

Embedded Discrete Cosine Transform

SENG 440 Final Project Report

Joel Chamberlain (SENG) - V00928917

joelchamberlain@uvic.ca

Mattias Kroeze (CENG) - V00934043

kroeze@uvic.ca

Abstract

In this project, we set out to implement the Discrete Cosine Transform (DCT) algorithm in an embedded systems context. To complete this, we would implement the DCT using Loeffler's Algorithm and further improve it by applying software optimization techniques that are relevant for the given algorithm. To test the implementation that we developed, we created a test bench that will outline the performance of each iteration of the algorithm, each of which used a different type of optimization technique, followed by a fully optimized final version. We had success in exploring different optimizations that could be used for the DCT algorithm and found large improvements when using NEON.

Introduction

Digital Signal Processing (DSP) is a key tool in modern computing, used in a varying range of applications such as communication and multimedia. The Discrete Cosine Transform (DCT) is one such DSP tool that is widely used as a pre-processing step in multimedia compression. Although it has the potential to be quite powerful, specific implementations vary in efficiency given the large amount of computation required. As embedded systems get smaller, the need for more efficient implementations of algorithms is crucial. This applies to products and services such as camera sensors, streaming, and IoT devices.

The DCT at its core is a mathematical technique used to transform a signal from the time domain to the frequency domain, which is typically then used in tandem with compression techniques. The DCT is typically used in JPEG image processing, usually processing 8x8 blocks of data at a time. At its simplest form, the DCT is an extremely expensive operation to perform and given that there is a large demand for the DCT to be used in small systems, there have been a plethora of different implementations that attempt to tackle the efficiency of this algorithm.

Although there are multiple versions of the DCT algorithm, Loeffler's Algorithm is a popular implementation in both software and hardware solutions given that the number of calculations have been reduced significantly (namely 64 multiplications to 11) while maintaining consistency. Given this increase in efficiency, it is incredibly useful when working with embedded systems given the need for this reduction in computational load.

This project focuses on implementing the 8x8 DCT using Loeffler's algorithm on an ARM Cortex-A15 processor, emulated via QEMU. The ARM Cortex-A15 is a high-performance member of the ARMv7-A architecture family, supporting SIMD instructions through the NEON engine, which can be leveraged to further accelerate the DCT. The choice of an emulated environment allows controlled performance testing while preserving the characteristics of an embedded ARM system.

Objectives

The objectives of this project are to:

1. **Implementation:** Create a working implementation of the 8x8 DCT using Loeffler's algorithm in C, targeted for an ARM system.
2. **Optimization:** Investigate compiler-level optimizations, algorithmic refinements, and potential SIMD acceleration using NEON instructions.
3. **Performance Analysis:** Measure the performance of the different DCT implementations we have implemented and experimented with, as well as compare and contrast the results.

Specifications

- **Processor:** ARM Cortex-A15 (emulated via QEMU virt-2.11 machine type)
- **Architecture:** ARMv7-A (32-bit)
- **Compiler:** gcc 8.2.1

Performance Achieved

The performance achieved for the fully optimized version of our Loeffler's DCT implementation was quite fast in comparison to our other less favourable implementations, with a speed increase of 78.2%.

Contributions

- Implemented Loeffler's algorithm for the DCT in C.
- Optimized for the ARM Cortex-A15 architecture using NEON SIMD to enable parallel floating point operations.
- Applied compiler optimization flags.
- Reduced memory overhead by storing intermediate results in registers.
- Created a test bench to test the effectiveness of our implementations.

Report Organization

The following sections discuss the function of each section in the report as well as their purpose.

Abstract

Summarize the approach, process, functionality, successes, and shortcomings of the project as a whole.

Introduction

Introduce the project and what we want to achieve, outline the specifications, the performance achieved, the contributions, as well as outlining the different sections of the report.

Background

Discuss the problem that this project is tasked to complete, as well as any information that is required to know before understanding the problem that we are trying to solve.

Algorithm Design

Discuss the chosen algorithm design, what were the decisions for critical portions of the algorithm such as how we are dealing with values, registers, variables, and what bit width we chose.

C Code and Optimization

Outline snippets of the C code that was developed, and highlight the optimizations used to improve the code performance.

Compilation and Assembly

Describe the process of compiling the DCT implementation for the ARMv7-A architecture, including the toolchain used, compiler flags, generation of assembly output, and how the build was tailored for execution on the QEMU-emulated Cortex-A15 platform.

Results

Enumerate the performance of the developed solution, discuss the overall results of the project.

Improvements

Discuss possible improvements that could be made to the developed solution given the current implementation.

Conclusions

Draw conclusions from the project.

Background

Discrete Cosine Transform (DCT)

The Discrete Cosine Transform (DCT) is a mathematical technique used to transform a signal or image from its spatial or time domain to the frequency domain which aids in data compression techniques. The DCT is equivalent to the Discrete Fourier Transform (DFT) of real and even functions and it differs from the DFT's use of complex numbers by using only real numbers, making it simpler. The DCT has an energy compaction property which is very effective. It outlines that most of the signal energy is concentrated in the few low frequency coefficients, which makes the DCT very effective in calculation, since few DCT coefficients can represent an entire block of pixels.

It is a pivotal technique in signal processing, data compression, mostly used in image and video compression for use in formats such as JPEG for images and MPEG for video. N by N blocks of data are processed with a common choice for N is 8. As multimedia devices get smaller, the need for optimizing this technique is pivotal in how people will consume multimedia, as it will need to have increased performance on smaller embedded machines.

Mathematical Background

There are a few versions of the DCT, but in its most simple form we have the 1D DCT. The 1D DCT for inputs $x(i)$ and outputs $X(u)$ is defined as:

$$X(u) = \frac{C(u)}{2} \sum_{i=0}^7 x(i) \cos \frac{(2i+1)u\pi}{16} \quad C(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{for } u \neq 0 \end{cases}$$

Figure 1: 1D DCT Equation.

The 2D DCT, which is the version used in image compression, can be described as applying the 1D DCT to each row of the image matrix, followed by applying the 1D DCT to each column of the image matrix and can be defined as:

$$X(u, v) = \frac{C(u)}{2} \frac{C(v)}{2} \sum_{i=0}^7 \sum_{j=0}^7 x(i, j) \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16}$$

Figure 2: 2D DCT Equation.

Computational Challenges of DCT

This naive implementation of the DCT is computationally expensive given that this approach would yield a time complexity of $O(N^2)$ for an N by N matrix. Since this would also involve many floating point multiplications (which are quite expensive to calculate) the cost of the naive implementation of the DCT would be enormous and would not be an effective solution on an embedded system.

Loeffler's Algorithm

To address the issue of this high computational need, as well as power draw, there have been several implementations of the DCT that try to tackle these issues, making an efficient version of the DCT that can be utilized on embedded systems and smaller devices. Loeffler's Algorithm is one such algorithm that is a fast, factorized version of the 8-point DCT. It was developed in 1989 by Loeffler, Ligtenberg, and Moschytz. It minimizes the number of multiplications needed, making it ideal for systems where computation cost and power draw are a concern. It completes a theoretical minimum number of multiplications at 11 and 29 additions for an 8-point DCT. This algorithm was the basis for many JPEG image encoders, both in software and hardware.

The algorithm is structured into stages based on the transformation of an 8×8 input matrix into its corresponding frequency coefficients. These stages include:

- **Reflectors:** Combine symmetric input elements using addition and subtraction, forming the butterfly operations that reduce computational complexity.
- **Rotators:** Perform approximate cosine-based multiplications using fixed or reduced-precision constants, enabling efficient implementation on hardware with limited floating-point performance.
- **Reordering:** Arrange the outputs from the previous stages into their correct positions in the final DCT coefficient matrix.

A diagram of Loeffler's Algorithm can be seen below:

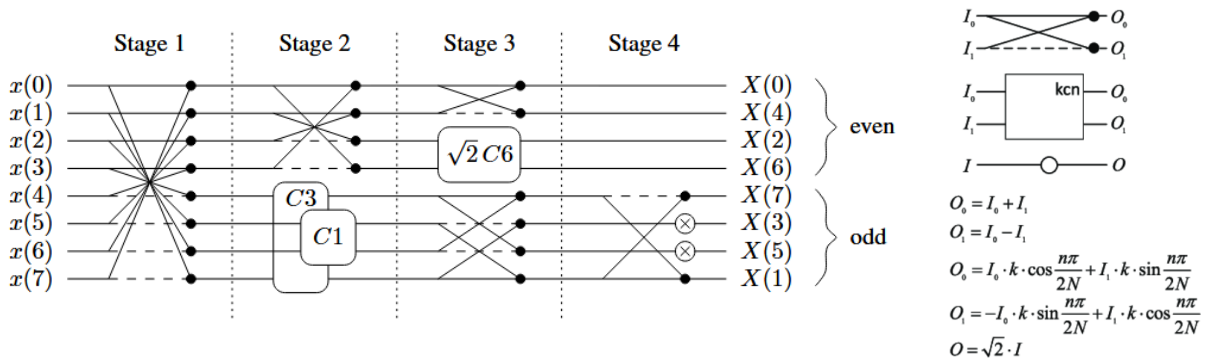


Figure 3: Diagram of Loeffler's Algorithm and Related Equations.

Embedded System Context

Since we are utilizing the Cortex-A15, which is a high-performance ARM processor based on the ARMv7-A 32-bit architecture, we can utilize NEON instructions, which allow for simultaneous operations on multiple data elements, further improving this implementation. NEON is quite useful especially in an embedded systems context, given that we can perform vectorized addition, subtraction, and multiplication efficiently, which will aid in improving the performance of the DCT in an embedded world.

Algorithm Design

For all intermediate values, we chose `int32_t` to retain sufficient precision while still using `int64_t` to temporarily hold products, before shifting to restore scale. With the largest NEON vector type (q registers), each register holds 4 lanes of 32-bit integers (`int32x4_t`), which allows us to vectorize the entire first stage of Loeffler's operation effectively. Using `int64_t` vectors would halve our vector width and provide more fractional precision than required, and `int16_t` would underutilize the available bit width and would provide more vectorized lanes than needed for the sum and diff operations.

To keep the implementation compliant with the integer-only restriction, we define a global fixed-point scale. To pick a scale that maximizes use of the 32-bit width, we first determine the maximum possible value anywhere across the Loeffler's data flow.

For an 8-bit input image, the worst case is an all-white block. In the first 1D pass, the highest term is the sum of the 8 inputs: $8 \times 255 = 2040$. After the second pass (to form the 2D DCT), the highest term sums eight of that value again, resulting in a maximum of 16320.

Therefore, the highest fixed-point scale that still fits within a signed 32-bit integer is Q17, since $16320 \ll 17 = 2139095040$, which is below the max representable signed value of 2147483647.

The input to our Loeffler implementation is an array of eight `int32_t` values pre-shifted to our global Q17 scale. We do this outside the Loeffler routine so the 2D pass does not shift and unshift twice. We left-shift the 8×8 input block once and rescale the output once at the end. Input images are stored as 1D vectors in `images.h`. We gather each 8×8 block with a sliding window and write the DCT result back to its corresponding location in a 320×320 output array.

Within the rotators, we multiply the operands by fractional constants. To satisfy the integer-only constraint, these constants are precomputed, scaled to Q17, rounded to their nearest integer, and stored in the header file. We define a `ROUND_SHIFT_Q` macro that adds $1 \ll 16$ before shifting right by our global Q17 scale to perform round-to-nearest. This normalizes our multiplication products back to Q17 and reverts our finalized DCT transformed values back to the integer scale.

To manage registers effectively, we use NEON in Stage 1 to compute all sums and differences in two vector operations. We load the low four inputs into one q register and the high four into another. We then reverse the entire high vector by flipping pairs using `vrev64q_s32` and rotating by two lanes with `vextq_s32`. Reversing the high vector lets us compute sums and diffs by aligned indices. Compiled assembly code is shown in the following block to demonstrate the use of NEON Q registers, along with the flip and rotate operations required to reverse lane order for the butterfly operations.

```
vld1.32    {d20-d21}, [r3]!    @ Load low bits into q10
vld1.32    {d16-d17}, [r3]    @ Load hi bits into q8
vrev64.32  q8, q8              @ Reverse to align indices
vext.8     q8, q8, q8, #8      @ Reverse to align indices
vsub.i32   q9, q10, q8         @ Calc diffs - store in q9
vrev64.32  q9, q9              @ Revert lane order
vext.8     q9, q9, q9, #8      @ Revert lane order
movw       s1, #12785          @ Start storing rotator constants
vmov.32    r7, d19[0]          @ Put calculated diffs into GPRs
vmov.32    r9, d18[1]          @ Put calculated diffs into GPRs
movw       r6, #64277          @ Start storing rotator constants
movw       r5, #52751          @ Start storing rotator constants
vmov.32    r4, d19[1]          @ Put calculated diffs into GPRs
movw       r1, #54491          @ Start storing rotator constants
movw       r8, #36410          @ Start storing rotator constants
movw       r2, #29126          @ Start storing rotator constants
vadd.i32   q8, q8, q10         @ Calc sums - store in q8
vmov.32    r3, d18[0]          @ Put calculated diffs into GPRs
```

From our empirical results, we only gained performance by vectorizing Stage 1. Beyond that point, we observed a significant performance hit, likely from the extra memory traffic and lane rearrangements needed for each subsequent operation. For the remaining stages, we operate in scalars, assigning each value to a variable that describes its stage and lane (e.g. `s1_7`). While we define more variable names than necessary and include many redundant renaming operations, the code is much easier to work with when following the provided dataflow diagram. Analysis of the resulting assembly code confirms that the compiler effectively handles register allocation and eliminates redundant renames, so there is no performance penalty. For example, while R2 contains the result of the stage 3 lane 0 sum, the compiler skips the redundant `s4_0 = s3_0` step and writes R2 directly to the output array's `v[0]`, as highlighted in red.

```
add    r2, lr, r3    @ s3_0 = s2_0 + s2_1
sub    lr, lr, r3    @ s3_1 = s2_0 - s2_1
add    r3, ip, r1    @ s4_7 = s3_7 + s3_4
sub    r1, r1, ip    @ s4_4 = s3_7 - s3_4
str    r2, [r0]      @ v[0] (s4_0)
str    lr, [r0, #16]  @ v[4] (s4_1)
str    s1, [r0, #24]  @ v[6] (s4_3)
stmib  r0, {r3, r6}   @ v[1], v[2] (s4_7, s4_2)
str    r1, [r0, #28]  @ v[7] (s4_4)
str    r4, [r0, #12]  @ v[3] (s4_5)
str    r8, [r0, #20]  @ v[5] (s4_6)
```


C Code and Optimization

This section will outline snippets of the C code that were used in the final optimized version of our version of Loeffler's algorithm implemented in C. It will outline the helper functions, then the full Loeffler DCT function stage by stage, along with the corresponding diagram of that stage.

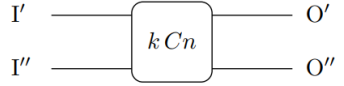
The versions of Loeffler's algorithm that were developed for this project are as follows:

- Routine: Implemented using routines only.
- Inline: Implemented using inline functions.
- Macro: Implemented using macros.
- Register: Implemented leveraging registers.
- NeonS1: Using NEON for Stage 1 in Loeffler's only.
- NeonReflect: Using NEON for the reflectors in all stages.
- NeonFull: Using NEON for reflectors and rotators in all stages.
- Optimized: A fully complete version, using the best optimizations from all other versions.

Let's walk through the final optimized version and highlight some important pieces in the code.

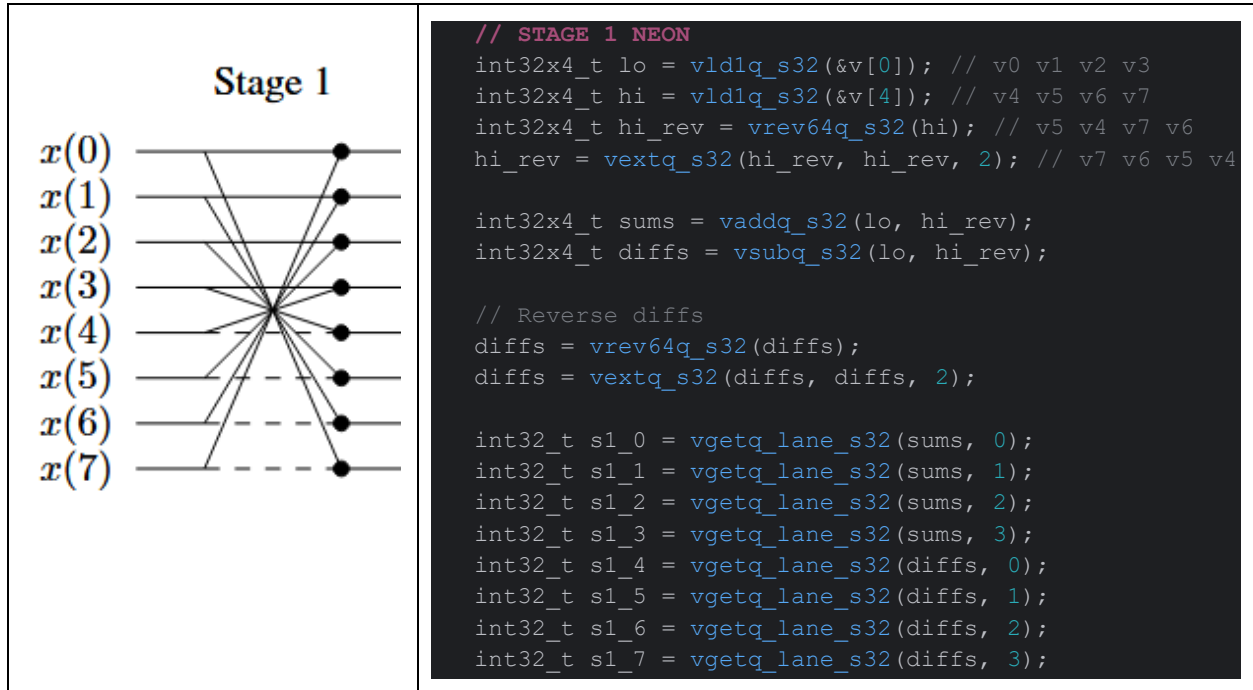
```
#define Q_SCALE 17
#define Q_ONE (1 << Q_SCALE)
#define ROUND_SHIFT_Q(x) ((int32_t)((x) + (1LL << (Q_SCALE-1))) >> Q_SCALE)

// HARDCODED SCALED CONSTANTS
#define C1 128554 // cos(pi/16) << 17
#define C3 108982 // cos(3*pi/16) << 17
#define S1 25570 // sin(pi/16) << 17
#define S3 72820 // sin(3*pi/16) << 17
#define R2C6 70936 // (sqrt(2)*cos(6*pi/16)) << 17
#define R2S6 171254 // (sqrt(2)*sin(6*pi/16)) << 17
#define R2 185364 // sqrt(2) << 17
```

 $O' = k I' \cos \frac{n\pi}{16} + k I'' \sin \frac{n\pi}{16}$ $O'' = -k I' \sin \frac{n\pi}{16} + k I'' \cos \frac{n\pi}{16}$	<pre>// MACRO DEFINITIONS #define ROTATOR(a, b, C, S) do { int64_t _ta = (a), _tb = (b); (a) = ROUND_SHIFT_Q((_ta*(C) + _tb*(S))); (b) = ROUND_SHIFT_Q((-_ta*(S) + _tb*(C))); } while (0)</pre>
---	---

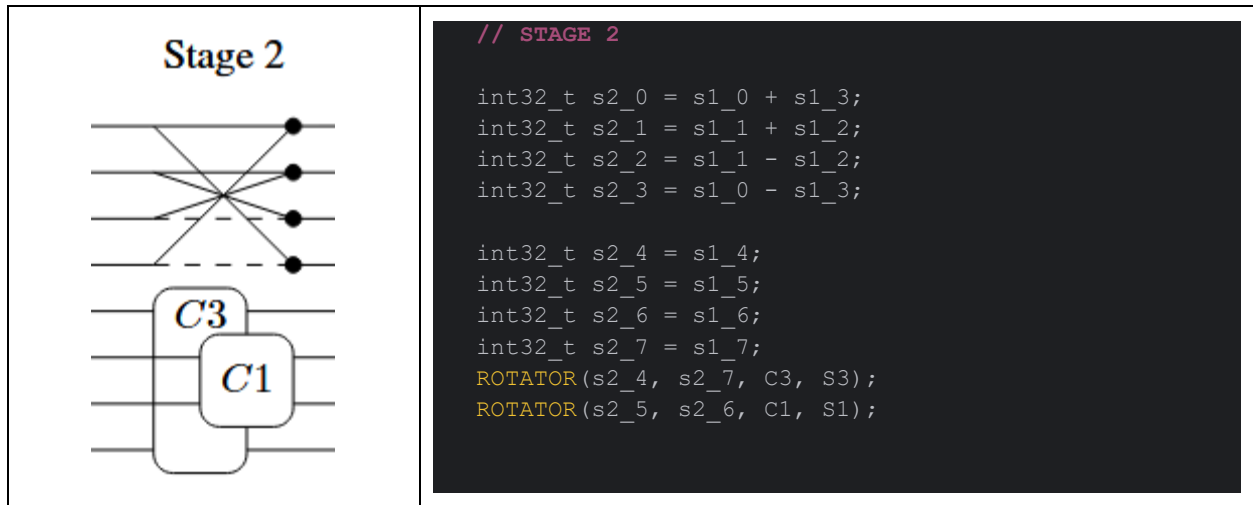
The ROUND_SHIFT_Q macro takes x (a 64-bit integer) in Q_SCALE fixed-point format and converts it back to a signed 32-bit integer with rounding. It first adds a rounding bias of 1 << (Q_SCALE - 1), which ensures round-to-nearest behaviour when the value is right-shifted. It then performs an arithmetic right shift by Q_SCALE bits to remove the fractional portion, and finally casts the result to int32_t.

The second helper function is the rotator macro that is used frequently in the Loeffler function. This rotator takes two input values, a and b, and utilizes the round and shift macro. The parameters C and S are the cosine and sine rotation values, stored as precomputed and scaled constants. Q_SCALE is the number of bits shifted for the fixed-point representation that we want to achieve (17). In this macro, we copy a and b into temporary int64_t variables so that we do not overflow. The resulting values a and b are calculated using the provided C and S constants, then cast to int32_t.

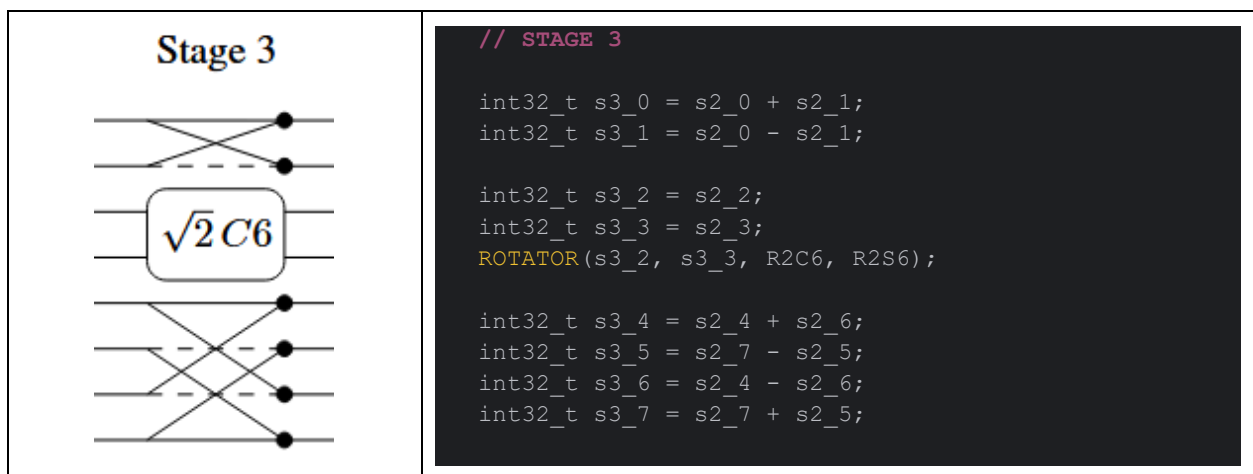


This code snippet represents Stage 1 of the Loeffler DCT algorithm. First, we load the first 4 values into NEON registers lo and hi, then we take the reverse of hi so that we can align the data pairs for use in the reflector operations. Once we have the data prepared, we will calculate the sums and differences, which are in the reflector stage of the operation. After the calculations have been completed, we reverse the diff order so that we can line up the data for the next stages in the algorithm, as well as extract the results of each value into int32_t. Notice that we are optimizing these operations by use of NEON registers, namely:

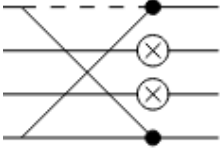
1. vldq_s32: Vector load for 128-bit int and 32-bit signed integers
 - a. Loads 4 consecutive 32-bit ints from memory into a NEON register
2. vrev64q_s32: Vector reverse with 64-bit elements and 32-bit signed integers.
 - a. Reverses the order of pairs of 2 32-bit integers.
3. vextq_s32: Vector extract with 32-bit signed integers.
 - a. Extracted a shifted vector by concatenating two vectors that select a continuous slice starting from an offset.
4. vgetq_lane_s32: Vector get lane 32-bit signed integer.
 - a. Extract the single 32-bit int at a specified index.



This code snippet represents Stage 2 of the Loeffler DCT algorithm. For the even (top) values, we compute the reflector operations as scalars and store them as new variables for use later in the algorithm. Odd values, however, will be redundantly renamed for clarity, then rotated using our defined macro.



This represents Stage 3 in the Loeffler DCT algorithm. The top two values are reflected, the middle values are rotated using the corresponding floating point constants, R2C6 and R2S6, for this stage. Then finally, the odd parts are reflected.

<h3 style="margin: 0;">Stage 4</h3> 	<pre style="margin: 0;">// STAGE 4 int32_t s4_0 = s3_0; int32_t s4_1 = s3_1; int32_t s4_2 = s3_2; int32_t s4_3 = s3_3; int32_t s4_4 = s3_7 - s3_4; int32_t s4_5 = ROUND_SHIFT_Q((int64_t)s3_5 * (int64_t)R2); int32_t s4_6 = ROUND_SHIFT_Q((int64_t)s3_6 * (int64_t)R2); int32_t s4_7 = s3_7 + s3_4;</pre>
---	--

This is Stage 4; we have the even values passing through to be used in the final step. The odd values are being reflected and scaled, where $s4_4$ and $s4_7$ are being reflected, and $s4_5$ and $s4_6$ are being multiplied by the constant $R2$, then rounded and right shifted using the macro `ROUND_SHIFT_Q`. Notice that we cast to `int64_t` to ensure that there are no overflow issues.

<div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> <p>— $X(0)$</p> <p>— $X(4)$</p> <p>— $X(2)$</p> <p>— $X(6)$</p> <p>— $X(7)$</p> <p>— $X(3)$</p> <p>— $X(5)$</p> <p>— $X(1)$</p> </div> <div style="margin-right: 10px;"> <p style="font-size: 3em;">}</p> <p style="font-size: 3em;">}</p> </div> <div> <p>even</p> <p>odd</p> </div> </div>	<pre style="margin: 0;">// FINAL REORDERING v[0] = s4_0; v[1] = s4_7; v[2] = s4_2; v[3] = s4_5; v[4] = s4_1; v[5] = s4_6; v[6] = s4_3; v[7] = s4_4;</pre>
--	--

This last step is augmenting the final order of the lanes to correspond with even and odd outputs, and storing them in our resulting vector.

Compilation and Assembly

Compilation and assembly were done using CMAKE. The file as well as specifications can be seen below.

```
cmake_minimum_required(VERSION 3.12.1)
project(DCT C)

set(CMAKE_C_STANDARD 11)

add_library(dct STATIC
    loeffler_optimized.c
    loeffler_inline.c
    loeffler_macro.c
    loeffler_routine.c
    loeffler_neon.c
    loeffler_register.c
)

set_source_files_properties(loeffler_neon.c loeffler_optimized.c
    PROPERTIES COMPILE_FLAGS "-march=armv7-a -mfpu=neon-vfpv4 -mfloat-abi=hard -O3")

add_executable(DCT main.c images.h)

target_link_libraries(DCT PRIVATE dct)
```

This was developed in JetBrains' CLion IDE with a remote toolchain that builds on root@localhost:2222

Specifications:

- GNU Make 4.2.1
- gcc version 8.2.1 20180801 (Red Hat 8.2.1-2) (GCC)

Flags:

- -march=armv7-a: Specify the target architecture.
- -mfpu=neon-vfpv4: Specifies the Floating Point Unit (FPU) and SIMD extensions to use.
- -mfloat-abi=hard: Specifies the floating-point calling convention to use hard floating point ABI.
- -O3: Optimization level 3, which enables aggressive compiler optimizations for speed.

Build Command:

/usr/bin/cmake --build /tmp/tmp.Ftjp5poh6g/DCT/cmake-build-debug-seng-vm --target DCT -- -j 30

Results

This section outlines the test bench that was created for the project, as well as any helper functions used for result visualization.

```
static double bench_loop(void (*loeffler_function)(int32_t [8]), int iterations) {
    static const int16_t input_raw[8] = {212, 87, 222, 113, 15, 137, 87, 14};
    int32_t v[8], input[8];

    for (int j = 0; j < 8; ++j) // SCALE INPUT VALUES
        input[j] = (int32_t) input_raw[j] << Q_SCALE;

    struct timespec t0, t1;
    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t0);

    for (int i = 0; i < iterations; ++i) {
        // LOOP FOR N ITERATIONS
        for (int j = 0; j < 8; ++j) // RESET INPUT
            v[j] = input[j];
        loeffler_function(v);
    }

    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t1); //RETURN AVG NS PER ITERATION
    return ((t1.tv_sec - t0.tv_sec) * 1e9 + (t1.tv_nsec - t0.tv_nsec)) / iterations;
}

static void benchmark(int runs, int iterations) {
    printf("Run,Routine,Inline,Macro,Register,NeonS1,NeonReflect,NeonFull,Optimized\n");
    for (unsigned run = 0; run < runs; run++) {
        printf("%u", run);
        printf(",%.2f", bench_loop(loeffler_routine, iterations));
        printf(",%.2f", bench_loop(loeffler_inline, iterations));
        printf(",%.2f", bench_loop(loeffler_macro, iterations));
        printf(",%.2f", bench_loop(loeffler_register, iterations));
        printf(",%.2f", bench_loop(loeffler_neon_s1, iterations));
        printf(",%.2f", bench_loop(loeffler_neon_reflect, iterations));
        printf(",%.2f", bench_loop(loeffler_neon_full, iterations));
        printf(",%.2f", bench_loop(loeffler_optimized, iterations));
        putchar('\n');
        fflush(stdout);
    }
}

int main(void) {
    //100 runs of 1,000,000 iterations per benchmark
    benchmark(100, 1000000);
    return 0;
}
```

This code snippet outlines the `bench_loop` function that was used to benchmark the runtime (in nanoseconds) of the Loeffler implementation that was used. For test purposes, we have a dummy 8-value array that will be used as input, as well as the number of iterations to be run to calculate an average run time. For benchmarking purposes, we first scale the raw input data to our Q17 scale and into `int32_t` values for use in fixed-point arithmetic. We get the CPU thread time using `clock_gettime`, then begin looping and applying the chosen Loeffler algorithm. To finish, we get the CPU thread time at the end of the loop of iterations, measure the total time taken divided by the number of iterations, and return that value. We used `CLOCK_THREAD_CPUTIME_ID` rather than `CLOCK_MONOTONIC` as it was far more stable and less variable on the current load of the host machine.

The benchmark function just runs `bench_loop` with every implementation of Loeffler's and prints it to stdout. In our case, we are benchmarking 100 runs with 1,000,000 iterations each. The results of a test bench were:

CPUTIME (ns) per 1D DCT

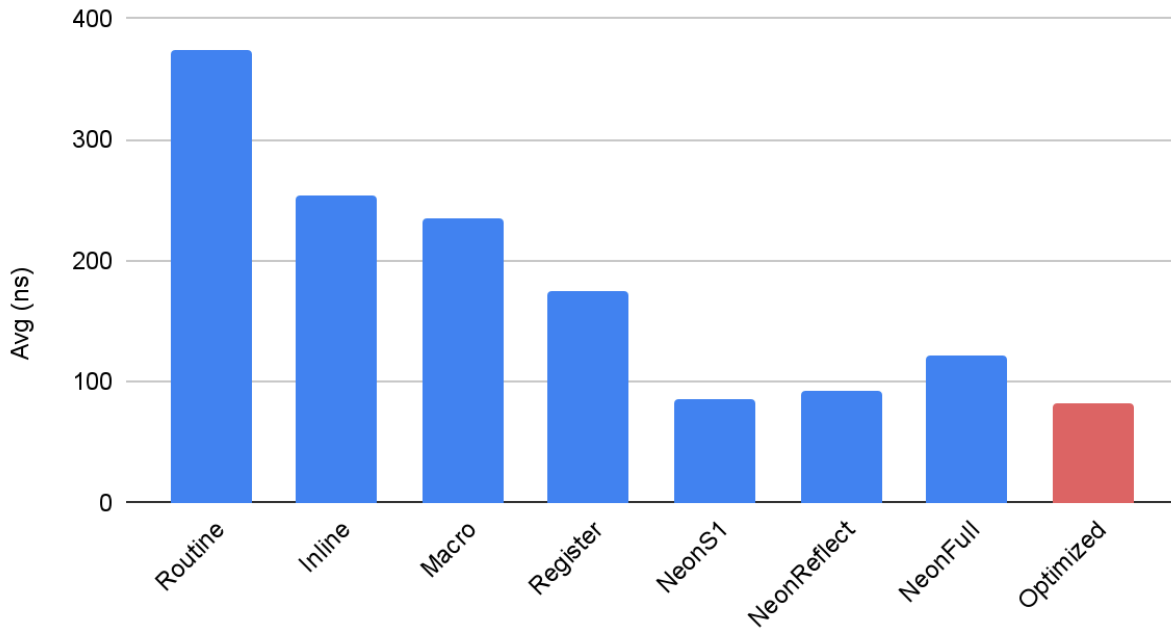


Figure 4: Bar Chart of the Test Bench Results

	Routine	Inline	Macro	Register	NeonS1	NeonReflect	NeonFull	Optimized
Avg (ns)	373.5983	254.3293	234.8328	174.1445	85.3348	91.7535	121.6252	81.4246

Table 1: Test Bench Results in Nanoseconds

As we can see, the fully optimized implementation of Loeffler's DCT had the fastest result time, whereas the routine implementation had the slowest. The inline, macro, and register implementations were a good initial start to optimizing the implementation, but when incorporating NEON into our implementations, that's when we saw the most improvement to the running time of the implementations.

To visualize our final DCT implementation, test photos were obtained, and processing was completed:



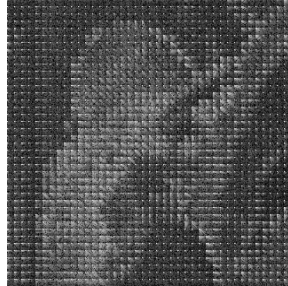



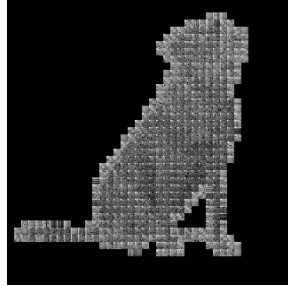



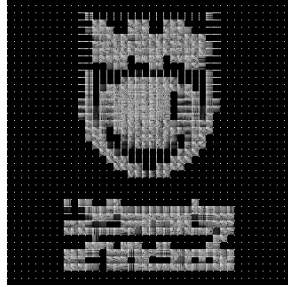

Original	Greyscale and Scaled	DCT Log Visualized	CV2 Reverted
			
			
 University of Victoria	 University of Victoria	 University of Victoria	 University of Victoria

Table 2: Original Photo, Pre-Processed, DCT Applied, Reverted.

These are the results for 3 input test images, where the first column represents the untouched original image, the second column is the grayscale and scaled image ready for DCT, the third column is a log-scaled visualization of the result of running the optimized 2D DCT implementation, then finally, the reverted image that effectively reconstructs the preprocessed image used as input. The processing of these visualizations will be discussed below.


```

from pathlib import Path
from PIL import Image

src = Path("./lenna.png")
name = src.stem
img = Image.open(src)

# Resize and convert to greyscale
img = img.resize((320, 320)).convert("L")
w, h = img.size

# Save image to header for c DCT
with open(f"{name}.h", "w") as f:
    f.write(f"const uint8_t "
           f"{name.upper()}[{w * h}] = {{\n")
    for i, px in enumerate(img.getdata(), 1):
        f.write(f"{px},")
        if i % 16 == 0:
            f.write("\n")

    f.write("};\n")

print(f"Saved header file to {name}.h")

```

```

#ifndef IMAGES_H
#define IMAGES_H

#define IMAGE_WIDTH 320
#define IMAGE_HEIGHT 320

const uint8_t DOG[102400] = {...}
const uint8_t LENNA[102400] = {...}
const uint8_t UVIC[102400] = {...}

#endif //IMAGES_H

```

This Python program prepares the images for use with our DCT implementation in C. Namely, it resizes the images to a consistent size of 320x320 and converts them to grayscale, and finally converts the image into a uint8_t array and writes it to a C header file for use with our Loeffler's algorithm.

```

// RUN 2D DCT WITH PROVIDED LOEFFLER FUNC
void dct_block(int32_t block[8][8], void (*loeffler_func)(int32_t [8])) {
    // 1D DCT ON ROWS
    for (int r = 0; r < 8; ++r) {
        loeffler_func(block[r]);
    }

    // 1D DCT ON COLS
    for (int c = 0; c < 8; ++c) {
        int32_t col[8];

        // CREATE COL
        for (int r = 0; r < 8; ++r) {
            col[r] = block[r][c];
        }

        loeffler_func(col);

        // WRITE COL BACK
        for (int r = 0; r < 8; ++r) {
            block[r][c] = col[r];
        }
    }
}

```

```

// CSV IS EASY TO COPY AND PASTE FROM STDOUT
static void write_csv(const int32_t img[IMAGE_HEIGHT][IMAGE_WIDTH]) {
    for (int r = 0; r < IMAGE_HEIGHT; ++r) {
        for (int c = 0; c < IMAGE_WIDTH; ++c) {
            if (c) putchar(',');
            printf("%d", img[r][c]);
        }
        putchar('\n');
    }
}

void dct_image(const uint8_t *img, void (*loeffler_func)(int32_t [8])) {
    static int32_t dct_out[IMAGE_HEIGHT][IMAGE_WIDTH];

    for (int r = 0; r < IMAGE_HEIGHT; r += 8)
        for (int c = 0; c < IMAGE_WIDTH; c += 8) {
            int32_t block[8][8];

            // PREP BLOCK FOR DCT
            for (int i = 0; i < 8; ++i)
                for (int j = 0; j < 8; ++j)
                    block[i][j] = img[(r + i) * IMAGE_WIDTH + (c + j)] << Q_SCALE;

            // 2D DCT
            dct_block(block, loeffler_func);

            // WRITE BLOCK BACK
            for (int i = 0; i < 8; ++i)
                for (int j = 0; j < 8; ++j)
                    dct_out[r + i][c + j] = ROUND_SHIFT_Q(block[i][j]);
        }

    write_csv(dct_out);
}

int main(void) {
    dct_image(LENNA, loeffler_optimized);
    return 0;
}

```

The previous code block corresponds to our image transformer that utilized our implemented `loeffler_func`. This returns the DCT processed image in CSV format as STDOUT. We can copy and paste to a CSV file for processing in helper Python scripts. This includes the functions `writescsv`, `dct_block`, and `dct_image`. The `dct_block` function will run a 2D DCT on the 8x8 input block. It will first run DCT on each row of the block, then each column, writing the result back into the input block. The `dct_image` function utilizes the `dct_block` function, splitting the input image into 8x8 blocks, scaling to use the fixed point format, applying the `dct_block` function, rounding results, then utilizing the `write_csv` to write to an output file.

Improvements

Further performance improvements could have potentially been obtained if we had spent more time optimizing the register handling in the full NEON implementation. We attempted to perform vector operations on every single stage and operation of Loeffler’s algorithm, but to our surprise, we found that its usage past stage 1 had a negative performance impact. This is likely due to the number of intermediate memory instructions required to reorder the data within the NEON registers so that they are positioned correctly for the next butterfly operation. Now that we have finished our optimized implementations, with hindsight, we believe that more performance could potentially have been extracted had we handled the NEON registers with more intentionality.

Rather than defining only four NEON registers for input low, input high, sums, and diffs respectively, we could have leveraged more of the 16 available 128-bit Q registers on an ARMv7 system. For example, we could define two independent NEON Q registers per stage, such that stage0_0 and stage0_1 would contain the low and high input values, respectively, with sums stored in stage1_0 and diffs in stage1_1.

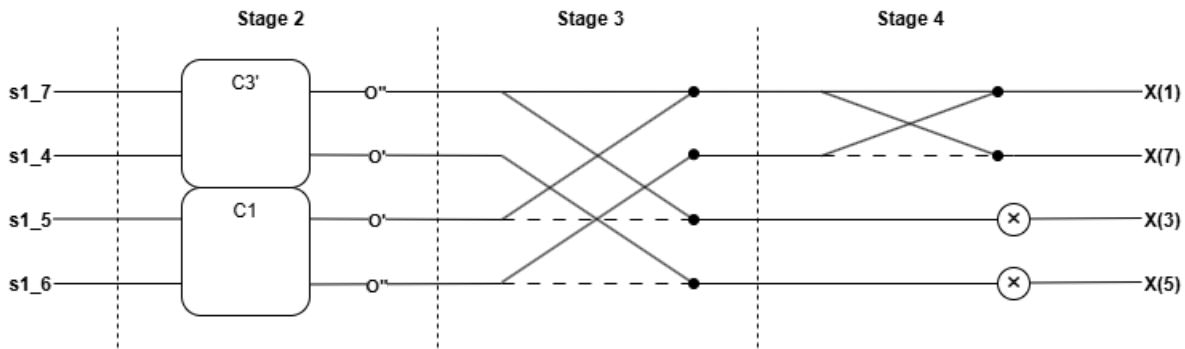


Figure 5: Potentially Improved Dataflow for NEON Utilization

A significant performance detractor was the number of reordering operations needed to move data from one NEON register into another with the correct indices for the subsequent operation. A future implementation could make more deliberate use of the NEON register file, with a potentially improved data flow, as shown in Figure 5. This approach applies a one-index roll to all higher-index lanes after stage 1, so that reflector operations are stacked rather than “overlapping”. This would allow the creation of a single vector reflector function capable of handling both the C1/C3 reflections in stage 2 and the $\sqrt{2}C6$ operation in stage 1 without requiring structural changes to the function, while also simplifying stages 3 and 4. Instead of relying on memory operations and temporary registers to reorder NEON register contents, further emphasis could be placed on using `vrev64q_s32` and `vextq_s32` to flip and reorder data directly within the already-populated registers.

Another small improvement we could have made would be to define the rotator constants at an even higher precision within a 64 bit integer, then scale down to match our chosen Q scale. This step would allow us to change our Q value without having to manually recompute all constants, while still satisfying the integer arithmetic restriction.

Conclusions

The results of this project showed the fully implemented DCT using Loeffler's Algorithm emulated on an ARM Cortex-A15 platform. We developed and compared different implementation approaches, namely using routines, inline functions, macros, registers, as well as NEON, in multiple areas, culminating in a final fully optimized version of the DCT algorithm that performs well. We created a test bench so we could compare the different optimizations, and we gained knowledge on the different implementations and their respective positives and negatives. We had a speed improvement of 78.2% from our first implementation to the final optimized one. Key techniques included NEON SIMD vectorization and fixed-point arithmetic. Overall, this project was a valuable learning experience in embedded systems performance tuning and the optimization strategies required to make such systems run efficiently.

References

- [1] "Fundamentals of NEON technology," Arm Developer, accessed Aug. 9, 2025. [Online]. Available: <https://developer.arm.com/documentation/den0018/a/Introduction/Fundamentals-of-NEON-technology>
- [2] `clock_gettime`, Linux manual page, section 3. [Online]. Available: linux.die.net/man/3/clock_gettime
- [3] "DCT (Discrete Cosine Transform)", A Security Site. [Online]. Available: asecuritysite.com/comms/dct2
- [4] OpenCV Documentation — Operations on arrays, OpenCV Core module, reshape function. [Online]. Available: docs.opencv.org/4.x/d2/de8/group__core__array.html#ga77b168d84e564c50228b69730a227ef2
- [5] `scipy.fftpack.dct`, SciPy v1.16.1 Manual. [Online]. Available: docs.scipy.org/doc/scipy/reference/generated/scipy.fftpack.dct.html