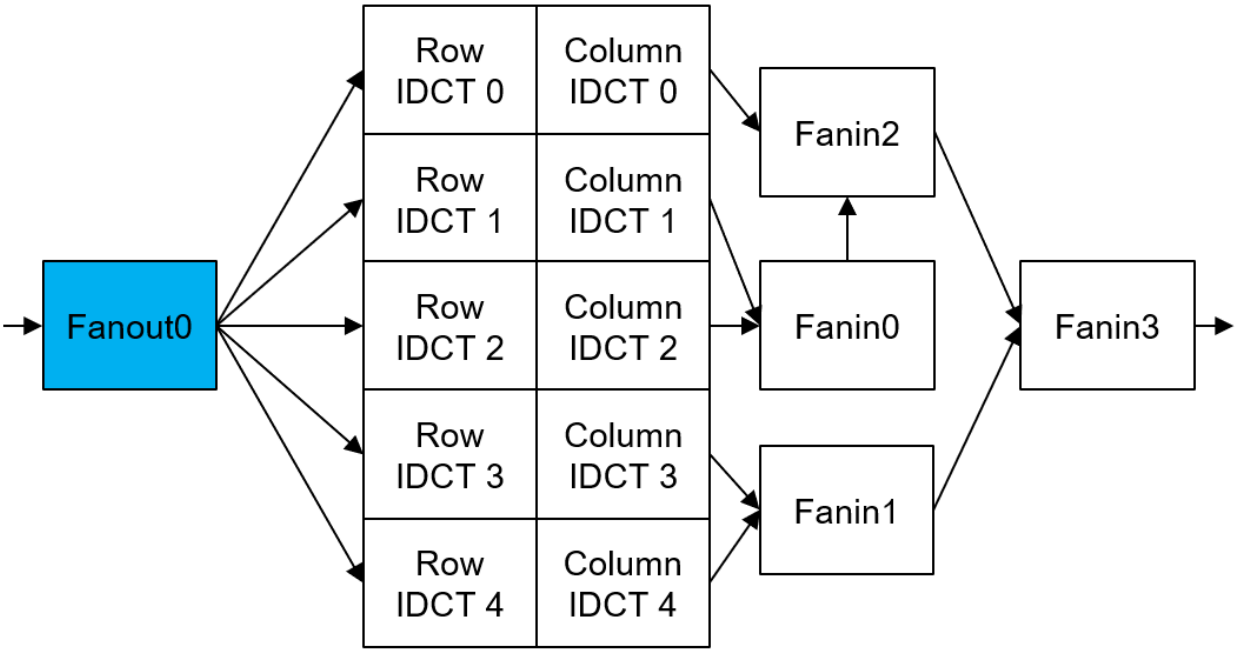


Figure 5.11: Processor Layout of version 4.3

Version 4.3 removes the inner fan-out cores and relies on a single fan-out core to broadcast data to all 5 engines, becoming a 15-core implementation. KiloCore processors each have 8 output ports available, allowing a single core to scatter data to at least 8 different destinations before requiring any additional fan-out cores.

As discussed in previous implementations, full-sized bursts to each DCT engine would cause stalling at the fan-out core, significantly slowing system performance. Instead of using larger bursts, the same philosophy is used in version 4.2, and data transfers are scheduled in smaller 32-word bursts instead of full 64-word bursts. Other scheduling algorithms were

tested, however, of them, a 32-word burst performed the best as it is the most fair algorithm, whereas other algorithms with heavier biases towards certain cores or uneven burst stages would incur larger penalties at a later time.

This version, although faster than version 4.0, also falls short of the theoretical maximum throughput. Although scheduling is employed, the benefit is not as great as only a single fan-out core is responsible for broadcasting input data to all 5-engines. Version 4.2 had multiple stages of fan-out before reaching the DCT engines, meaning that multiple fan-out cores would only be responsible for a few DCT engines. Additionally, because there are multiple stages, it benefited from some amount of pipelining.

5.4.5 Multi-engine DCT Version 4.4

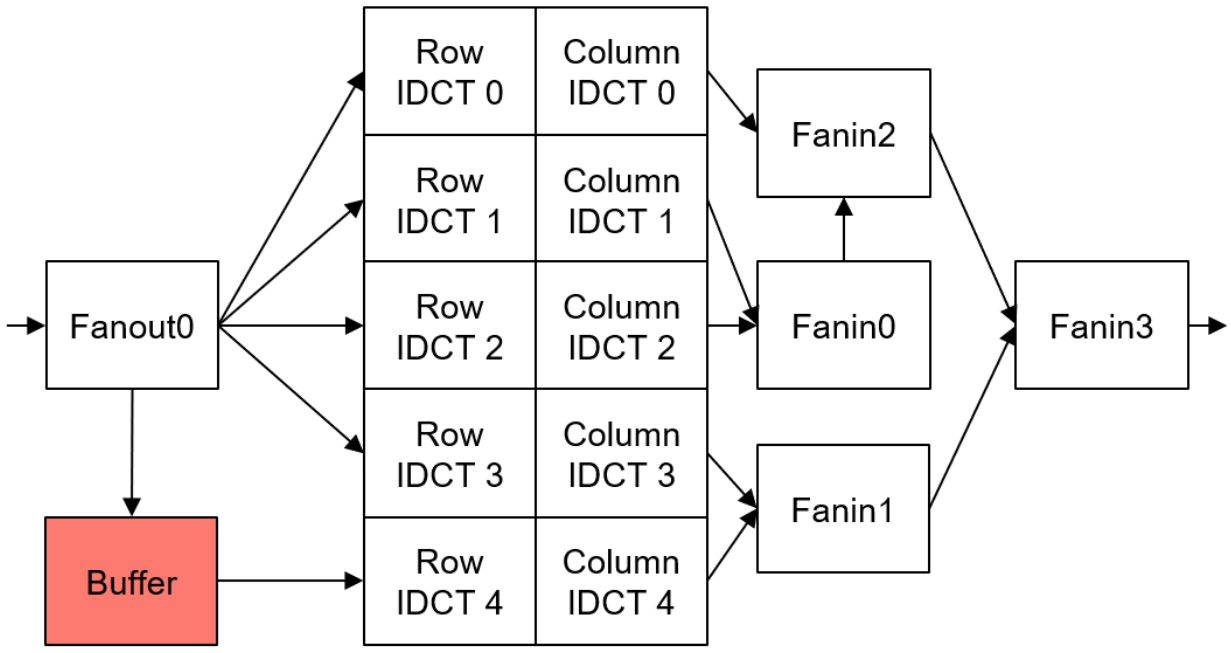


Figure 5.12: Processor Layout of version 4.4

Version 4.4 places a buffer core between the Fan-out core and the bottom DCT engine, resulting in a 16-core implementation. The implementations helped illustrate that the reason

for the loss in performance is the amount of time it takes to fully complete one loop through all the DCT engines. As the Fan-out core is solely responsible for all 5 engines, the time it takes to loop back to perform the second burst of 32 words was longer than the time it took for the first DCT engine to process its first burst, causing all DCT engines to sleep while waiting for additional data. This implementation uses version R&C version 3.6 DCT engine.

The addition of the buffer core allowed for a 64-word burst just for the last DCT engine. Because the last DCT engine can receive all its data in a single burst, a quicker loop back is possible to the first DCT engine is possible, showing a slight increase in performance from version 3.4.

5.4.6 Multi-engine DCT Version 4.5

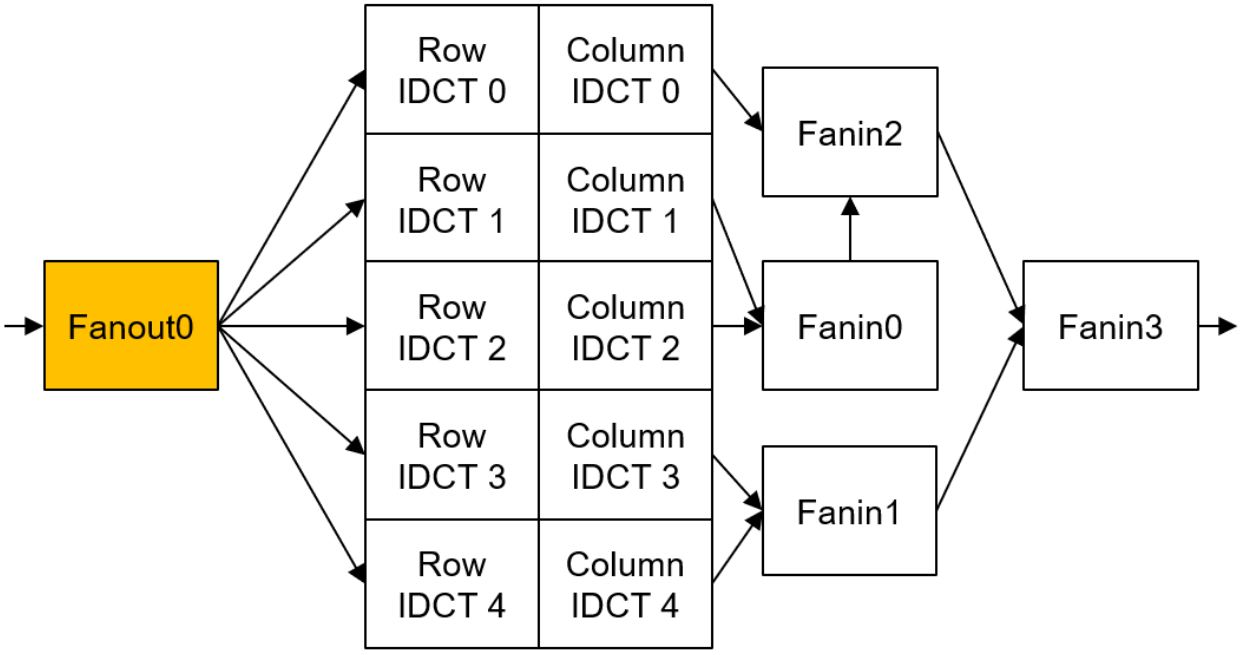


Figure 5.13: Processor Layout of version 4.5

Version 4.5 removes the buffer core and no longer schedules smaller bursts to each DCT engine, reverting to a 15-core design. The Fan-out core now makes full-size bursts to each DCT engine. To facilitate this without incurring stalls, the Row DCT cores also

reverted to storing a full 8x8 matrix in local memory before computing the DCT. This allows the Row DCT cores to use their local memory as a software FIFO, allowing for full-sized bursts to be unimpeded by the hardware FIFO's smaller depth of 32 words. This change only reflects the Row DCT cores and only the method in which they read input data. The DCT computation itself and the communication to the downstream Column DCT cores are left unchanged. This implementation uses version R&C version 3.6 DCT engine.

5.4.7 Multi-engine DCT Version 4.6

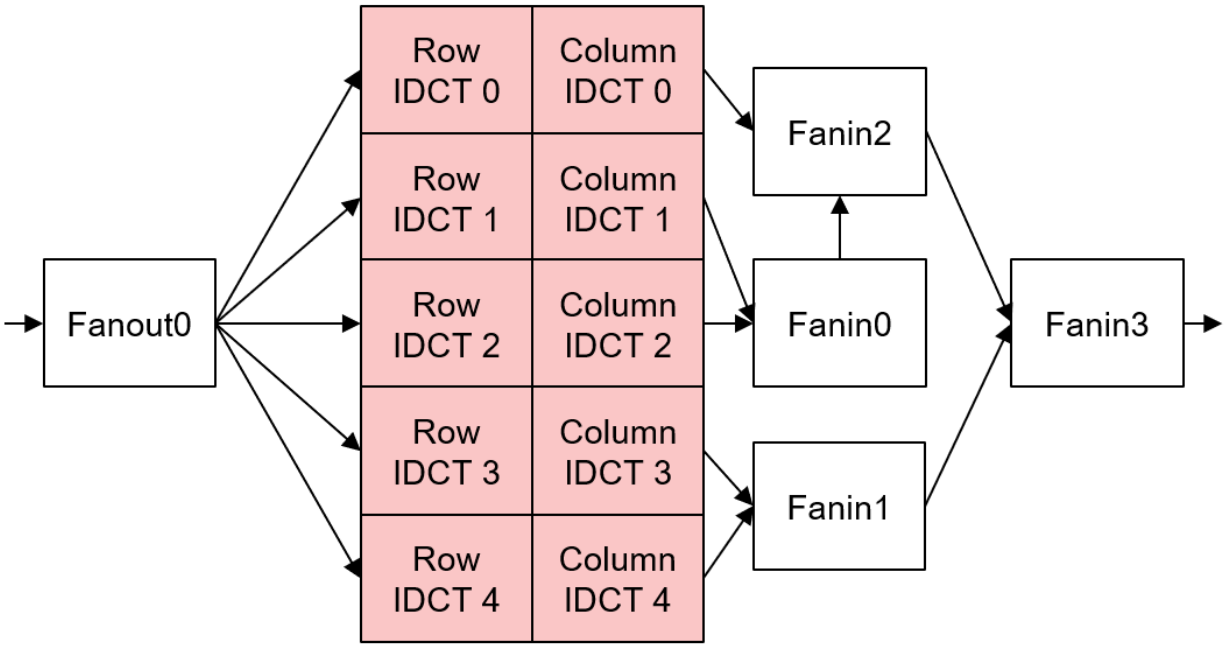


Figure 5.14: Processor Layout of version 4.6

Version 4.6 adopts the latest single-engine DCT implementation, version 3.7, which uses adjusted coefficients to extract values from the cores' accumulator more efficiently. Multi-engine explorations were done before version 3.7 was realized.

5.4.8 Multi-engine DCT Version 4.7

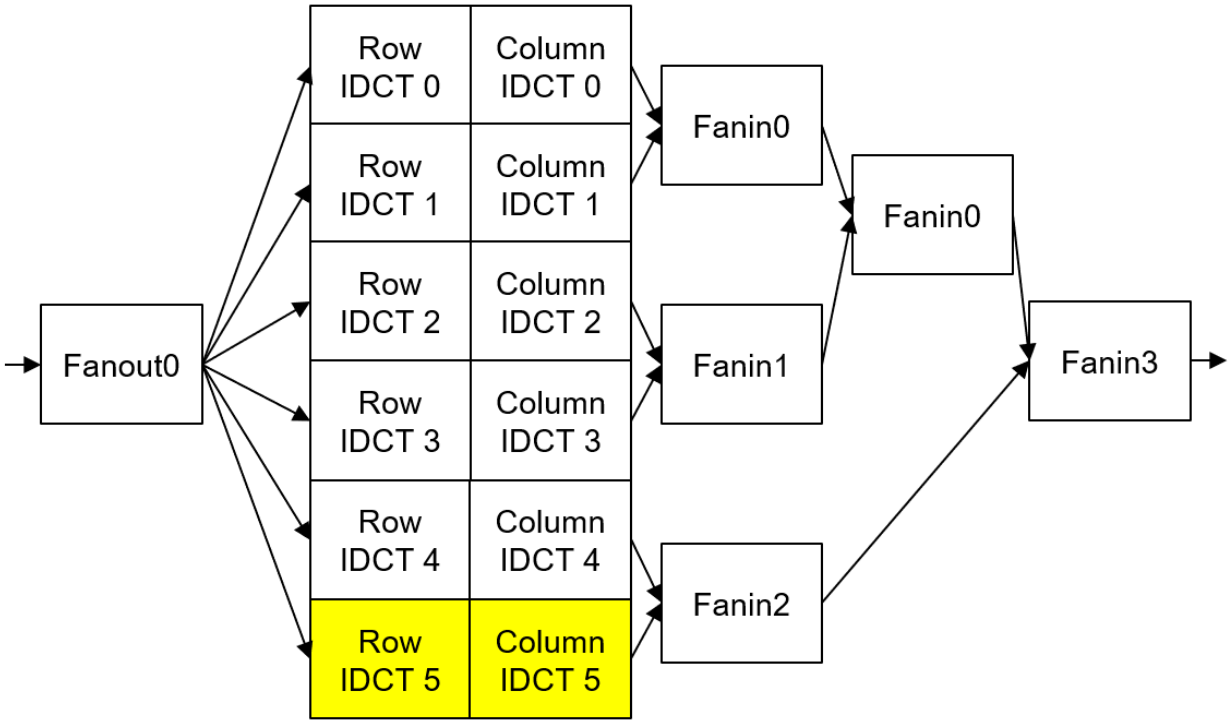


Figure 5.15: Processor Layout of version 4.7

Version 4.7 scales the previous implementation from 5 engines to 6 engines. An additional Fan-in core is required in order, requiring an additional 3 cores to implement, becoming an 18-core design.

Chapter 6

Simulation Results of the Discrete Cosine Transform

6.1 Overview

This section discusses the simulation results of all DCT implementations on the KiloCore platform, including throughput, power efficiency, area efficiency, energy, and accuracy. Tests were done with a custom JPEG decoder and with the DCT in isolation, where tests with the decoder had data taken just before the DCT. An energy-efficient and high-throughput

While the quality of the images and the speed of the JPEG decoder do affect the overall system performance where the DCT may not be a bottleneck, all results discussed in this section ignore the external effects of the JPEG decoder and analyze results of the DCT in isolation.

Throughput was calculated with equation 5.1:

$$Throughput = \frac{(\#\ of\ Pixel\ Components) * 10^{-6}}{Execution\ Time} \tag{6.1}$$

SNR was calculated with equation 5.2:

$$SNR = 10 * \log_{10} \frac{\sum_{n=0}^{N} |(KiloCore\_Data[n])^2|}{\sum_{n=0}^{N} |(KiloCore\_Data[n] - Gold\_Ref[n])^2|} \tag{6.2}$$

Area was calculated with equation 5.3:

$$Area = (\#\ of\ cores) * 0.05545 \frac{mm^2}{core} \tag{6.3}$$

Throughput per area was calculated with equation 5.4:

$$Throughput\ per\ Area = \frac{Throughput}{Area} \tag{6.4}$$

Energy per word was calculated with equation 5.5:

$$Energy\ per\ Word = \frac{Power}{Throuhgput} \tag{6.5}$$

Area Normalized energy-delay product (EDP) was calculated with equation 5.6:

$$Area\ Normalized\ EDP = \frac{Energy\ per\ Word}{Throughput\ per\ Area} * 1000 \tag{6.6}$$

Results are reported in 1.2 and 2.29 GHz, where designs operating at 1.2 GHz have better energy efficiency while designs at 2.29 GHz display better area efficiency. Each core can operate safely at 1.2 at 0.9V. However, so long as certain ALU carry and zero flag instructions are avoided (which are not utilized in all implementations), KiloCore can operate at 2.29 GHz at 1.1V [13]. An increase in the clock speed for each core increases the throughput proportionally, which is explored in later sections.

6.2 Throughput of KiloCore Implementations

This section showcases throughput changes across different versions of the 2D DCT. Results in this section are testing with the DCT isolated, where the input is fed directly into the DCT and the output is taken as soon as the DCT is done computing. While computing the transform by itself is not influenced by the input, the overall speed of a whole compression or decompression algorithm depends on other parts, such as the Huffman decoder or quantization in JPEG decoding, where the quality of an image can change the

entire system performance considerably.

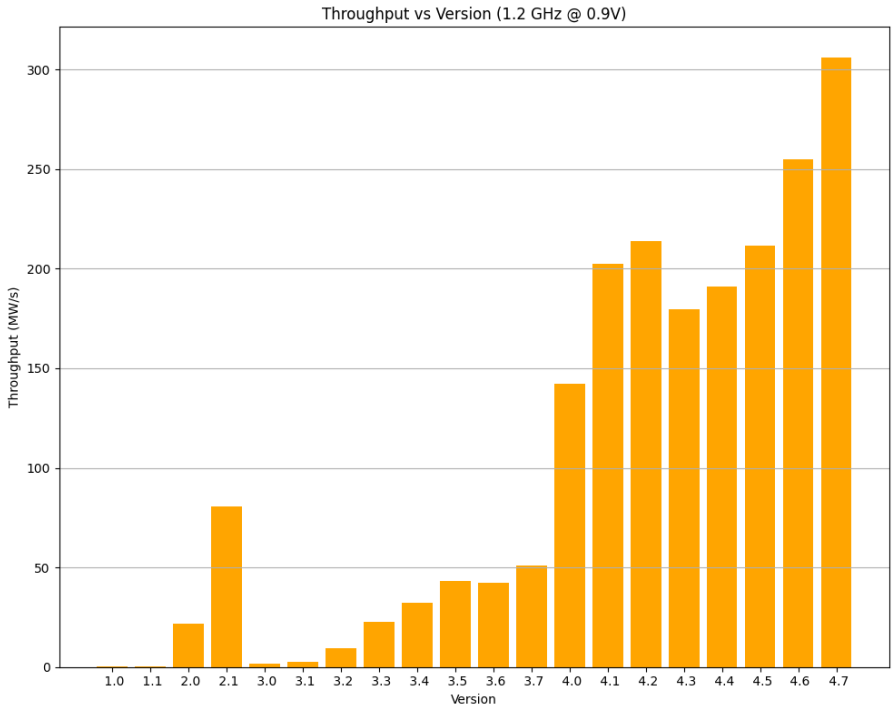


Figure 6.1: Throughput results @ 1.2 GHz

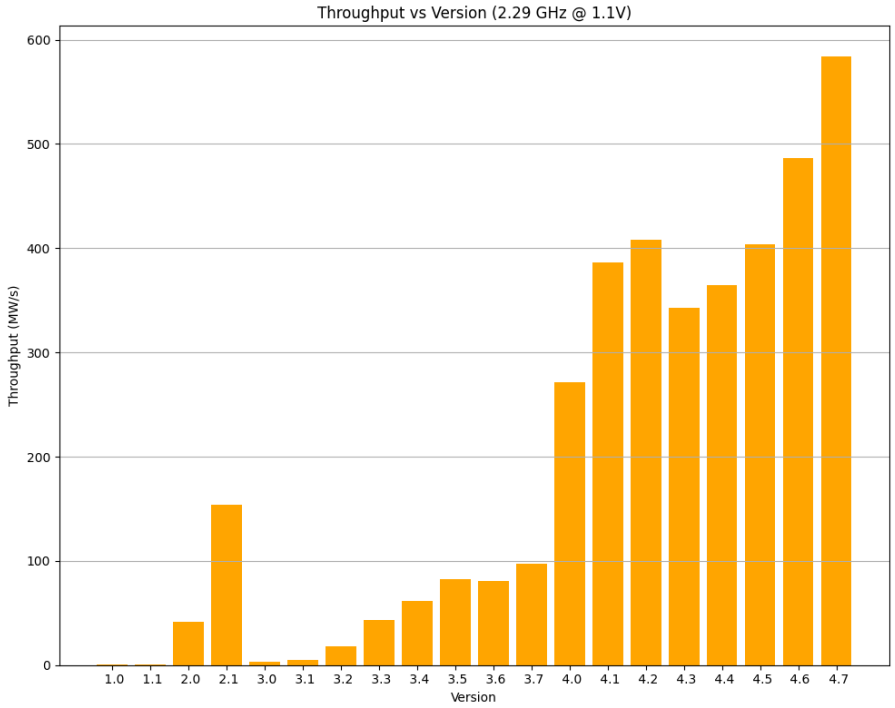


Figure 6.2: Throughput results @ 2.29 GHz

Figures 6.1 and 6.2 present throughput results at 1.2 and 2.29 GHz respectively. Performance generally increases with each version, with much larger spikes in throughput appearing at version 4.0 when multi-engine designs are under consideration. An exception to this is with version 2.1 which corresponds to an early exploration of the AAN DCT. This implementation is based on [11] and is a 4-core implementation and expansion of the standard AAN DCT, or version 3. This increase in throughput is due to a combination of large amounts of loop unrolling and splitting the row and column DCTs into 2 different cores, where each core calculates only half of either the row or columns of the DCT. While there is a significant increase in throughput relative to other single-engine designs, this implementation has an additional area cost associated (which is discussed in a later section). Additionally, when compiled into assembly, each core contains over 500 instructions, which far exceeds the number that can be held in a single core due to all loops being completely unrolled. Version 4.6 and 4.7 are the only two multi-engine designs that utilize the latest single-engine DCT,

with version 4.7 at 2.29 GHz possessing the highest throughput.

6.3 SNR of KiloCore Implementations

This section focuses on precision between the different implementations of the DCT. While more bits are better, JPEG requires at least 8 "good" bits for the decoder to be considered to have marginal losses with the original image [14]. This corresponds to roughly 50 dB, where results that are above this threshold can be considered identical in the context of image compression. Each DCT design was tested against SciPy's DCT in Python.

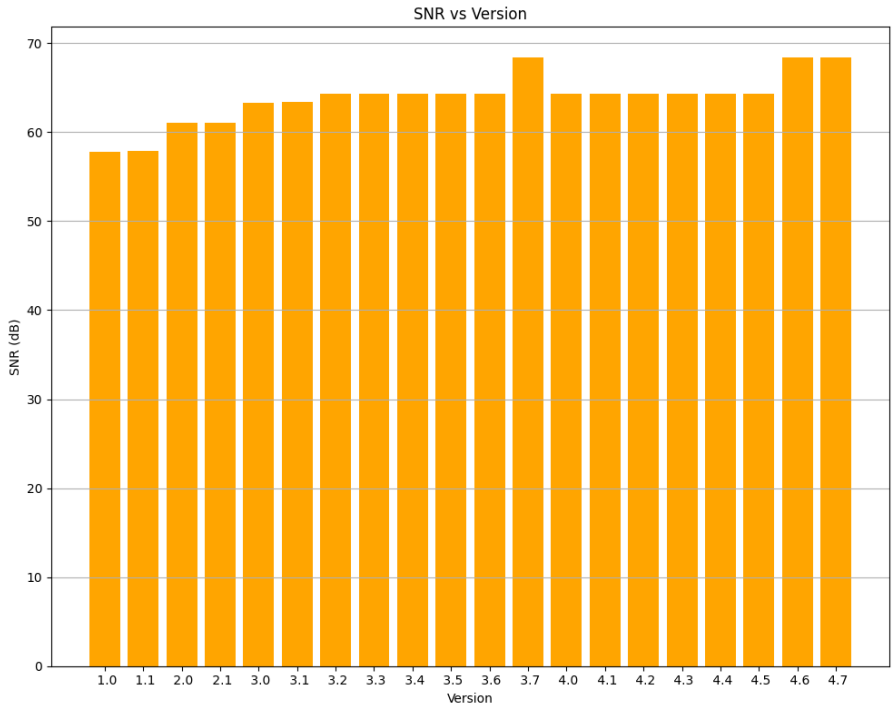


Figure 6.3: SNR results vs Python Golden Reference

Figure 6.3 presents the SNR among all 20 different DCT implementations. Each implementation produces an SNR well above 50 dB, with newer versions generally increasing in precision. Versions 3.2-3.7 had all rounding techniques removed. This was done for two reasons: to negate additional overhead due to rounding and to take advantage of the 40-bit accumulator. Versions that do not use the accumulator must be truncated to maintain 16

bits between operations; however, MAC results are stored in the accumulator, preserving all information between operations. This only requires one truncation at the end, where the final 16 bits are extracted from the MAC, making rounding less necessary. Figures 6.4 and 6.5 show accuracy results between implementations 3.0 and 3.7 compared to a SciPy golden reference. While there is much more bias for version 3.7's results, the magnitude of the error is less than in version 3.0.

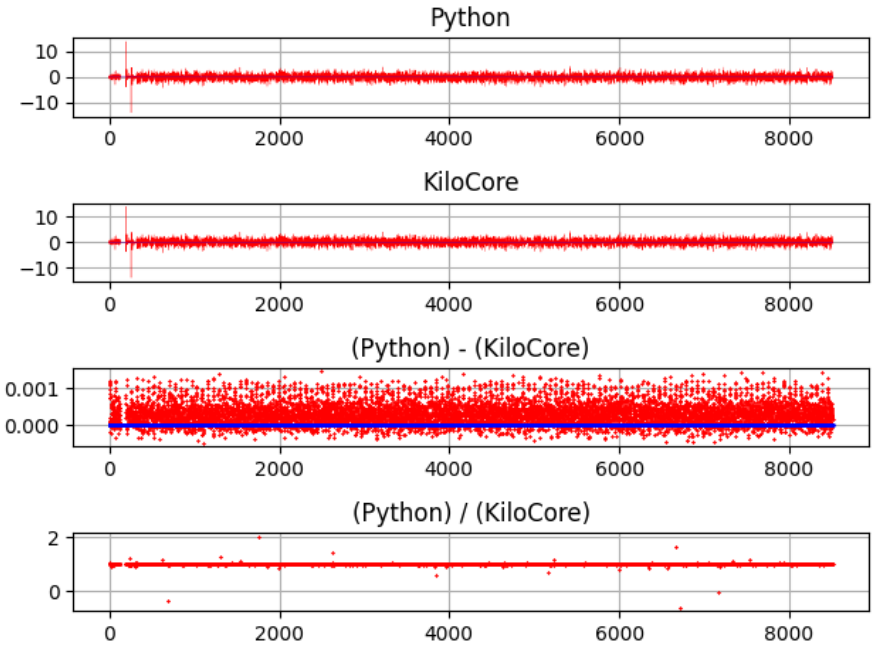


Figure 6.4: Diff results for version 3.7

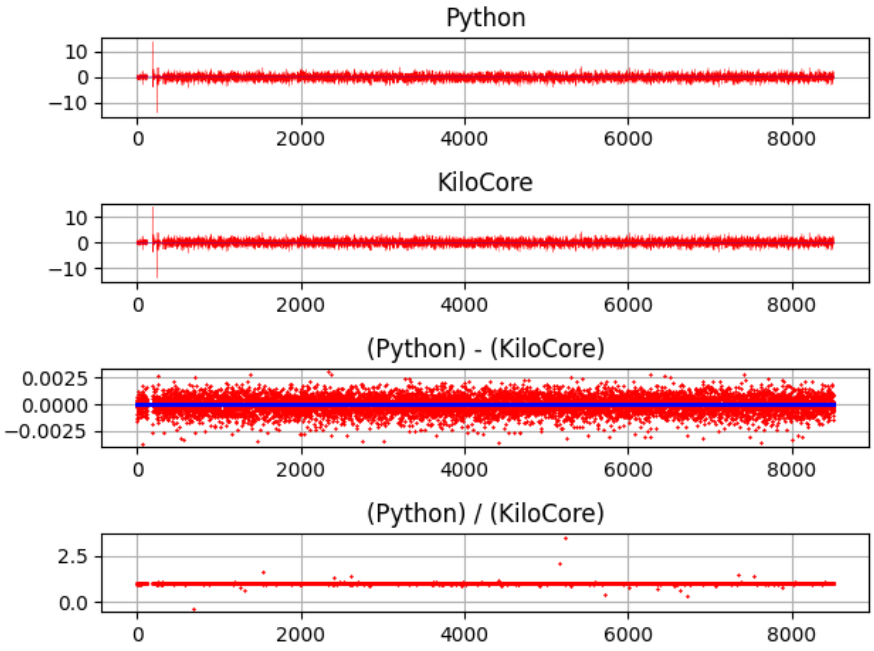


Figure 6.5: Diff results for version 3.0

Versions 1.0-1.1 have notably less precision than all other versions, being the only two below 60 dB. This is primarily due to the number of operations required to fully compute the naive DCT, where a single element requires 64 iterations. A larger number of operations results in greater error, as each operation requires truncation down to 16 bits, where the effect of rounding is diluted.

Versions 3.2-3.6 and 4.1-4.5 all share the same SNR as these implementations all utilize the MAC, where computation using the MAC is completely identical across all implementations. Version 3.7 introduces an adjustment to the fixed-point notation of coefficients, also reflected in versions 4.6 and 4.7, resulting in a slight increase in SNR for these three implementations.

6.4 Area of KiloCore Implementations

This section briefly discusses the area taken by each implementation. As shown in Figure 5.7, the area is split between two different sections, the single-engine and multi-engine

implementations. The majority of single-engine designs are two-core implementations, except version 2.1 and 3.5 which are four and three-core designs respectively.

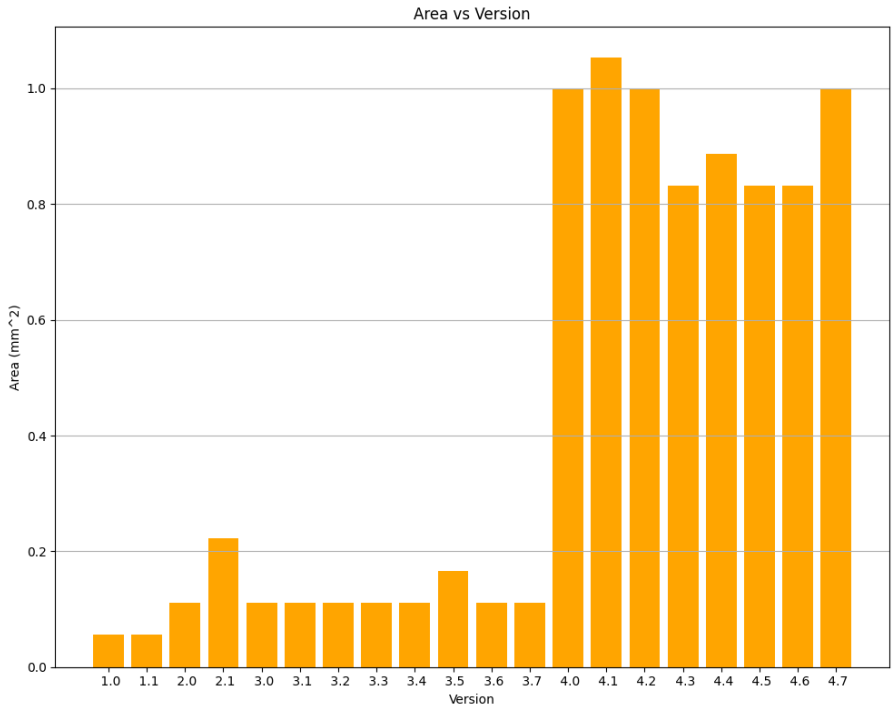


Figure 6.6: Area vs Version

6.5 Throughput per Area of KiloCore Implementations

This section discusses the overall area efficiency of each design, considering the throughput of each implementation against the area utilized. The throughput numbers are taken from section 5.2, and area taken from 5.4.

Figure 6.7: Throughput per Area @ 1.2 GHz

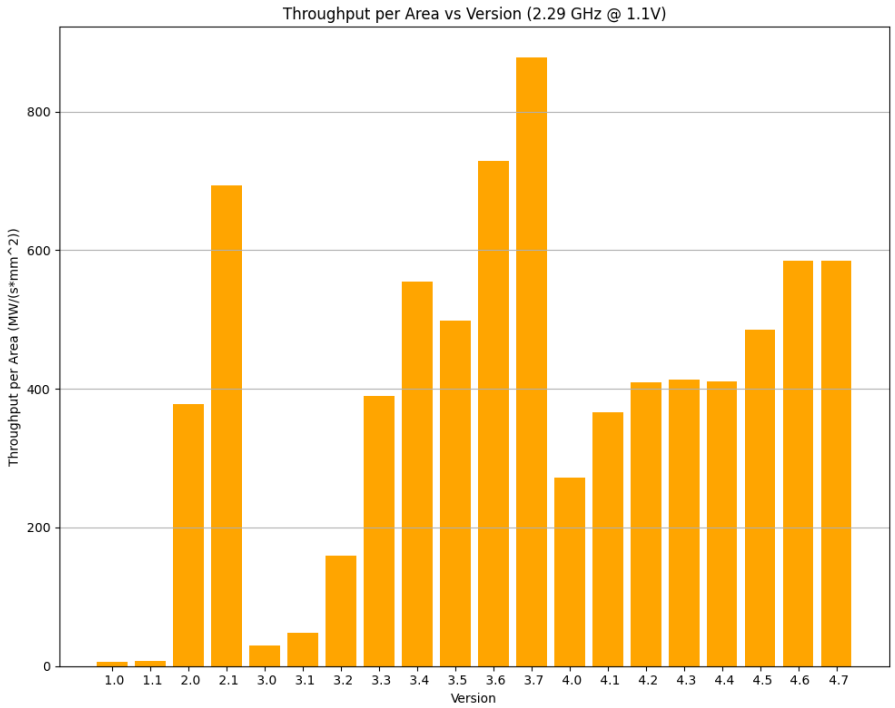


Figure 6.8: Throughput per Area @ 2.29 GHz

Figures 6.7 and 6.8 show area efficiency for each version of the DCT at 1.2 and 2.29 GHz respectively. Where throughput and accuracy generally increased with each version, overall throughput per area does not entirely follow this trend. Implementations that stand out as the most efficient are versions 2.1, 3.6, and 3.7 at 2.29 GHz, which are all single-engine implementations. Version 3.7 shows the best efficiency, with version 3.6 being a runner-up. Compared to version 2.1, the actual measured throughput is considerably lower in version 3.7; however, since version 2.1 is a 4-core implementation, it has lower area efficiency compared to versions 3.6 and 3.7.

A significant drop in efficiency occurred for versions 4.0-4.7; at this point, multi-engine implementations were being considered. If only the raw DCT cores were being measured, these versions would have the same efficiency as version 3.6 and 3.7, depending on which DCT engine was used; however, fan-out and fan-in mechanisms must be handled by additional cores, increasing the overall area usage where the addition of another engine results in three additional cores per each design. Efficiency asymptotes after about 5 engines are implemented, as shown when comparing versions 4.6 and 4.7 where there is no notable difference. This should hold for up to a fan-out of 8; however, if the fan-out exceeds 8 (the number of outputs per core), an additional fan-out core is required, adding 4 additional cores instead of 3.

6.6 Power of KiloCore Implementations

This section briefly discusses power consumption from each DCT implementation. Power results are taken directly from the KiloCore simulator, and are measured at 1.2 and 2.29 GHz. Much like area, power draw is separated into two sections, single-engine and multi-engine implementations, where power draw is primarily a function of the number of cores and secondarily a function of its computational complexity. Version 2.1 stands out as it is a 4-core design, thus yielding higher power. As power is both scaled by frequency and voltage, results at 1.2 GHz consume less power.

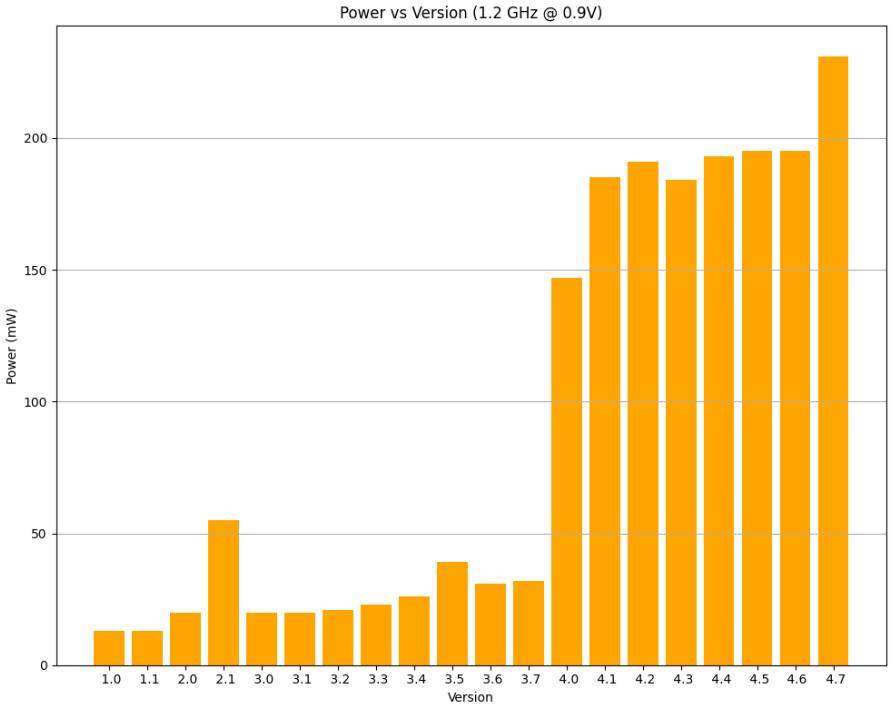


Figure 6.9: Power vs Version @ 1.2 GHz

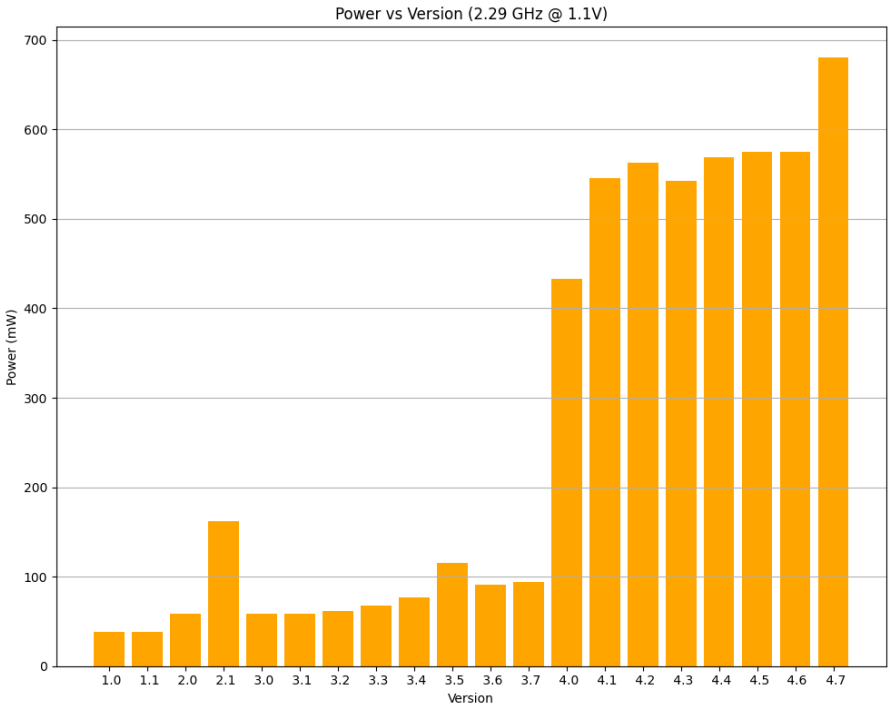


Figure 6.10: Power vs Version @ 2.29 GHz

6.7 Energy per Word of the Various KiloCore Implementations

This section discusses the energy required to compute a single word, where a word is a single matrix element. The power numbers are taken from section 5.6, and throughput from 5.2.

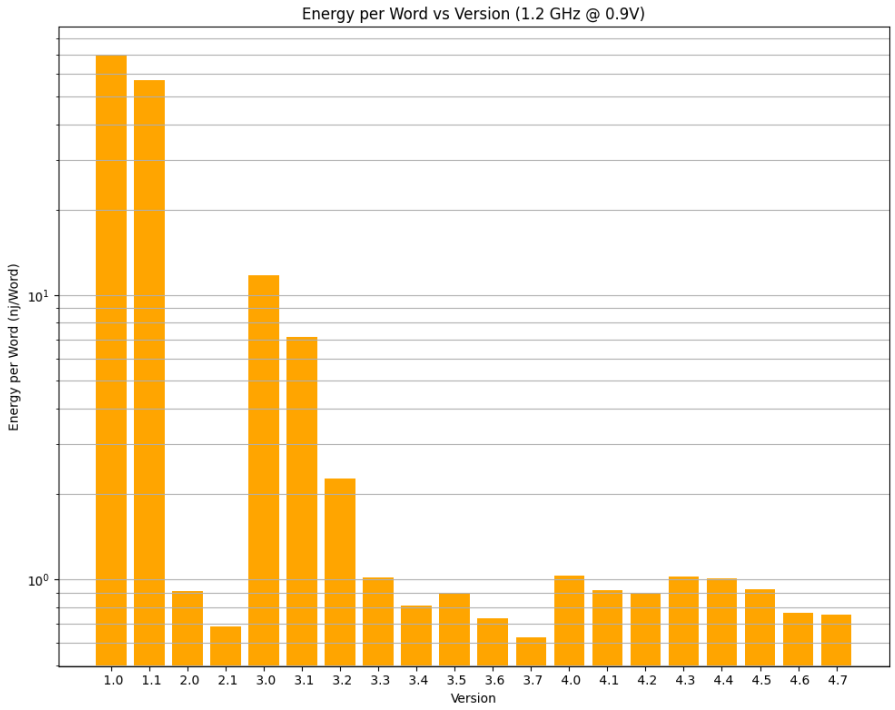


Figure 6.11: Energy per Word vs Version @ 1.2 GHz

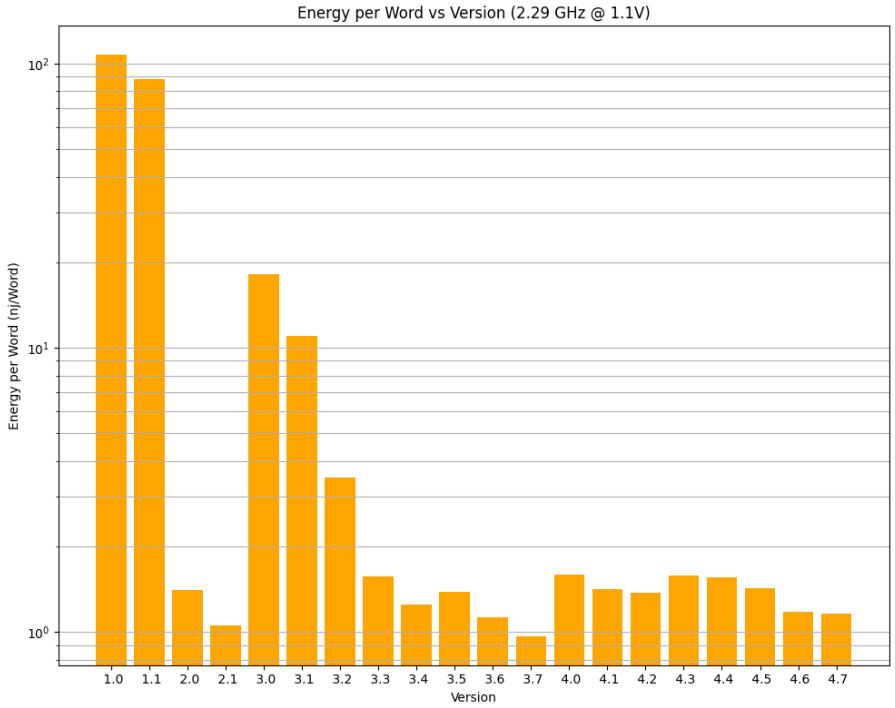


Figure 6.12: Energy per Word vs Version @ 2.29 GHz

Figures 6.11 and 6.12 show the energy per word results at 1.2 and 2.29 GHz respectively. Energy consumed per word computed generally trends down, with an exception for the AAN DCT implementations in version 2.0 and 2.1. The most energy-efficient implementations are versions 2.1, 3.6, and 3.7 at 1.2 GHz, which align closely with the most area-efficient results in section 5.5. Version 3.7 is the most energy-efficient of the three, with version 2.1 being a runner-up.

6.8 Area Normalized Energy Delay Product of KiloCore Implementations

This section discusses the energy-delay product for each DCT implementation. Individual throughput and energy metrics are taken from sections 6.5 and 6.7 respectively.

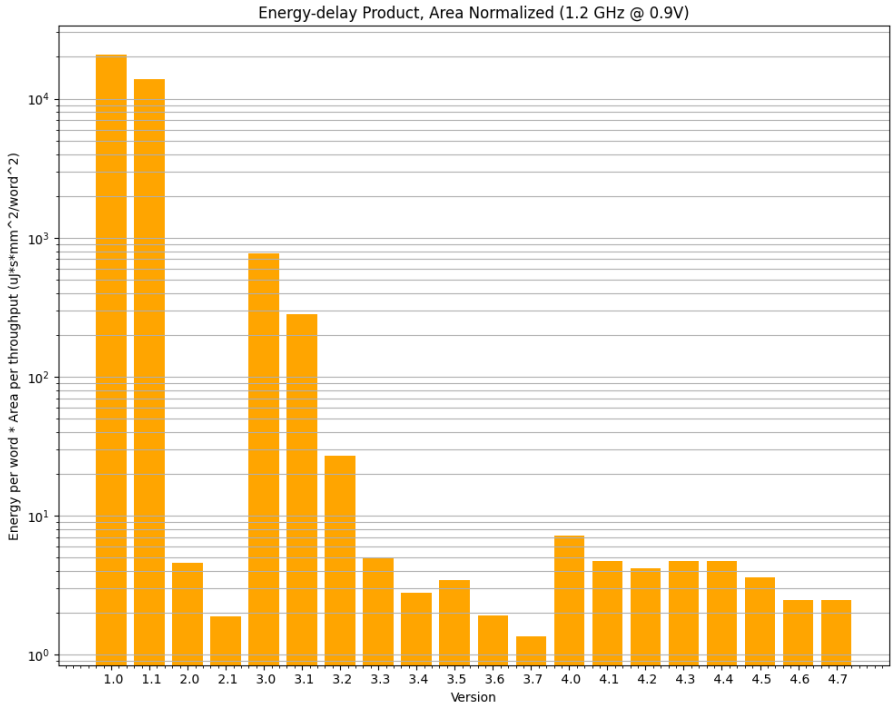


Figure 6.13: Energy Delay Product (Area Normalized) vs Version @ 1.2 GHz

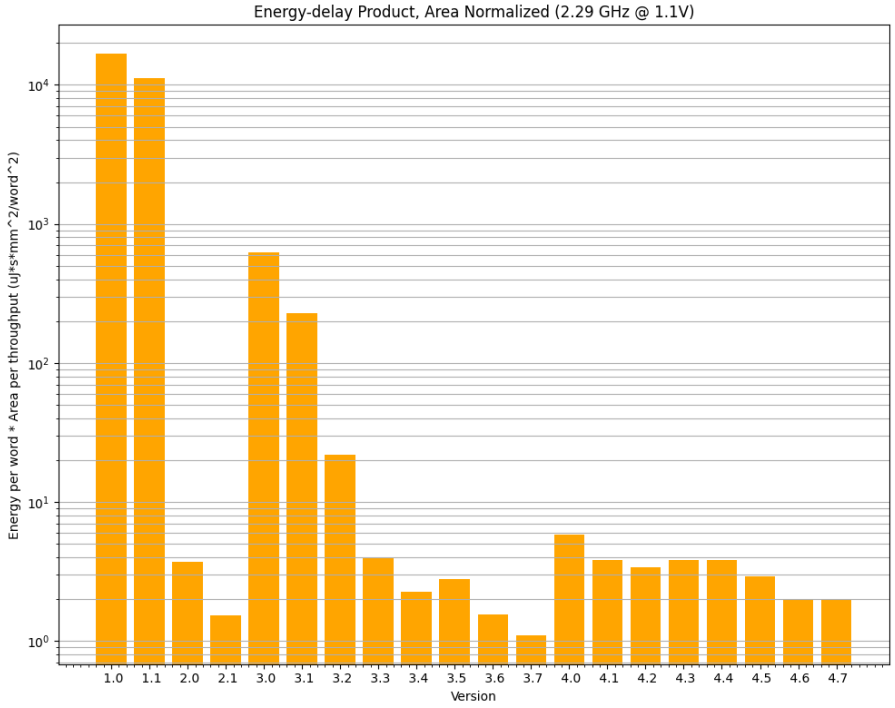


Figure 6.14: Energy Delay Product (Area Normalized) vs Version @ 2.29 GHz

The energy-delay product (EDP) is a simple measure of how a system balances its energy usage with its performance. However, as the DCT can be arbitrarily scaled, EDP is normalized with area and uses the throughput per area results instead of pure throughput. Figures 6.13 and 6.14 show the energy-delay product, normalized to the area, for each version, with the metric generally trending down with each version, with an exception in versions 2.0 and 2.1 which explored the AAN DCT. Multi-engines see an increase in this metric as the area drastically increases to account for multiple engines. A lower value is better in this category.

This metric accounts for both area efficiency and energy efficiency, weighing both values equally. This showcases which designs have the best overall trade-off between energy, area, and throughput. As discussed in previous sections, version 3.7 at 2.29 GHz beats all other implementations in both energy per word and throughput per area, which is exemplified in Figures 6.13 and 6.14.

6.9 Energy per Word vs Area per Throughput of KiloCore Implementations

This sections shows the tradeoff between the energy consumption per word and the area per throughput. Figures in this section are plotted logarithmically. Points closer to the origin are better in this case.

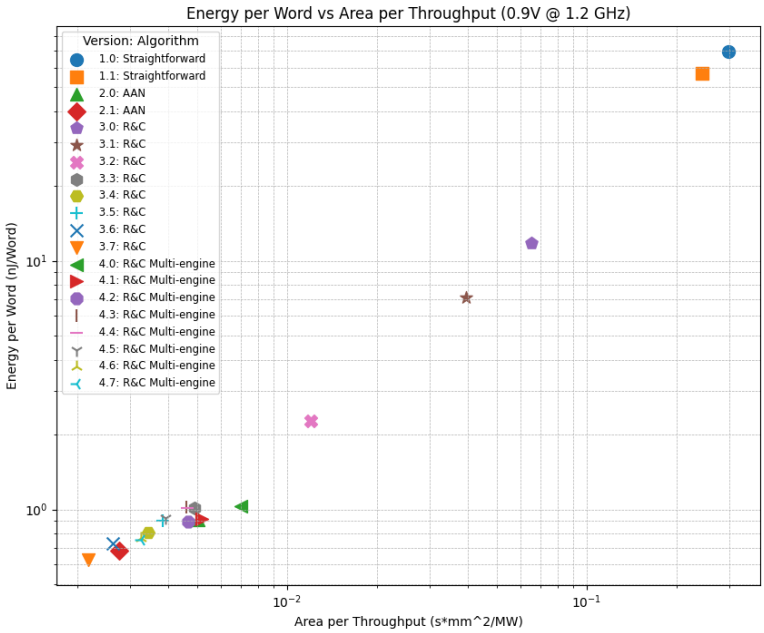


Figure 6.15: Energy per Word vs Area per Throughput @ 1.2 GHz

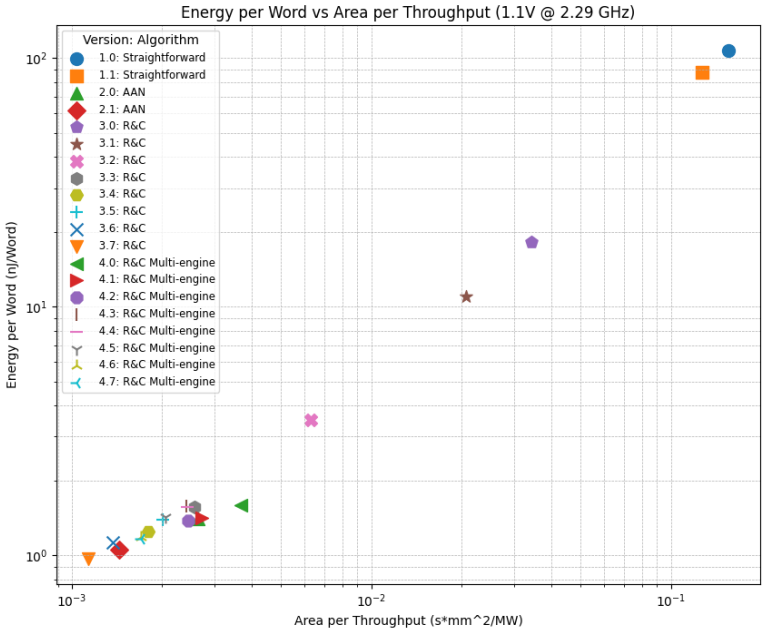


Figure 6.16: Energy per Word vs Area per Throughput @ 2.29 GHz

Figures 6.15 and 6.16 showcase all 20 implementations. Two additional figures are provided that zoom in on designs closest to the origin.

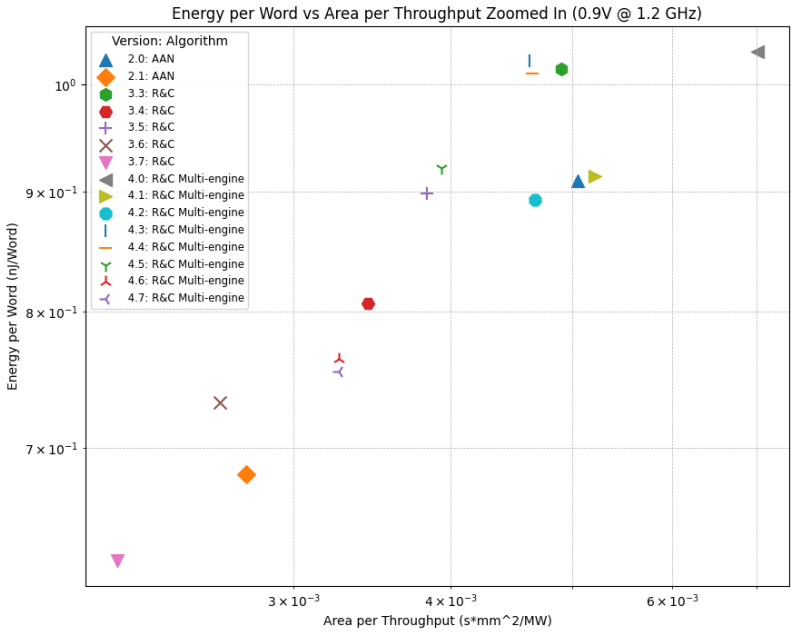


Figure 6.17: Energy per Word vs Area per Throughput Zoomed in @ 1.2 GHz

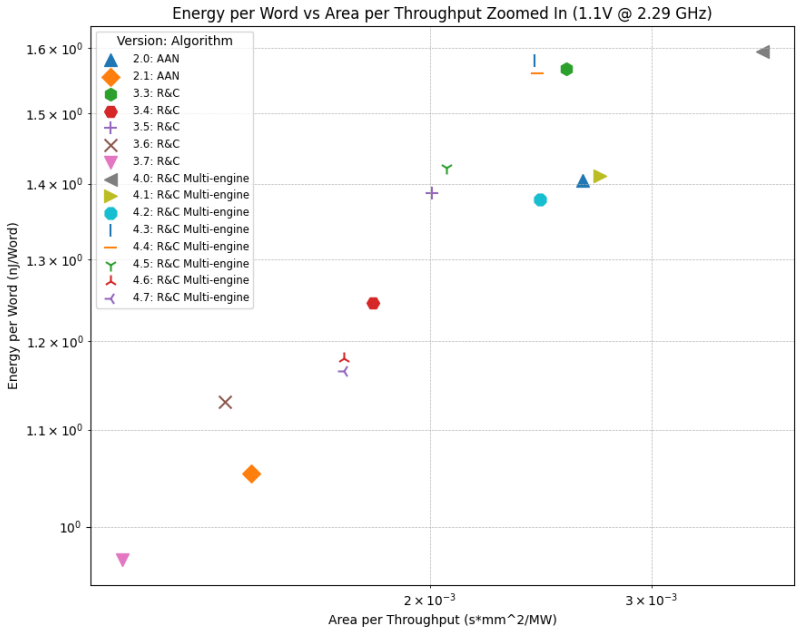


Figure 6.18: Energy per Word vs Area per Throughput Zoomed in @ 2.29 GHz

Figures 6.17 and 6.18 show the energy per word vs area per throughput for designs closest to the origin in 1.2 and 2.29 GHz respectively. Version 3.7 for both 1.2 and 2.29 GHz is the closest to the origin compared to all other versions, beating all other implementations in both metrics. Two runner-ups to version 3.7 are version 3.6 and version 2.1, where version 3.6 is more area-efficient and version 2.1 is more energy-efficient. Versions at 1.2 GHz tend to have better energy efficiency, whereas versions at 2.29 GHz showcase better area efficiency.

6.10 Summary

DCT version 4.7 boasts the highest throughput, alongside being one of the more energy-efficient and highest throughput per area implementations. This comes at the cost of having a higher area overall as it is a multi-engine implementation.

DCT version 3.7, although not being the fastest single-engine implementation, is both the best in terms of area and energy efficiency with high throughput utilizing only two cores.

It additionally has the best SNR (version 4.7 uses version 3.7) among other implementations due to optimizations in how data is extracted from the 40-bit accumulator. This includes the removal of rounding techniques that were seen in implementations that did not use the MAC. Version 3.7 @ 1.2 GHz achieves the best energy per word, and version 3.7 @ 2.29 GHz achieves the highest throughput per area.

Version	Throughput (MW/s)	SNR (dB)	Area (mm^2)	Throughput per Area (MW/s*mm^2)	Power (W)	Energy per Word (nJ/Word)	EDP, Area Normalized ($(\mu J*s*mm^2/word^2)$)
1.0	0.1869	57.77	**0.05545**	3.371	**0.01300**	69.56	20640
1.1	0.2280	57.87	**0.05545**	4.112	0.01300	57.02	13870
2.0	21.98	61.02	0.1109	198.2	0.02000	0.9100	4.592
2.1	80.62	61.02	0.2218	363.5	0.05500	0.6820	1.877
3.0	1.701	63.32	0.1109	15.34	0.02000	11.76	766.6
3.1	2.799	63.45	0.1109	25.24	0.02000	7.145	283.1
3.2	9.255	64.33	0.1109	83.45	0.02100	2.269	27.19
3.3	22.65	64.33	0.1109	204.3	0.02300	1.015	4.970
3.4	32.24	64.33	0.1109	290.7	0.02600	0.8060	2.774
3.5	43.41	64.33	0.1664	260.9	0.03900	0.8980	3.443
3.6	42.33	64.33	0.1109	381.7	0.03100	0.7320	1.919
3.7	51.03	**68.41**	0.1109	**460.1**	0.03200	**0.6270**	**1.363**
4.0	142.4	64.33	0.9981	142.7	0.1470	1.032	7.238
4.1	202.4	64.33	1.054	192.1	0.1850	0.9140	4.758
4.2	213.9	64.33	0.9981	214.3	0.1910	0.8930	4.168
4.3	179.9	64.33	0.8318	216.3	0.1840	1.023	4.730
4.4	191.0	64.33	0.8872	215.3	0.1930	1.010	4.693
4.5	211.6	64.33	0.8318	254.4	0.1950	0.9210	3.621
4.6	255.1	64.33	0.8319	306.7	0.1950	0.7640	2.492
4.7	**306.1**	**68.41**	0.9981	306.7	0.2310	0.7550	2.461

Table 6.1: Results @ 1.2 GHz

Version	Throughput (MW/s)	SNR (dB)	Area (mm^2)	Throughput per Area (MW/s*mm^2)	Power (W)	Energy per Word (nJ/Word)	EDP, Area Normalized ((μJ*s*mm^2/$word^2$))
1.0	0.3570	57.77	**0.05545**	6.432	**0.03800**	107.4	16690
1.1	0.4350	57.87	**0.05545**	7.847	0.03800	88.02	11220
2.0	41.94	61.02	0.1109	378.2	0.05900	1.405	3.715
2.1	153.9	61.02	0.2218	693.7	0.1620	1.053	1.518
3.0	3.246	63.32	0.1109	29.27	0.05900	18.15	620.1
3.1	5.341	63.45	0.1109	48.16	0.05900	11.03	229.0
3.2	17.66	64.33	0.1109	159.3	0.06200	3.503	21.99
3.3	43.23	64.33	0.1109	389.8	0.06800	1.567	4.021
3.4	61.53	64.33	0.1109	554.8	0.07700	1.245	2.244
3.5	82.84	64.33	0.1664	498.0	0.1150	1.387	2.785
3.6	80.77	64.33	0.1109	728.3	0.09100	1.131	1.552
3.7	97.38	**68.41**	0.1109	**878.1**	0.09400	**0.9680**	**1.102**
4.0	271.7	64.33	0.9981	272.2	0.4330	1.594	5.855
4.1	386.3	64.33	1.054	366.6	0.5450	1.411	3.849
4.2	408.1	64.33	0.9981	408.9	0.5630	1.379	3.372
4.3	343.2	64.33	0.8318	412.7	0.5420	1.579	3.827
4.4	364.5	64.33	0.8872	410.9	0.5690	1.560	3.797
4.5	403.9	64.333	0.8318	485.6	0.5740	1.423	2.930
4.6	486.8	64.333	0.8318	585.3	0.5740	1.180	2.016
4.7	**584.1**	**68.41**	0.9981	585.2	0.6805	1.165	1.991

Table 6.2: Results @ 2.29 GHz

Chapter 7

Comparisons to Other Discrete Cosine Transform Implementations

7.1 Overview

This section compares performance metrics of the KiloCore DCT implementations against general-purpose devices. The technology node, throughput, area, execution time, throughput per area, power, and energy per block are considered. All versions in this section are tested against a 2048 by 2048 grayscale image, where only the size and type of image are necessary for comparison, as this only looks at the DCT in isolation.

For this comparison, DCT implementation 3.7 stood out the most for highest efficiency and highest throughput respectively. Results are taken from [] which represent results on the NVIDIA Tesla T4 [15], NVIDIA GT-720m [3], Intel i5-4200M [3], and Altera Stratix III [16]. Since power and area are not discussed in [15] and [3] for the GPU and CPU results, both metrics were taken from each product's specifications [17] [18] [19], where the devices' TDP are used to represent power consumption and die area to define the area. The number of cores utilized for the i5-4200M was not stated either, and so results assume only a single core was used, or half the die area. Because throughput was not stated in these findings, throughput was extrapolated with Equation 6.1. Two additional KiloCore entries

are added to the table to reflect the highest possible throughput (24-engine design) and most energy-efficient designs (V3.7 at 115 MHz).

$$Throughput = \frac{\#\ of\ pixels}{Execution\ Time} \tag{7.1}$$

Device	Technology	Throughput (MW/s)	Area (mm^2)	Execution Time (ms)	Throughput per Area (MW/s*mm^2)	Power (W)	Energy per Word (nJ/Word)	Energy Delay Product (μJ*s/$word^2$)
NVIDIA Tesla T4	12nm	6533	545.0	0.6420	11.99	70.00	10.71	893.2
Altera Stratix III	90nm	2234	133.0	1.870	16.80	0.4400	0.1961	11.67
NVIDIA GT-720m	28nm	917.6	116.0	4.570	7.910	16.50	17.98	2273
Intel i5-4200M	22nm	22.54	65.00	186.1	0.3470	18.50	820.9	2367000
KiloCore V3.7 @ 1.2 GHz	32nm	51.03	**0.1109**	82.19	459.9	0.03200	0.6270	1.363
KiloCore V3.7 @ 2.29 GHz	32nm	97.38	**0.1109**	43.07	**877.6**	0.09400	0.9680	**1.103**
KiloCore V3.7*24 @ 2.29 GHz	32nm	2290	3.992	1.831	573.6	2.722	1.188	2.071
KiloCore V3.7 @ 115 MHz	32nm	4.888	**0.1109**	858.1	44.04	**0.001320**	0.2700	6.131

Table 7.1: Data Comparison Between KiloCore and General Purpose Hardware

The area for the Altera Stratix III was not given, so a crude estimation of the area was made based on the number of (logic units, area, etc.) specified in [16] and the Stratix III specification sheet [20].

To represent all devices with an even playing field, numbers have also been scaled to 32nm. Accurate scaling between different technologies can be done with polynomial equations discussed in the following work [21]. In particular, two important equations are required when scaling for delay and energy. For the Altera Stratix III, FPGA frequency scaling follows a different, slower trend than generally offered when scaling with technology,

where scaling is more dependent on wire delay than on technology. As a result, the original frequency scaling is maintained.

As described in [21], delay and power require scaling factors from the source technology and the target technology. The scaling factors for delay and power are shown in Equations 7.2 and 7.3.

$$DelayFactor = a_{d3}V^3 + a_{d2}V^2 + +a_{d1}V + a_{d0} \tag{7.2}$$

$$EnergyFactor = a_{e2}V^2 + a_{e1}V + a_{e0} \tag{7.3}$$

To scale to another technology, Equations 7.3 and 7.4 are used:

$$Delay_x = \frac{DelayFactor_x}{DelayFactor_y} * Delay_y \tag{7.4}$$

$$Energy_x = \frac{EnergyFactor_x}{EnergyFactor_y} * Energy_y \tag{7.5}$$

In Equations 7.2 and 7.3, V represents the operating voltage of the device. As voltage numbers were not explicitly given by any additional papers discussed in this work, an operating voltage of 1V was assumed for all devices except KiloCore. Coefficients in Equations 7.2 and 7.3 are discussed in [21].

Finally, scaled power is taken from the following Equation 7.6:

$$Power = \frac{Energy\ per\ word * Throughput}{1000} \tag{7.6}$$

Device	Scaled Through-put (MW/s)	Scaled Area (mm^2)	Scaled Execution Time (ms)	Scaled Throughput per Area (MW/s*mm^2)	Scaled Power (W)	Scaled Energy per Word (nJ/Word)	Scaled Energy Delay Product (μJ*s/$word^2$)
NVIDIA Tesla T4	2428	2774	1.727	0.8753	146.1	60.16	68730
Altera Stratix III	2234	21.28	1.877	105.0	0.1210	0.05417	0.5160
NVIDIA GT-720m	862.0	162.4	4.860	5.307	20.92	24.28	4574
Intel i5-4200M	19.12	115.9	213.7	0.1471	29.43	1539	10460000
KiloCore V3.7 @ 1.2 GHz	51.03	**0.1109**	82.19	459.9	0.03200	0.6270	1.363
KiloCore V3.7 @ 2.29 GHz	97.38	**0.1109**	43.07	877.6	0.09400	0.9680	1.103
KiloCore V3.7*24 @ 2.29 GHz	2290	3.992	1.831	573.6	2.722	1.188	2.071
KiloCore V3.7 @ 115 MHz	4.888	**0.1109**	858.1	44.04	**0.001320**	0.2700	6.131

Table 7.2: Data Comparison Between KiloCore and General Purpose Hardware, Scaled to 32nm

7.2 Throughput and Execution Time of Various Device Implementations

Compared to KiloCore version 3.7*24 at 2.29 GHz, the NVIDIA Tesla T4 showcases better throughput, yielding 1.06× better performance when both are scaled to 32nm. While all GPUs and CPUs discussed were scaled to an older technology, the Stratix III is scaled from 90nm to 32nm. As a result, it is the only device that sees improvement in performance. KiloCore still maintains better throughput compared to the Altera Stratix III, NVIDIA GT-720m, and Intel i5-4200M, beating these devices by 1.03×, 2.7×, and 120× respectively.

7.3 Area of Various Device Implementations

KiloCore has the lowest area compared to other devices, with version 3.7 being at least 192x when compared to the Altera Stratix III at 32nm. This is especially true compared to all other devices, even when unscaled. KiloCore is orders of magnitudes smaller than the Tesla T4, GT-720m, and i5-4200M (25000×, 1460×, and 1170× respectively).

7.4 Throughput per Area of Various Device Implementations

KiloCore version 3.7 @ 2.29 GHz has the highest throughput per area, beating the Altera Stratix III by 8.35× and up to 5966× when compared to the i5-4200M. This makes it the most area-efficient design and a compelling option for integration on SoCs with strict area constraints; designs requiring both minimal area and high throughput benefit greatly from utilizing KiloCore as an area-efficient, high-throughput, and programmable compute solution.

7.5 Energy per Block of Various Device Implementations

The Altera Stratix III beats all KiloCore implementations profiled, boasting about a 5x improvement in energy consumption per word processed compared to KiloCore's DCT version 3.7 at 115 MHz in which KiloCore operates at 0.56V. This difference grows for other KiloCore DCT implementations that operate at higher voltages and area, beating the next most efficient result by over 11×. However, KiloCore version 3.7 at 115 MHz is more energy efficient than general-purpose hardware, beating these implementations by at least 90× and up to 5700×.

7.6 Area Normalized Energy-Delay Product of Various Device Implementations

The Altera Stratix III boasts the highest energy-delay product, even when normalized to area, owing to impressive energy efficiency. In this particular metric, KiloCore DCT version 3.7 yields the highest energy-delay product among other KiloCore implementations. However, when compared to the Stratix III, falls short, with the Stratix III FPGA yielding 2.14× higher energy-delay product when scaled to 32nm.

The energy-delay product showcases designs that achieve the best balance between throughput, area, and area. While KiloCore falls short of the Stratix III, it only does so by a very small margin, owing the KiloCore's better throughput per area. Compared to other devices profiled, it beats the Tesla T4 by 62484×, the GT-720m by 4158×, and the i5-4200M by 9510000× when scaled to 32nm.

7.7 Energy per Throughput vs Area per Throughput of Various Device Implementations

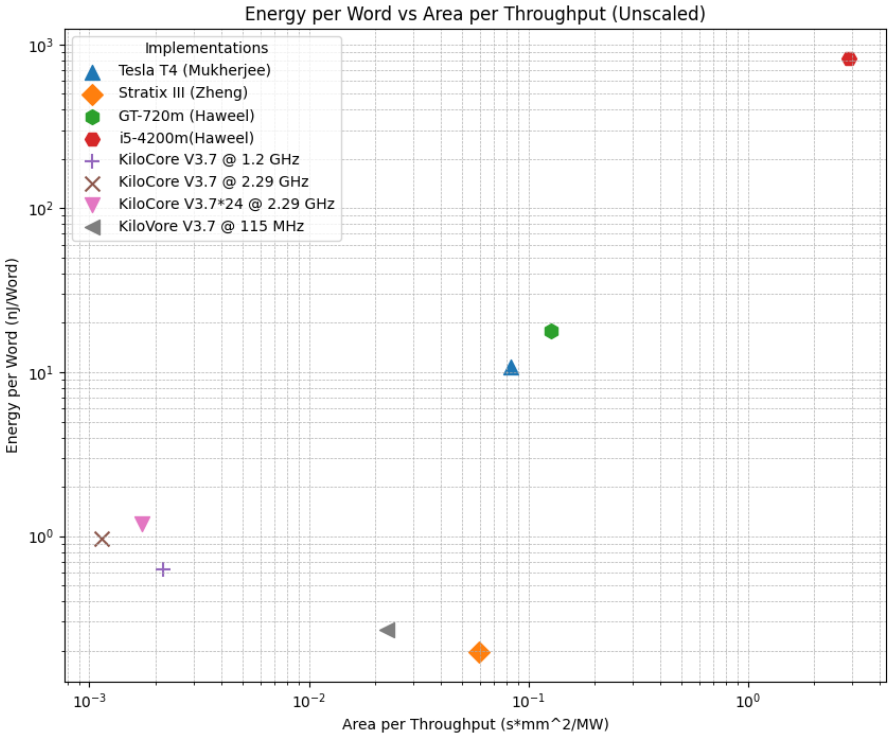


Figure 7.1: Energy per Word vs Area per Throughput Unscaled

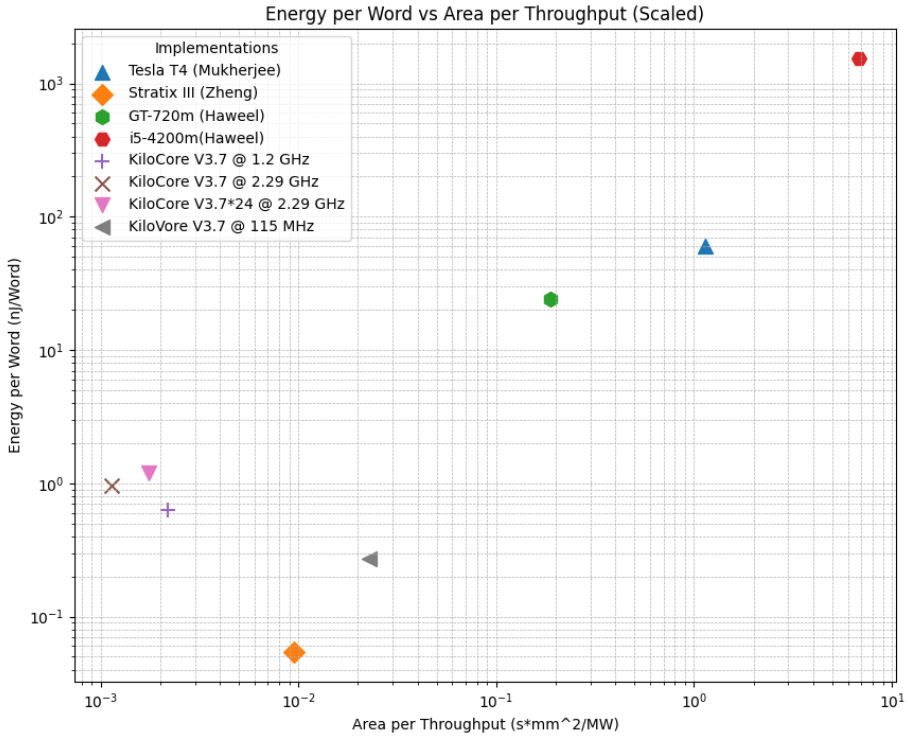


Figure 7.2: Energy per Word vs Area per Throughput Scaled

Figures 7.1 and 7.2 show the energy per word versus the area per throughput both unscaled and scaled. This graph showcases how a design balances area, throughput, and energy. Being closer to the origin is better in this case.

As shown in Figures 7.1 and 7.2, all KiloCore implementations (except version 3.7 at 115 MHz, 0.56V) beat all other devices profiled in throughput per area. Unscaled, it is orders of magnitude better than all other devices, with version 3.7 at 2.29 GHz beating all other KiloCore implementations in this metric.

However, KiloCore loses on energy per word compared to the Stratix, which showcased impressively low energy consumption. This is true when unscaled and further exemplified when scaled to 32nm, being roughly an order of magnitude better than KiloCore implementations.

7.8 Summary

KiloCore DCT implementations beat all other devices in terms of throughput and area (and thus, throughput per area). When scaled to 32nm, KiloCore was notably able to beat both GPU implementations as well as the FPGA in terms of throughput, yielding better performance up to nearly 120×. Differences in area are much greater, with KiloCore beating all devices in this particular metric by several orders of magnitude, at most over 10000×. As a result, KiloCore ultimately beats all profiled devices regarding area efficiency.

However, KiloCore DCT implementations fall short of the Altera Stratix III FPGA in terms of energy consumed per word. Although beating the NVIDIA Tesla T4, GT-720m, and Intel i5-4200M, KiloCore falls short of the Stratix III FPGA by roughly 5× at its best. This additionally translates into the energy product delay, which boasts an ultimately 2.14× advantage over the best KiloCore implementation in this instance.

Looking at these results overall, this work showcases two different options for DCT implementations, with KiloCore far more optimized in terms of area, and the Stratix III more optimized for energy. However, KiloCore's high throughput and area efficiency make it a competitive option compared to other general-purpose hardware on the market.

Chapter 8

Conclusion and Future Work

8.1 Thesis Summary

This work summarizes current research on implementations of the discrete cosine transform and inverse discrete cosine transform on a fine-grain many-core platform. This novel architecture, fine-grain many-core, is defined, and details of the KiloCore processor array, are explained.

Different DCT algorithms are defined and implemented in 20 different designs on the KiloCore processor array, targeting image-processing applications. These designs showcase architectural explorations enabled by this novel platform, utilizing both serial and parallel designs. Fine-grain many-core implementations are profiled and compared against several devices in throughput, area, power, energy, execution time, throughput per area, and energy-delay product. Data from all devices are scaled to 32nm.

Results showcase very high improvements in area and throughput per area, beating all devices in throughput per area by at least 8.35× and up to 5966× as well as in area by at least 192× and over 25000×. Although the Altera Stratix III boasts very impressive energy efficiency, beating KiloCore with a 5× improvement, this novel platform beats out the NVIDIA Tesla T4 by 223×, the NVIDIA GT-720m by 90× and the i5-4200m by over 5700×. These results showcase KiloCore as a very competitive option for SoC design and

integration in smaller devices.

8.2 Future Work

As it was only briefly discussed in this thesis, the exploration of a multi-stage DCT was not fully realized in this work due to time limitations. This was briefly looked at in version 2.1 KiloCore DCT, which implemented row and column DCTs as two separate stages that each computed half of a given matrix. This, however, was not explored in the explicit Row and Column algorithm. While very area-efficient, multi-engine DCTs received additional area penalties in fan-in cores. Exploration of splitting individual DCT engines into multiple stages may help reduce this penalty as the necessity for additional fan-in (and potentially fan-out) cores may be reduced.

Additionally, only three DCT algorithms were explored in this work: the straightforward 2D DCT, the AAN DCT, and Row and Column DCT. Many works cited in this thesis, among other papers, have explored novel DCT algorithms that may further reduce computational complexity on a fine-grain many-core platform. Exploration of these algorithms may further improve single-engine performance.

Chapter 9

Bibliography

[1] Stanford University, "JPEG Compression: Discrete Cosine Transform (DCT)," n.d. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/dct.htm

[2] A. Obukhov and A. Kharlamov, "Discrete Cosine Transform (DCT) 8x8 Module for CUDA," NVIDIA Corporation, Tech. Rep., 2008. [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/dct8x8.pdf

[3] R. T. Haweel, W. S. El-Kilani, and H. H. Ramadan, "GPU implementation of a modified signed discrete cosine transform," in *2014 9th International Conference on Informatics and Systems*, 2014, pp. PDC–68–PDC–73.

[4] J. Zhou and P. Chen, "Generalized Discrete Cosine Transform," in *2009 Pacific-Asia Conference on Circuits, Communications and Systems*, 2009, pp. 449–452.

[5] "IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform," *IEEE Std 1180-1990*, pp. 1–12, 1991.

[6] M. Popović and T. Stojić, "The fast computation of DCT in JPEG algorithm," in *9th European Signal Processing Conference (EUSIPCO 1998)*, 1998, pp. 1–4.

[7] B. Baas, Z. Yu, M. Meeuwsen, O. Sattari, R. Apperson, E. Work, J. Webb, M. Lai, T. Mohsenin, D. Truong, and J. Cheung, "AsAP: A fine-grained many-core platform for DSP applications," *IEEE Micro*, vol. 27, no. 2, pp. 34–45, Mar. 2007.

[8] 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 *IEEE HotChips Symposium on High-Performance Chips*, Aug. 2016.

[9] B. Bohnenstiehl, "Design and Programming of the KiloCore Processor Arrays," Ph.D. dissertation, University of California, Davis, CA, USA, Mar. 2020, http://vcl.ece.ucdavis.edu/pubs/theses/2020-1.bbohnenstiehl/.

[10] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[11] T. Abbott and B. Baas, "A Scalable JPEG Encoder on a Many-Core Array," in *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2023, pp. 411–418.

[12] P. Zhu, J. Liu, S. Dai, and G. Wang, "Scaled aan for fixed-point multiplier-free idct," *EURASIP J. Adv. Sig. Proc.*, vol. 2009, 12 2009.

[13] 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, Apr. 2017.

[14] D. Vasoya, P. Samir, S. Patel, and S. Pradhan, "Performance Optimization of Transformation Technique -DCT using NVIDIA CUDA," 2010.

[15] D. Mukherjee, "Parallel implementation of discrete cosine transform and its inverse for image compression applications," *The Journal of Supercomputing*, vol. 80, pp. 1–24, 07 2024.

[16] M. Zheng, J. Zheng, Z. Chen, L. Wu, X. Yang, and N. Ling, "A Reconfigurable Architecture for Discrete Cosine Transform in Video Coding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 3, pp. 810–821, 2020.

[17] TechPowerUp, "Nvidia tesla t4 specs," n.d. [Online]. Available: https://www.techpowerup.com/gpu-specs/tesla-t4.c3316

[18] Intel Corporation, "Intel Core i5-4200M Processor (3M Cache, up to 3.10 GHz) Specifications," n.d. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/76348/intel-core-i54200m-processor-3m-cache-up-to-3-10-ghz/specifications.html

[19] TechPowerUp, "NVIDIA GeForce GT 720M Specs," n.d. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-gt-720m.c2297

[20] Intel Corporation, "Intel Stratix 10 Device Overview," n.d. [Online]. Available: https://www.intel.com/programmable/technical-pdfs/654316.pdf

[21] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm," *Integration, the VLSI Journal*, vol. 58, pp. 74–81, 2017, http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/.

[22] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A fine-grained 1,000-processor array for task parallel applications," *IEEE Micro*, vol. 37, no. 2, pp. 63–69, Mar. 2017.

[23] A. Stillmaker, "Design of energy-efficient many-core MIMD GALS processor arrays in the 1000-processor era," Ph.D. dissertation, University of California, Davis, Davis, CA, USA, Dec. 2015, http://vcl.ece.ucdavis.edu/pubs/theses/2015-1.stillmaker/.

[24] 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*, Jun. 2016.

[25] B. M. Baas, "A parallel programmable energy-efficient architecture for computationally-intensive DSP systems," in *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, Nov. 2003.