12-1-2012

A Study of the use of SIMD instructions for two image processing algorithms

Eric Welch

A Study of the use of SIMD Instructions for Two Image Processing Algorithms

by

Eric Michael Welch

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

In

Electrical Engineering

Approved By:

Dr. Dorin Patru
Associate Professor, Department of Electrical and Microelectronic Engineering
Thesis Advisor

Dr. Eli Saber
Professor, Department of Electrical and Microelectronic Engineering

---

Dr. Gill Tsouri
Assistant Professor, Department of Electrical and Microelectronic Engineering

---

Dr. Sohail Dianat
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
December 2012

Dedication

Dedicated to my family for always supporting and encouraging me.

Acknowledgements

I would like to thank my advisor Dr. Dorin Patru for his guidance and support; my committee members Dr. Eli Saber and Dr. Gill Tsouri for their input on the project; and HP liaison Kurt Bengtson for providing this opportunity.

Abstract

Many media processing algorithms suffer from long execution times, which are most often not acceptable from an end user point of view. Recently, this problem has been exacerbated because media has higher resolution. One possible solution is through the use of *Single Instruction Multiple Data* (SIMD) architectures, such as ARM's NEON. These architectures take advantage of the parallelism in media processing algorithms by operating on multiple pieces of data with just one instruction. SIMD instructions can significantly decrease the execution time of the algorithm, but require more time to implement.

This thesis studies the use of SIMD instructions on a Cortex-A8 processor with NEON SIMD coprocessor. Both image processing algorithms, bilinear interpolation and distortion, are altered to process multiple pixels or colors simultaneously using the NEON coprocessor's instruction set. The distortion algorithm is also altered at the assembly level through the removal of memory accesses and branches, adding data prefetch instructions, and interlacing ARM and NEON instructions. Altering the assembly code requires a deeper understanding of the code and more time, but allows for more control and higher speedups. The theoretical speedup for the bilinear interpolation and distortion algorithms is three and four times respectively. The actual measured speedup for the bilinear interpolation algorithm is more than two times, and for the distortion algorithm is more than three times. The results show that SIMD instructions can provide a speedup to image processing algorithms following a correct sequence of modifications of the code.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

The time required to process media, such as images and audio, has become increasingly longer over the past few years due to the increase in resolution. The speed of computing processors has not kept up with the time required to process images. One solution to this problem is the implementation of *Single Instruction Multiple Data* (SIMD) instruction sets. The SIMD instructions operate on multiple data with just one instruction. Instructions can be applied to data sets of four or more operands simultaneously. SIMD architectures, such as Intel's WMMX and SSE and ARM's NEON, can exploit the parallelism present in many image processing algorithms by operating on multiple pixels at a time. This can significantly increase the speed of algorithms by a factor of two or more, but additional time is required to implement the instructions.

An ARM processor is used in many embedded applications such as cellular phones, televisions, and printers. An ARM processor is a 32-bit *Reduced Instruction Set Computer* (RISC) with a load/store architecture. The processor's architecture is licensed from ARM and implemented by manufacturers such as Texas Instruments, Marvell, and others. The manufacturers implement the architecture, add custom components, and manufacture the processor. Advantages of an ARM processor include a simple unified design and low power consumption. The unified design allows programmers to easily change from one processor manufacturer to another without learning a new instruction set. The ARM processors aim to be high performance with low power consumption. The low power consumption is ideal for mobile devices, which often have a limited supply of battery power.

Recently, ARM processors have included two SIMD options, ARMv6 SIMD and NEON SIMD. The ARMv6 SIMD is included in the ARMv6 architecture and above. These SIMD instructions operate on the traditional 32-bit ARM registers, and can process up to four 8-bit operands at a time. The ARMv7 architecture introduced the NEON SIMD coprocessor in the Cortex-A8. This coprocessor is separate from the ARM processor and can process up to sixteen 8-bit operands at a time. The NEON coprocessor contains four times the capacity of the ARMv6 SIMD, which can increase the speedup even more.

Combining the ARM processor with the NEON SIMD coprocessor is ideal for embedded systems. Most embedded systems, such as cellular phones and printers, perform large amounts of media and data processing. In most cases, the user requires this processing to occur quickly, which is possible with SIMD instructions. Because most embedded systems already include an ARM based processor, changing to an ARM based processor with NEON coprocessor is trivial. The hardware may have to be altered slightly, but the software can remain mostly the same. The only major change is rewriting the code to include the SIMD instructions, which can be time consuming. The main drawback of using SIMD instructions is the increased development time.

Previous studies on the use of SIMD instructions produced a speedup of less than three times. This thesis demonstrates how a speedup of greater than three times can be attained using SIMDs and other optimization techniques. The remainder of this thesis focuses on the implementation of NEON SIMD instructions on a bilinear interpolation algorithm and a distortion algorithm. The remaining chapters are organized as follows: Chapter 2 describes SIMD instructions, the NEON instruction set, and previous works related to SIMD image processing. Chapter 3 describes the hardware and software setup used, and the two algorithms used for testing. Chapter 4 presents the various test cases for both algorithms. Chapter 5 presents and discusses the results from all the test cases. Chapter 6 concludes the thesis with concluding remarks, and ideas for possible future work.

Chapter 2
Background

In the past few years, more emphasis has been placed on multimedia processing in computers. Image and audio files have become higher resolution, which requires more processing time than lower resolution files. To counteract the increased processing time, *single instruction multiple data* (SIMD) instruction set extensions have been developed to process more data during each instruction cycle. Section 2.1 explains the SIMD instructions, followed by section 2.2 which explains the NEON SIMD instructions from ARM, and finally section 2.3 explains how SIMD instructions can be specifically applied to image processing.

2.1 SIMD Instruction Set Extensions

SIMD instruction set extensions have become more popular over the years, and are being included in most current computer processors. Each SIMD instruction processes multiple data during its execution. The SIMD architecture can be implemented in two ways, modifications to the main processor or the addition of a coprocessor. The former uses the main processor's 32 or 64 bit registers with small modifications to the functional units. The latter adds an additional coprocessor with separate larger 128 or 256 bit registers and functional units. When operating on the main processor's registers, very little additional hardware is needed for implementation. Using a coprocessor architecture requires larger registers and larger functional units, which adds additional hardware and complexity to the design. However, each instruction is able to process more data compared to the main processor architecture.

SIMD registers are divided into multiple lanes of 8 bits to 32 bits. Because most multimedia processing occurs with either 8 or 16 bit operands, up to 32 operands can be processed at a time with 256-bit registers . Figure 2.1 shows an example of addition using 32-bit registers divided into four lanes of 8 bits. Each individual lane of register A is added to each individual lane of register B to form the result in register C. Normally, this addition would require four instructions and four cycles to complete, but the SIMD addition requires one instruction and would most likely be completed in one cycle. This is

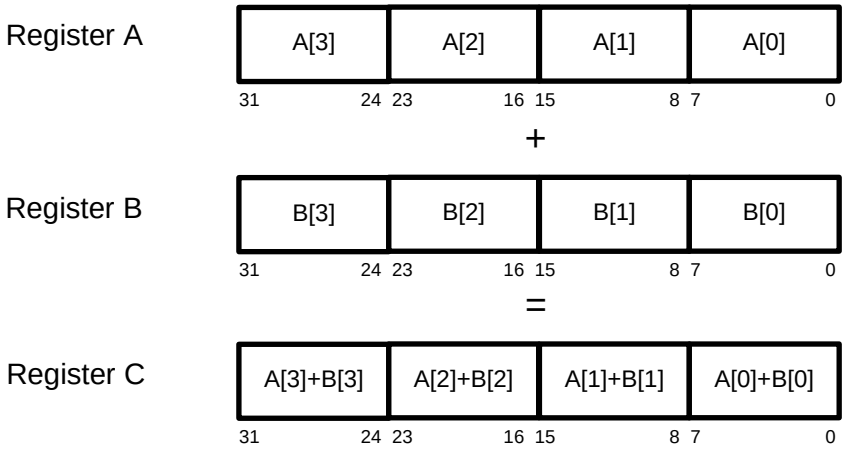


Figure 2.1: Example of SIMD Addition

a speedup of four times, which is fairly significant if this operation is occurring in a large loop.

Using SIMD instructions is typically more time consuming than writing non-SIMD code. For example, one has to ensure that each operand is in the correct lane and enough space is available to complete the operation. SIMD libraries or coding in assembly is often the best way to use the instructions. The libraries have functions that compile into SIMD instructions, which make writing the code easier. This allows the programmer to modify just the parts of the C code that need to be parallelized. To achieve best performance, SIMD instructions should be written at the assembly level. At this level, one has more control over what operands are in each register and can better optimize for performance. Since writing assembly code is even more time consuming and difficult, it is often done only when high performance is needed. Increasingly, compilers are able to vectorize loops and code SIMD instructions directly. Vectorizing a loop involves removing loop iterations with the use of SIMD instructions. The vectorizing compilers are still being developed and currently only vectorize about half of the possible loops .

SIMD instructions can significantly decrease the processing time of programs which are parallelizable. Although, speedups of four or eight times are theoretically possible, practically these will be less. Overhead involved with using the instructions as well as non-vectorizable parts of the code will cause the speedup to be less than theoretical. The benefits of using SIMD instructions come with a cost. More time will be needed to implement these programs and the programmer will have to be more aware of how the

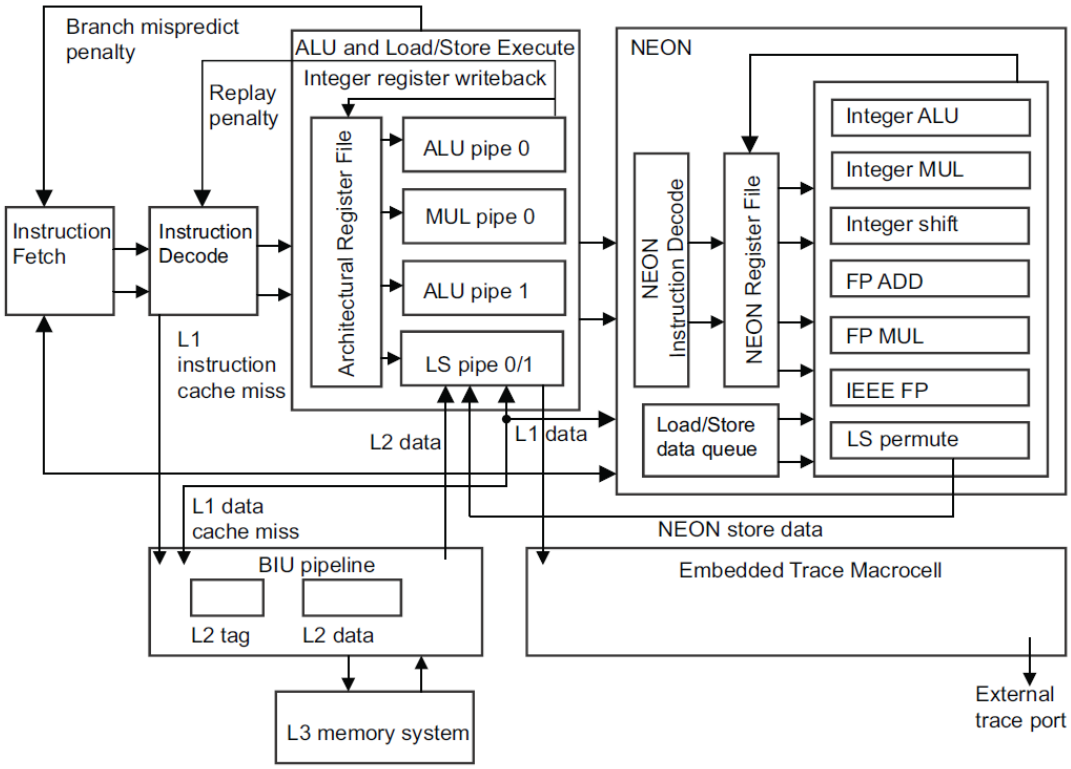


Figure 2.2: ARM and NEON Pipeline for the Cortex-A8 [4]

operations can be executed in parallel. For some applications, the cost may outweigh the benefit, but for others this potential speedup is critical for the success of the program.

2.2 NEON SIMD Architecture

Many different SIMD architectures have been developed by different companies for use in their processors. ARM processors implement the NEON SIMD architecture, which consists of a coprocessor that is included in all Cortex-A8 processors and optional in Cortex-A9 processors . The full Cortex-A8 ARM and NEON pipeline is shown in Figure 2.2. The ARM processor fetches SIMD instructions from the L1 instruction cache, and forwards them to the NEON coprocessor, which then decodes and executes the instructions. The coprocessor contains an integer *Arithmetic Logic Unit* (ALU), multiply unit, shift unit, and a floating point addition and multiply unit. The coprocessor and processor's pipelines are 13 stages deep and all the functional units are pipelined to allow the execution of multiple instructions at a time. The NEON coprocessor has the

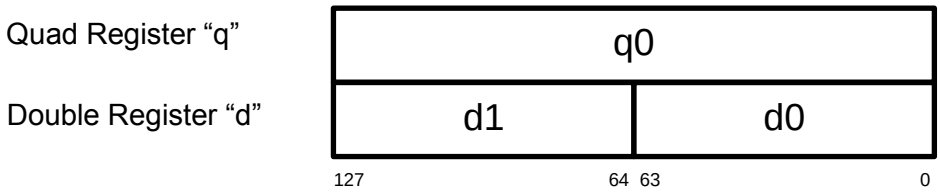


Figure 2.3: Partitioning of Quad Registers into Double Registers

Table 2.1: Instruction Cycle Timing

Instruction Type	Instruction Example	Number of Cycles
ALU	AND, SUB, MOV, ADD	1
Multiply	MUL, MLA	2
Load/Store	LDR, STR	1
NEON ALU	VADD, VAND, VSUB	1
NEON Multiply	VMUL, VMLA	4
NEON Load/Store	VLD1, VST1	2
NEON Conversion	VCVT	2

ability to access the data in either the L1 data cache or L2 cache. It also has a separate register file from the ARM processor consisting of either 32 64-bit registers or 16 128-bit registers. The 128-bit quad registers are partitioned in half to create two 64-bit double registers as shown in Figure 2.3. The quad registers are labeled as q0 through q15, and the double registers are labeled as d0 through d31. These registers can be split into lanes consisting of 8, 16, 32, or 64 bits, and contain signed or unsigned integers, floating-point numbers, or polynomials .

Coding using NEON SIMD instructions must be done to fully utilize the processor and avoid hazards which can cause stalls. Table 2.1 shows most instructions, with the exception of multiplication, complete in one cycle. Also, the functional units are pipelined, therefore structural and data hazards do not occur very often. However, stalls can occur when moving data from the coprocessor to the ARM processor, or when the ARM and NEON load/store units access the same cache line. The former will cause a stall of 20 cycles for both the ARM and NEON pipelines. The latter can cause a stall of up to 20 cycles to handle cache ordering issues. The processor also has the option to dual issue instructions. This involves issuing two instructions in the same cycle, but one of the instructions must be either a load/store or a data move between processor and

coprocessor registers. Because of the large strides SIMD instructions take when processing data, the NEON coprocessor also has access directly to the L2 cache. If an L2 cache miss occurs from the NEON pipeline, then the main memory will be accessed and only the L2 cache will be filled .

2.3 Image Processing using SIMD instructions

Image processing speed can be significantly increased using SIMD instructions. Most images contain many pixels which are sequentially stored in memory. Each pixel consists of one or more 8-bit integer values which describe the intensity of the color(s) in the image. There is one color channel for black and white images and usually three color channels for color images. With non-SIMD image processing, the 8 bits only fill a quarter of the standard 32-bit register. Any operations on this register work on the full 32 bits, and therefore, some of the processing is done on the unneeded 24 bits. Many image algorithms are linear, and thus, the result from one pixel calculation does not affect other pixels . SIMD takes advantage of this parallelism by placing multiple sequential pixels into one register, and processing occurs on these pixels concurrently.

Theoretically, SIMD instructions could produce a speedup factor of four to eight times when used with image processing . They have already been shown to provide speedups of 1.25 to 2 times in video processing algorithms , . This is significantly below the theoretical four times speedup, but it is still fairly significant for some algorithms. Speeding up algorithms can also effect power consumption. If the processor finishes the task much sooner, then it will have more time to go into low power mode and thus decrease power consumption. Also, if a processor and coprocessor are concurrently active, then the energy consumption may increase during that time. Speeding up any algorithm could significantly affect the end user with faster processing and decreased power consumption.

Intel's SIMD instructions are known as *Streaming SIMD Extensions* (SSE), and they operate in a similar way to ARM's NEON. These instructions can be used for image and digital signal processing in Intel's processors. The SSE architecture replaced the MMX architecture and includes eight 128-bit registers for integer or floating-point numbers. One study used the SSE instructions to speed up the algorithms for a sepia filter and crossfade filter. The former converts an image to sepia tone, and the latter fades together two separate images. Because the filters work on uncorrelated pixels, the processing can happen on multiple pixels at a time. The algorithms processes four pixels

Table 2.2: Results from Intel SSE Study

Filter	Integer Speedup	Floating-point Speedup
Sepia	2.6	1.9
Crossfade	2.7	1.9

per iteration using SEE, and therefore, the theoretical speedup is four. Table 2.2 shows SIMD extensions provide an actual speedup of about 2.6 to 2.7 times for an integer only approach with the sepia filter depending on the resolution. The crossfade filter algorithm produced a speedup of about 1.9 times depending on the resolution.

These studies prove that SIMD extensions can increase the performance of image and video processing algorithms depending on the image size, although the actual speedup so far is much lower than the theoretical speedup.

Chapter 3
Description of System

The use of SIMD instructions is tested on an ARM processor containing a NEON coprocessor with two image processing algorithms. Section 3.1 and 3.2 describes the hardware and software setups, respectively, used for testing. Section 3.3 describes the bilinear interpolation and distortion algorithms used in implementing the SIMD instructions.

3.1 Hardware Setup

A BeagleBone prototyping board from beagleboard.org was chosen because of its use of a Texas Instruments AM3359 Cortex-A8 processor. The BeagleBone board can directly connect to the host PC using a standard USB-A to USB-mini connector or via an optional JTAG connector. The USB client allows *Secure Shell* (SSH) terminal access and *SSH File Transfer Protocol* (SFTP) file transfer between the host PC and BeagleBone board. The board contains 256 MB of random access memory and a 2 GB microSD card, which provides plenty of memory for image processing. The microSD card comes preloaded with the Angstrom distribution of the Linux kernel version 3.2.14. The kernel allows programs to be easily compiled and provides easy file manipulations. The board also includes Ethernet and USB host ports, which allows for file transfer and installation of new packages .

The AM3359 processor runs at 500 Mhz when powered via USB and 720 Mhz when powered by an external power supply. The processor includes 32 KB each of L1 instruction cache and L1 data cache, and 256 KB of L2 cache. The L1 and L2 caches are 4-way and 8-way set associative, respectively, and have a line size of 64 bytes. The L2 cache has a 128-bit interface to the main memory, which corresponds to the size of the NEON registers. The processor's bootloader is stored 176 KB ROM, and is used to start the Linux kernel .

The Cortex-A8 is built on the ARMv7 RISC architecture, which includes 14 general purpose registers, one link register, one *program counter* (PC) register, and one *Current Program Status Register* (CSPR). The general purpose registers can hold any data or address for computation. The link register contains the return address when a branch

with link instruction is performed, or can be used as a general purpose register. When returning from a branch, the value from the link register is loaded into the PC register. The PC contains the address of the instruction to be issued next to the processor. The CSPR contains condition flags, such as overflow and carry, and the current mode of the processor. The architecture can execute either ARM or Thumb instructions. The former is the standard 32-bit instruction set included on ARM processors, and the latter is a compressed 16-bit instruction set, which allows more compact code to be compiled.

The Cortex-A8 includes program flow prediction, NEON advanced SIMD coprocessor, *vector floating point* (VFP) coprocessor, dual issue pipeline, and four performance counters. Program flow prediction is used to help avoid branch misses, and includes a 512-entry 2-way set associative branch target buffer. Each branch miss incurs a 13-cycle penalty because the pipeline must be flushed. Therefore, branch misses must be kept to a minimum. The NEON SIMD instructions were discussed previously in section 2.2. The VFP coprocessor is a floating point architecture that allows for fast floating point number operations. The VFP uses the same registers as the NEON coprocessor and supports either single or double precision floating point numbers. The dual issue pipeline allows a load or store instruction to be issued with another instruction providing no data, structural, or control hazards occur. Dual issuing can save many cycles and make load and store instructions less costly to perform. The performance counters are used to measure events triggered by the processor including branch predictions, cache accesses and misses, and stalls incurred by full instruction queues or data transfers . By default, the counters are not enabled on the BeagleBone board and must be enabled in the kernel or via a kernel module.

The BeagleBone prototyping board can measure current and power consumption in two ways. The first method is using the on chip current measurement setup as described in the BeagleBone *System Reference Manual* (SRM) . This uses an analog input to the processor to measure the voltage drop over a 0.1 ohm resistor. From this voltage and the resistor value, the power consumption of the board can be measured. The second method is to directly measure the current into the board using a 5 volt power supply. Based on the current and power supply voltage, the power consumption can be measured. The on chip method is preferred because the program can set checkpoints throughout execution to record the current. This can be used to see how the board's power consumption changes throughout the different stages of the program. According to the SRM the board's current should be between 170 mA and 350 mA.

3.2 Software Setup

The ARM Development Studio 5 (DS-5) was chosen for the IDE. DS-5 contains the GNU compiler version 4.5.1 and the ARM compiler version 5.01 for the ARM Linux kernel. The compilers enable programs to be compiled on the host PC and run on the board under the Linux kernel. The GNU version of the compiler was chosen because of its superior optimizations, including automatic vectorization, and open-source nature. DS-5 also contains a debugger, which is compatible with the BeagleBone. This allows stepping through a program, providing the location of errors, and inspection of the ARM and NEON register files. The IDE also contains support for SFTP, which is used to transfer the program and input data to the board and retrieve the output data, including resulting image and performance results.

The GNU compiler is an open source compiler which can compile programs for use on ARM-Linux kernel. This compiler includes many advanced optimizations including function inlining, loop unrolling, instruction reordering, and automatic vectorization. The compiler also supports intrinsic functions for NEON SIMD. These functions can be called directly from C and will compile into NEON assembly instructions. Built-in functions are also included to provide hints about program execution to the compiler. The hints can include what data will be accessed next so the compiler can preload the cache or can include the likely direction a branch will take .

The processor's performance counters must be enabled from software within the kernel or in a kernel module to allow profiling of programs. The counters are located in coprocessor 15, the system coprocessor, which contains registers that have information about the processor's configuration. The kernel had to be recompiled to allow a kernel module to be built. The Linux kernel version 3.2.23 was compiled with the PROFILING, FTRACE, ENABLE_DEFAULT_TRACERS, and HIGH_RES_TIMERS options enabled to allow the profiling. The kernel module is used to enable user mode access to the performance counters by setting the USEREN register . After user mode access is given, the counters are interfaced with the *perf.cpp* file shown in Appendix A. This file initializes the counters and output file using inline assembly. The code starts the counters with the *perf_init* function, allows checkpoints throughout execution with the *perf_checkpoint* function, and stops the counters and closes the file with the *perf_exit* function. The *perf_init* function receives the values for the performance metrics under investigation from the command line input when executing the code. The counter is selected with the PMNXSEL register, the metric's value is set via the EVTSEL register,

and the counters are enabled using the PMNC register. Also, the output file “perf.csv” is opened, and the start time is recorded. The *perf_checkpoint* function receives the name for the checkpoint and whether this checkpoint is valid. This function selects the counter with the PMNXSEL register, and reads the performance metrics from the PMCNT register. The output file is written with the checkpoint name, counter values, time the checkpoint is called, and the value of the counter overflow register, FLAG. The FLAG register will report overflow if the counters exceed the 32-bit dimension. The *perf_*exit function is called at the end of the program to stop and write the final values of the counters, and close the file containing the results.

3.3 Algorithms under Investigation

Two algorithms are selected to test the NEON SIMD instructions. Both algorithms are used in image processing, and because they are linear, the processing can be accomplished in parallel. Section 3.3.1 and section 3.3.2 describes the bilinear interpolation and distortion algorithms, respectively.

3.3.1 Bilinear Interpolation Algorithm

Bilinear interpolation algorithms are used frequently in image processing. The purpose of bilinear interpolation is to either enlarge or shrink an image to a specified dimension. When an image is enlarged, the algorithm will attempt to fill in the missing data by averaging the surrounding pixels. Figure 3.1 shows an example 3x3 image which is interpolated to a 5x5 image. The algorithm takes the original image and expands it on the interpolated image (shown in grey). This process leaves space between the pixels (shown in white). This space is filled in by averaging the pixels around it. For example, four pixels surrounding the three in the interpolated image are one, two, four, and five. These four values are added together and divided by four to calculate the new value. At the sides of the image, the interpolation may occur with less than four values. After the averaging, fractional numbers are left. Because fractions cannot be values for pixels, the values must be rounded to the nearest integer. This type of algorithm uses a lot of floating point operations which is slower than integer operations in most processors. For performance reasons, an integer-only algorithm is chosen for testing.

The algorithm chosen was written by Etienne Sobole and the modified code is shown in the in Appendix B, and will be used as the baseline for comparison. An example input and output image is shown in Figure 3.2. This image was interpolated by

Original Image

1	2	3
4	5	6
7	8	9

Interpolated Image

1	1.5	2	2.5	3
2.5	3	3.5	4	4.5
4	4.5	5	5.5	6
5.5	6	6.5	7	7.5
7	7.5	8	8.5	9

Rounded Interpolated Image

1	2	2	3	3
3	3	4	4	5
4	5	5	6	6
6	6	7	7	8
7	8	8	9	9

Figure 3.1: Bilinear Interpolation Example

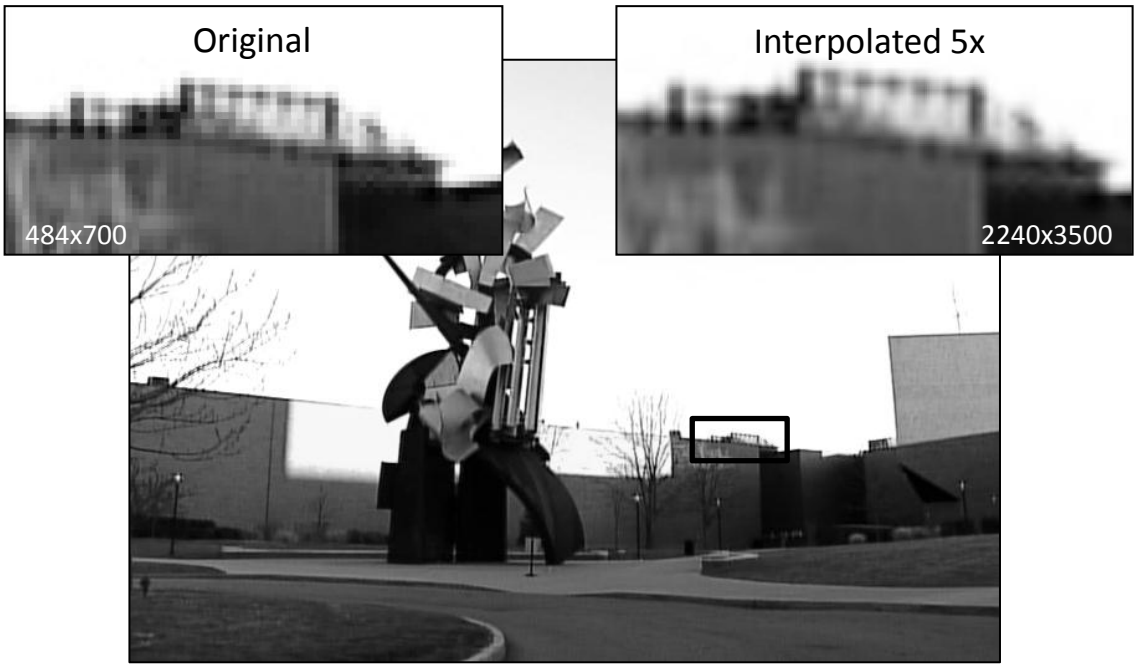


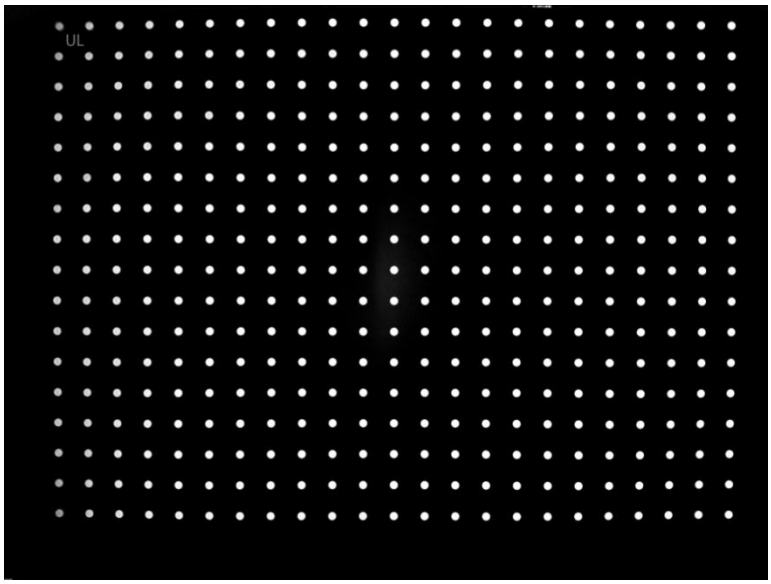
Figure 3.2: Example Image Interpolated by a Factor of 5

a factor of five from 484x700 pixels to 2240x3500 pixels. The interpolated image appears less blocky and smoother between the transitions from one object to another. This algorithm enlarges an image to a specified dimension, but cannot shrink the image. Also, there are no floating point operations, and the processing occurs in one pass. This helps increase performance because, when compared to integer, floating point manipulations usually take more time. Also, processing in one pass causes the destination image to be stored in memory just once, and this helps reduce the latency caused by cache accesses. The algorithm assumes that the color channels are stored as a 32-bit value, and all three color channels are contained in the lower 24 bits. The code starts by first determining the step through the source image as a 16-bit number. Next, it loops through the destination image starting in the x-direction. In the inner loop, the four surrounding pixels are retrieved from the source image. The destination pixel is calculated based on these four values with each color channel being processed separately. The result is written back to memory and the process is repeated for the remaining pixels. The only change from the original algorithm was moving from four color channels to three color channels. With only integer calculations and few loops, this baseline algorithm has very high performance.

3.3.2 Distortion Algorithm

The distortion algorithm was developed by HP and is used as the baseline for comparison. This algorithm removes the perceived distortion from a captured image. The program accepts an image as a *.dat file, created by MATLAB, and contains the 8-bit raw pixel information for the source image. The *.dat file is divided into thirds, where each third corresponds to a color channel. The program also accepts a distortion matrix input, which is smaller than the input image and contains multiple 2-D vectors. The vectors are used to map the pixels from the source image to the destination image. This matrix is a floating point matrix, but is converted to integer representation to aid in increasing the performance. Figure 3.3 shows an original image and the image after the algorithm was applied. The results are very subtle, but it can be seen that the white dots in the source image are not perfectly aligned and have a slight convex curve to them. The processing works by moving and interpolating the pixels so these dots appear more aligned. Normally, this algorithm takes a few seconds to process. When combined with others, the processing of an image can take tens of seconds, which is too high for the

Original Image



Processed Image

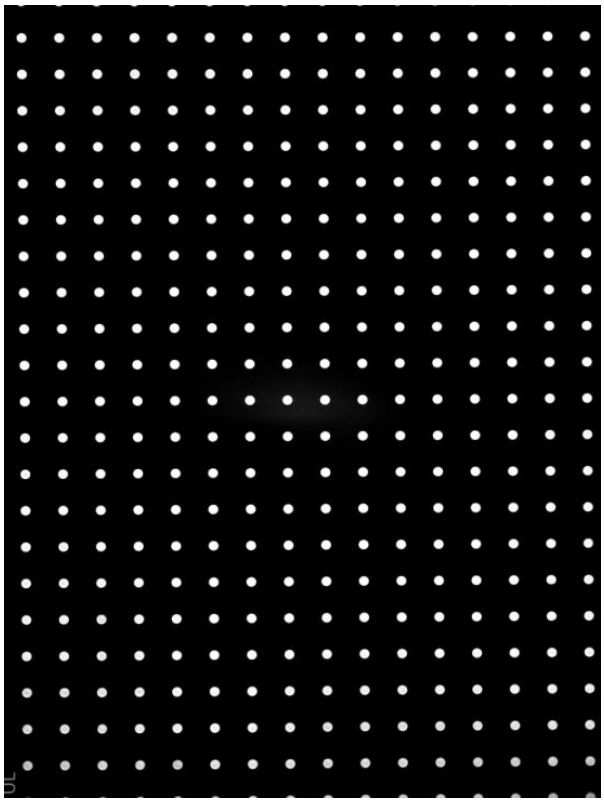


Figure 3.3: Distortion Algorithm's Baseline Results

end user. Using SIMD instructions can be help increase the performance of this algorithm.

The main image processing occurs in two nested loops, which move over the entire destination image as shown in Figure 3.4. The function *map1bli* begins by first setting the scale of the image, and setting the *x* and *y* indices of the distortion matrix with the *SetIndexX* and *SetIndexY* functions, respectively. This information is used by the algorithm to determine which value from the distortion matrix must be used. Next, the distortion vectors (*dvx* and *dvy*) are calculated in the *GetDistortionVector* function. The vectors are based on the distortion matrix and the current pixel being processed in the destination image. The function contains static variables (*cx, cy, ccx, ccy*), that don't change every time the function is called. At the end of the function, the variables *pxindex* and *pfyindex* are set equal to *xindex* and *fyindex*, respectively. The new values of *xindex* and *fyindex* are compared to the saved values, *pxindex* and *pfyindex*, as shown. If they are equal, then the processing of the static variables is skipped to help increase the performance. If they are not equal, then the static variables must be recalculated. These variables are then used to calculate the distortion vectors, *dvx* and *dvy*. The vectors contain an integer part in the 16 most significant bits and a fractional part in the 16 least significant bits. The *GetDistortionVector* function returns the vectors to the *map1bli* function.

The fractional and integer parts of the distortion vectors are separated, and the integer part is used to determine the correct pixel from the source image. Next, the values of this pixel and three surrounding pixels are retrieved from the source image. Bilinear interpolation occurs between these pixels based on the fractional parts of the distortion vectors, and the resulting value is saved to the destination image. The process continues for all the pixels in the destination image. The color channels are processed separately; therefore, the *map1bli* function is called three times to process the three channels. This allows different distortion matrices to be applied to each channel.

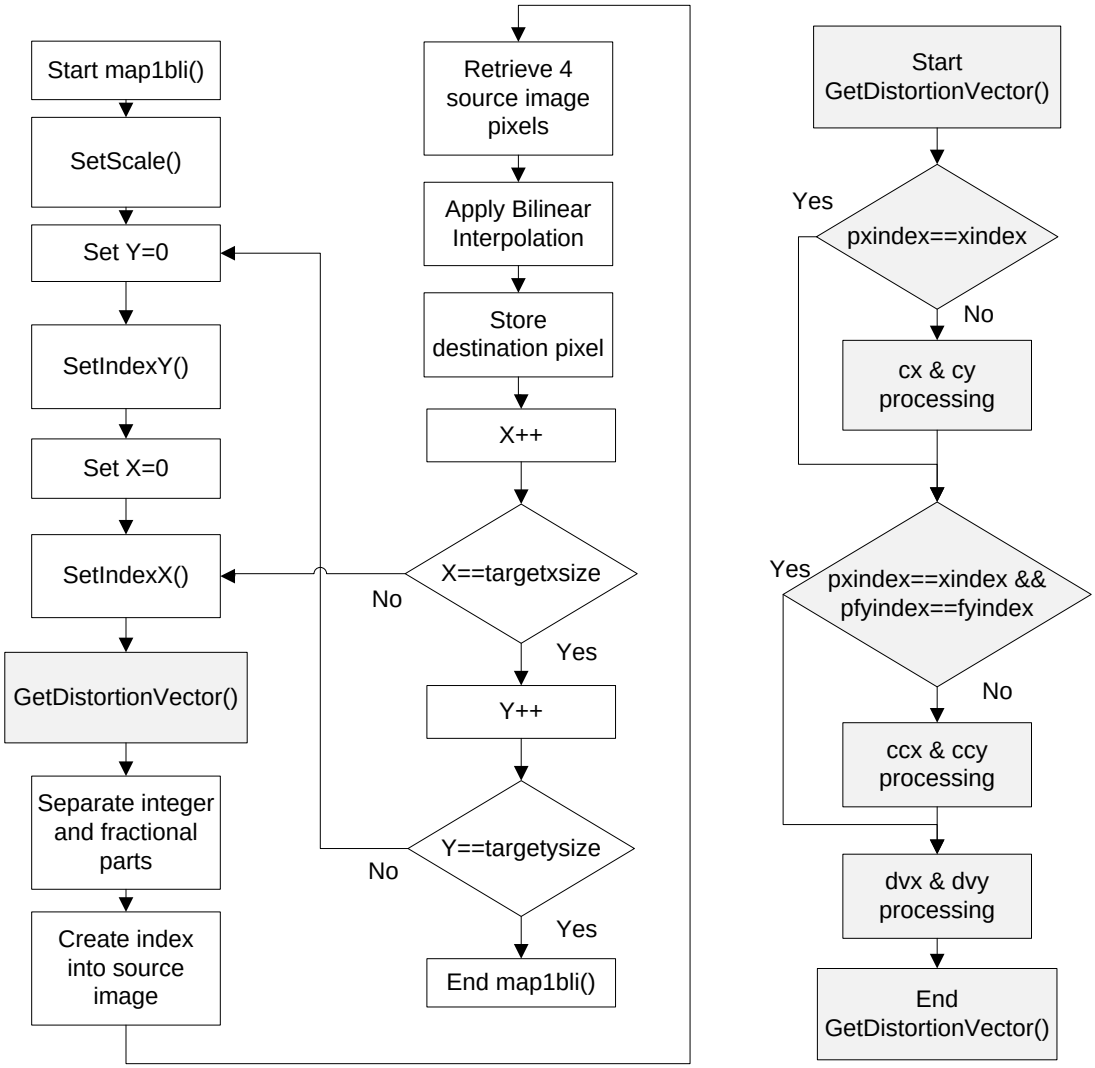


Figure 3.4: Distortion Algorithm's Program Flow

Chapter 4
Experimental Procedure

The following sections describe the tests which were performed that utilized the NEON SIMD instructions. Section 4.1 and Section 4.2 describes the tests related to the bilinear interpolation algorithm and the distortion algorithm, respectively.

4.1 Bilinear Interpolation Tests

The bilinear interpolation algorithm has three test cases with each test performed over five different interpolation factors, and with a source image of one million total pixels. The NEON based code is written manually because the vectorizing compiler cannot find any vectorizable loops. The first test case is the baseline, which is described in Section 3.3.1. Section 4.1.1 describes NEON1, which is the first test using the SIMD intrinsic functions with parallel color channel processing. Section 4.1.2 describes NEON2, which is the second test using the SIMD intrinsic functions with the processing of four pixels concurrently.

4.1.1 NEON1 Test

NEON1 is the first test case involving the NEON SIMD intrinsic functions. The code is shown in Appendix C, and the program's flow is shown in Figure 4.1 with the vectorized parts in dark grey. This test processes all three color channels in parallel rather than sequentially. A lane of the NEON registers is not used because the image has three color channels, but four lanes in each register. All NEON variables use the 128-bit quad registers which require variables that are either 16 bits and fill eight lanes or 32 bits and fill four lanes.

This test starts by calculating the variables *hc1* and *hc2* without the use of SIMD instructions. These variables are then duplicated into separate NEON registers, referenced as *hc1vec* and *hc2vec*. The duplication instruction copies the value into each of the eight lanes. The same process is done for the variables *wc1* and *wc2* which are stored in NEON variables *wc1vec* and *wc2vec*, respectively. The image processing begins by first retrieving the four pixels used for interpolation and storing them into *pixelavec* and *pixelbvec*. The four pixels are chosen based on the values of *OffsetX*,

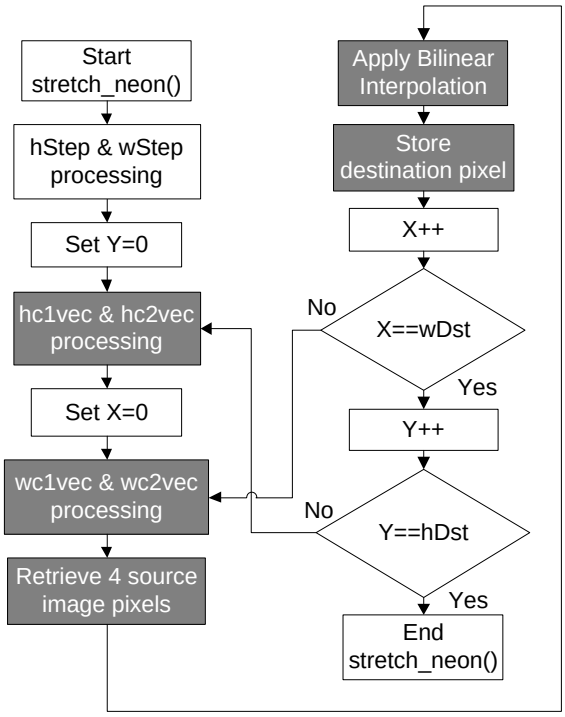


Figure 4.1: Bilinear Interpolation's Program Flow for NEON1

OffsetY, and the source image's dimensions. The register layouts for the *pixelavec* and *pixelbvec* variables are shown in Figure 4.2. Because *pixel1* and *pixel3* are stored sequentially in memory, they are loaded with one instruction into a double NEON register (64 bits). Next, the values are reinterpreted from two 32-bit values to eight 8-bit values, and then extended to 16 bits. The extension fills the quad registers and allows the image processing to occur on a width of 16 bits. The same process is done for *pixelbvec* with the *pixel2* and *pixel4* variables, which are also stored sequentially in memory. The *builtin_prefetch* function is used to preload the cache with the next likely source data. The function's first argument is the address of the expected data, the second argument is set to zero for read/write access, and the third argument is set to two for locality. The locality determines how long the data should stay in the cache. The remaining image processing is similar to the baseline code except for the use of SIMD intrinsic functions. Many of the shift and bitwise AND operations are not needed because of how the NEON registers are set up. At the end, one double register contains the result with four lanes of 16-bit values. The values are reduced to four lanes of 8-bit values and stored in memory pointed to by the *Dst* variable. The *Dst* pointer is incremented, the

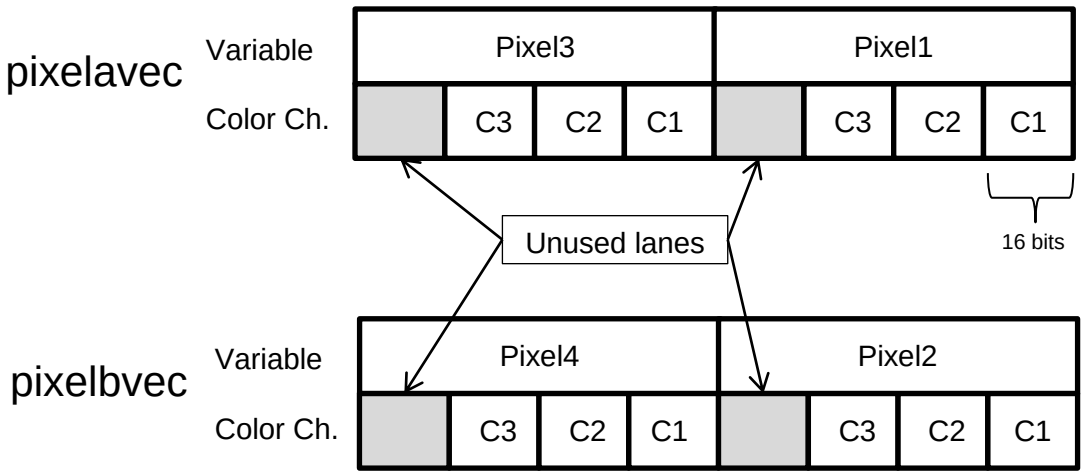


Figure 4.2: Bilinear Interpolation NEON1's SIMD Register Setup

coefficients are increased, and the process repeats in a loop through all the pixels in the destination image.

4.1.2 NEON2 Test

NEON2 is the second test case involving the NEON SIMD intrinsic functions. The code is shown in Appendix D, and the program's flow is shown in Figure 4.3 with the vectorized parts in dark grey. This test processes four sequential pixels in parallel rather than one at a time. As with the baseline, each color channel is processed separately. The variables use the quad registers, and each pixel has a 32-bit lane. Unlike NEON1, this setup does not waste lanes because four values are being processed concurrently and four lanes are available for processing.

This test is very similar to the baseline except for the use of SIMD intrinsic functions. The factors *hc1* and *hc2* are calculated using ARM instructions and copied into the four 32-bit lanes of *hc1vec* and *hc2vec*, respectively. The values for *wc1* and *wc2* change with each *x-loop* iteration. The *x-loop* is the inner loop of the processing and defines the x-coordinate for the destination pixel. Therefore, they are calculated as a four element array in a loop, and a NEON instruction is used to load them from memory into the 128-bit registers. Each lane of the source image registers is set individually, because the values loaded into the NEON registers may not appear sequentially in memory. The interpolation part of the processing is accomplished in the same way as the baseline code except four pixels are processed concurrently. The whole destination register is

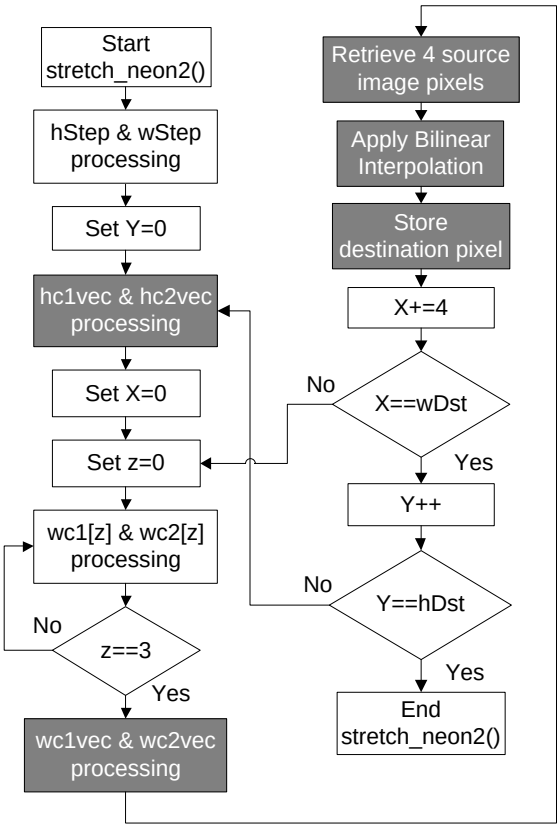


Figure 4.3: Bilinear Interpolation's Program Flow for NEON2

sent to memory pointed by *Dst*. The destination pointer is incremented by four during each *x-loop* iteration and decremented at the end of the *x-loop* if the destination width is not divisible by four. This allows the *x-loop* to overstep and return if the destination image width is not divisible by four. This method does waste some processing time on pixel values that are in the end discarded.

4.2 Distortion Tests

The distortion algorithm is run with twelve different test cases with each test using the same input image of eight million pixels and three color channels, and a 23 by 17 distortion matrix with two dimensional vectors. The code was compiled with the vectorizing compiler, but it could not find any vectorizable loops in the image processing part of the code. Therefore, the SIMD instructions were inserted manually. The first test is the baseline code as described in Section 3.3.2. The next four tests use the NEON SIMD intrinsic functions, and are described in Section 4.2.1 through Section 4.2.4. Section 4.2.5 through Section 4.2.7 describes the three assembly based tests in which

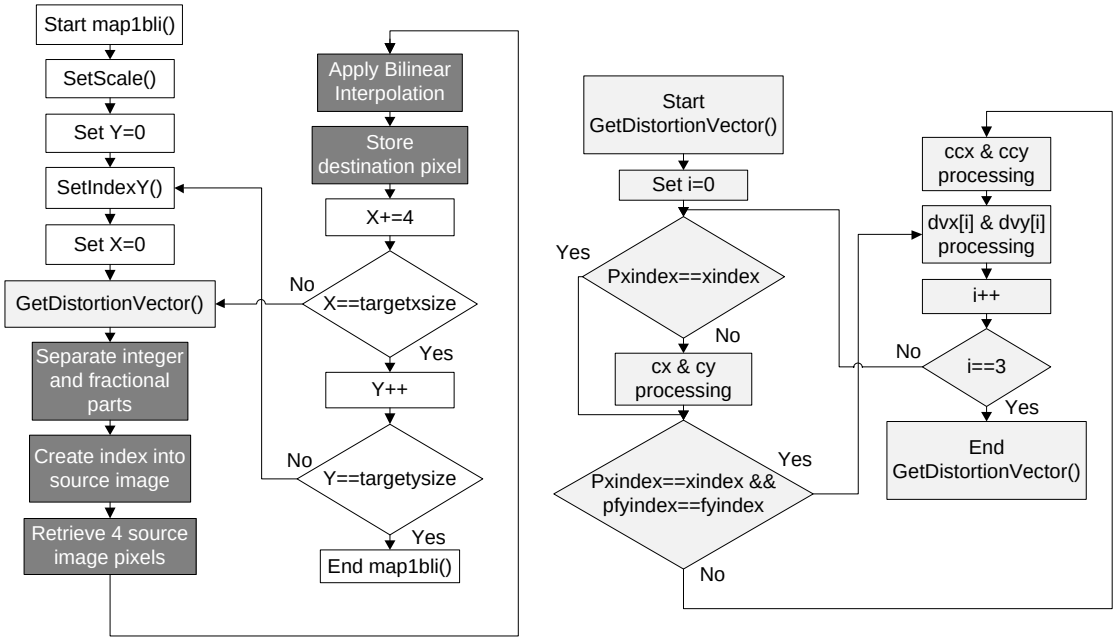


Figure 4.4: Distortion Algorithm's Program Flow for the NEON1 Test

the assembly is altered with SIMD instructions and other techniques. The remaining tests attempted additional ways to speed-up the execution of the algorithm. Section 4.2.8 explains the move from 32-bit operands to 16-bit operands. Section 4.2.9 discusses the test using both the integer and floating point functional units. Section 4.2.10 discusses using both the ARM processor and NEON coprocessor in parallel during the image processing. Section 4.2.11 discusses enhancements at the assembly level made to the baseline code without using NEON instructions.

4.2.1 NEON1 Test

This NEON1 test case applies NEON SIMD intrinsic functions to the main image processing by computing four pixels per iteration instead of one pixel as shown in Figure 4.4 with the vectorized parts in dark grey. First, the *GetDistortionVector* function is altered by including the *SetIndexX* function so an extra function call can be eliminated. Second, because the *GetDistortionVector* function is a part of the code that cannot be calculated easily in parallel with SIMD instructions, it is executed four consecutive times using only ARM instructions. The result is saved to two 4-element C arrays, referenced as *dvx[]* and *dvy[]*, which are then loaded into NEON registers. Next, the fractional and integer parts are separated and the index into the source image is created using parallel operations with the NEON coprocessor. The index is saved as a vector to memory

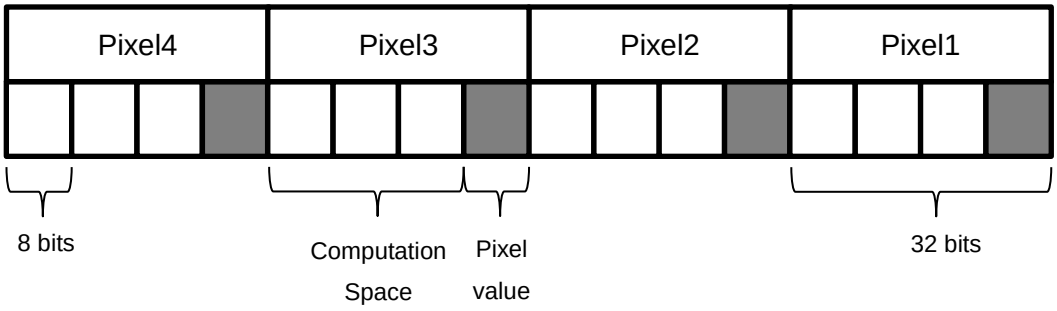


Figure 4.5: Distortion Algorithm's SIMD Register Setup

because the SIMD registers cannot be used as an index into memory. Saving the vectors to memory avoids the 20 cycle stall when transferring from the NEON coprocessor to the ARM processor. The ARM registers are loaded with the index from memory, which is then used by the NEON coprocessor to load the source image data into SIMD registers. Each NEON register lane is loaded individually from the source image because the pixels may not occur sequentially in memory and therefore multiple pixels cannot be loaded with one instruction. Figure 4.5 shows how the four pixels are placed in a SIMD register. The pixels are 8-bit values, but they are loaded into 32-bit lanes because the bilinear interpolation step requires 32 bits to perform the computations. The bilinear interpolation of the four pixels occurs concurrently using the source image's values and the fractional parts of the distortion vector. Each lane is saved individually to the destination image array. The destination array is incremented and the loop repeats until the destination image has been processed.

4.2.2 NEON2 Test

The NEON2 test case adds onto the NEON1 test case with vectorizing the calculations in the *GetDistortionVector* function. The program flow for this test is shown in Figure 4.6 with the vectorized parts in dark grey. One way to accomplish the parallelizing is to compute all four components of the distortion vectors, *dvx[]* and *dvy[]*, in parallel rather than in a loop. This requires removing the *pfyindex* and *pxindex* comparisons and computing the static variables *cx, cy, ccx,* and *ccy* during every function call. These are rarely recomputed (about once every 150 function calls) as recomputing them every function call would likely increase the time this function takes to complete. This option was not chosen for its likely performance decrease.

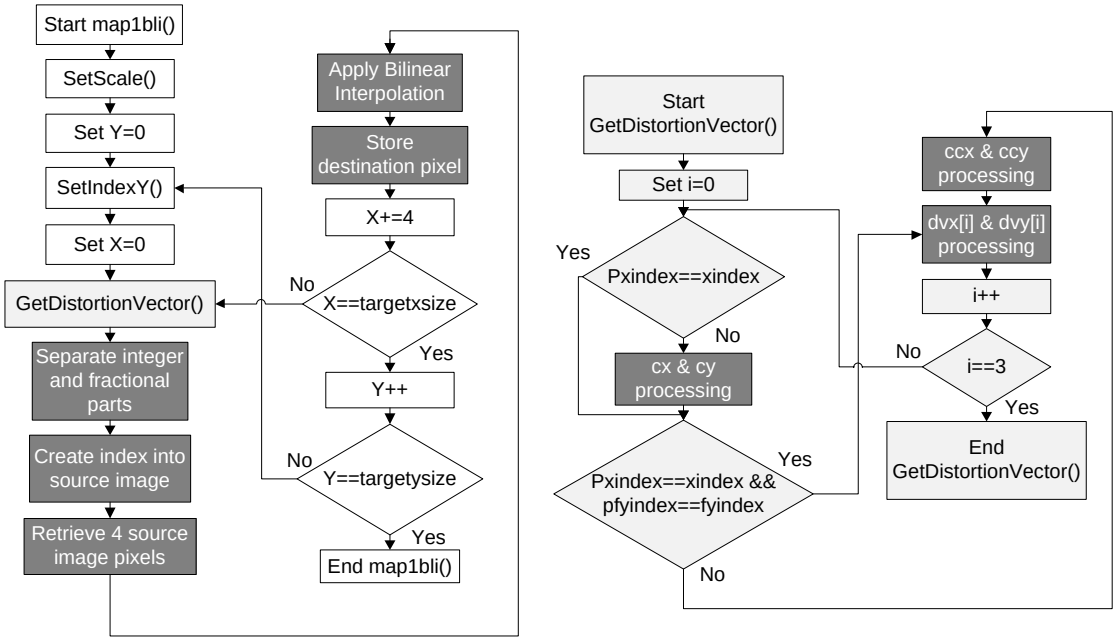


Figure 4.6: Distortion Algorithm's Program Flow for the NEON2 Test

Another option is to leave the current structure of the function, and parallelize the computing of the *cx, cy, ccx,* and *ccy* variables and the distortion vectors. This option is more difficult because those variables are not easily calculated in parallel. Also, additional instructions are needed to ensure the data is in the correct lanes. The result of this function is two distortion vectors that are contained in NEON registers. This eliminated the need to load the vectors from memory to be processed. The main image processing is identical to the NEON1 test. This option is chosen because it does not recompute the static variables and thus should have increased performance.

4.2.3 NEON3 Test

The NEON3 test case adds onto the NEON1 test case with minor rearranging of the code. Figure 4.7 shows the program flow for this test with the vectorized parts shown in dark grey. The *GetDistortionVector* function was moved to before the start of the inner *x-loop* and to the middle of image processing. The former is needed for the first run of the *x-loop*, and the latter will precompute the distortion vectors for the next iteration of the loop. However, the precomputation does not occur during the last iteration of the *x-loop* because the precomputation is not needed. The rearranging attempts to operate the ARM and NEON processors more concurrently, and help decrease the amount of stalls due to data dependencies. Because *GetDistortionVector()* uses mostly the ARM

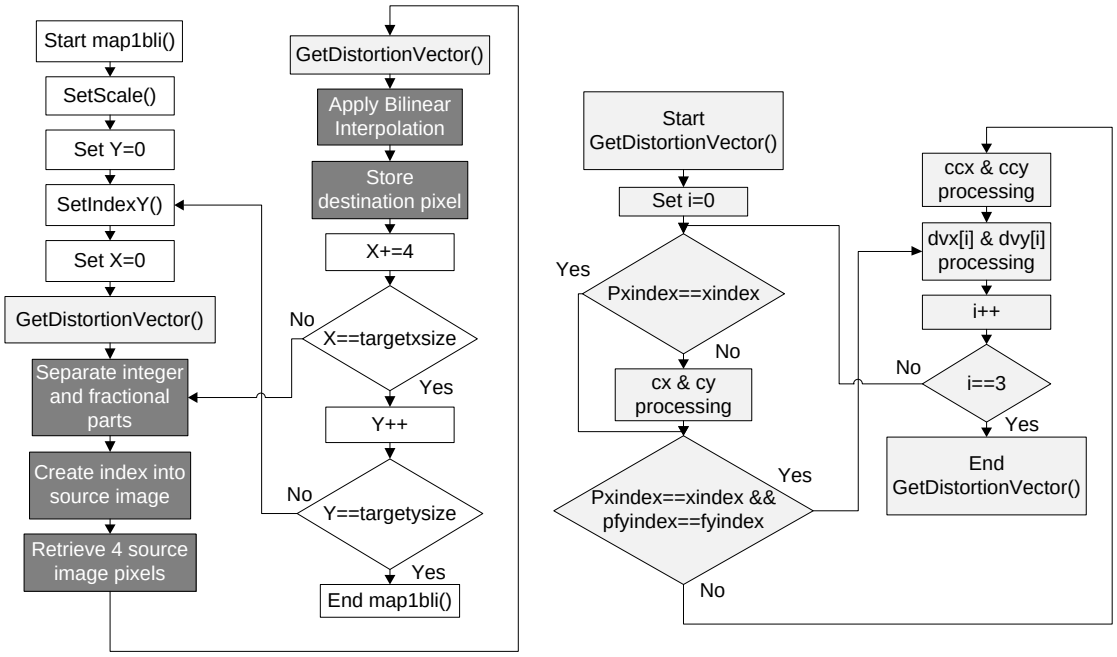


Figure 4.7: Distortion Algorithm's Program Flow for the NEON3 Test

processor and the image processing uses mostly the NEON coprocessor, placing this function in the middle of the image processing should allow the processors to act more in parallel.

4.2.4 NEON4 Test

The NEON4 test case adds onto the NEON3 test case with the use of the compiler's "hint" functions. The cache "hint" function is used to prefetch the expected source and destination images for the next iteration of the *x-loop*. The *builtin_prefetch* function is implemented with the expected next address of the source or destination pixels as the first argument. The second argument is set to zero for the read only source image and set to one for the write to the destination image. The third argument is set to two to leave the data in the cache as long as possible. The "hint" function for the branch prediction is used when calling the *GetDistortionVector* function within the image processing part of the code. This is accomplished with the *builtin_expect* function, which uses the comparison expression as the first argument, and the expected result of the comparison as the second argument. Because the *GetDistortionVector* function is called every *x-loop* iteration except for the last, it can be expected that the branch will always be true. Therefore the second argument is set to one which tells the compiler the branch is usually taken.

4.2.5 ASM1 Test

The ASM1 test case starts with the assembly code from the NEON4 case. Figure 4.8 shows the program flow for this test with the NEON vectorized parts shown in dark grey. The compiler performed a few optimizations with the code. First, it inlined the *GetDistortionVector* function both before the *x-loop* (shown as *GetDistortionVector*) and in the middle of image processing (shown as *GetDistortionVector_prefetch)*. Inlining functions decreases the number of branches, which can decrease the branch mispredictions. Second, the *i-loops* in both of these functions are unrolled by four and a few unneeded branches are eliminated. This should help reduce the number of program counter changes and possibly the number of branch mispredictions. The compiler *builtin_prefetch* function is compiled into an assembly *PLD* instruction. This instruction signals to the memory system that a data load from the specified address is likely. The compiler *builtin_expect* function did not compile into an assembly instruction and there is no evidence that this function is implemented.

Using the compiler's assembly code, this test removes one branch and some unneeded loads from and stores to memory. The first change removes the equality check for *pfyindex* and *fyindex*. The *fyindex* variable only changes after the *SetIndexY* function is called, and the *pfyindex* variable is set equal to *fyindex* after the distortion vectors are computed. Therefore, the *GetDistortionVector* function always initially processes *cx, cy, ccx,* and *ccy* because it is after the *SetIndexY* function. In the *GetDistortionVector_prefetch* function, the equality check for *fyindex* and *pfyindex* is not needed because they will always be equal. The second change involves altering how the program stores static variables used by the distortion vector functions. The compiler handles the variables by storing their address, instead of the actual value, to the stack. To access these variables, the address must first be loaded from the stack and then the value can be loaded or stored based upon that address. This was changed to save or load the value directly to or from the stack which eliminated a load for each of the static variables. The third change altered the calculations of the *cx, cy, ccx,* and *ccy* variables. The compiler does not fully utilize the ARM registers and therefore intermediate values are stored to memory rather than kept in registers. The code is rearranged and registers changes such that the intermediate values were rarely stored to memory, which eliminated many load and stores instructions.

The NEON SIMD code has a few modifications as well. One modification helps to more fully utilize the NEON register file by keeping constant values in registers. Some

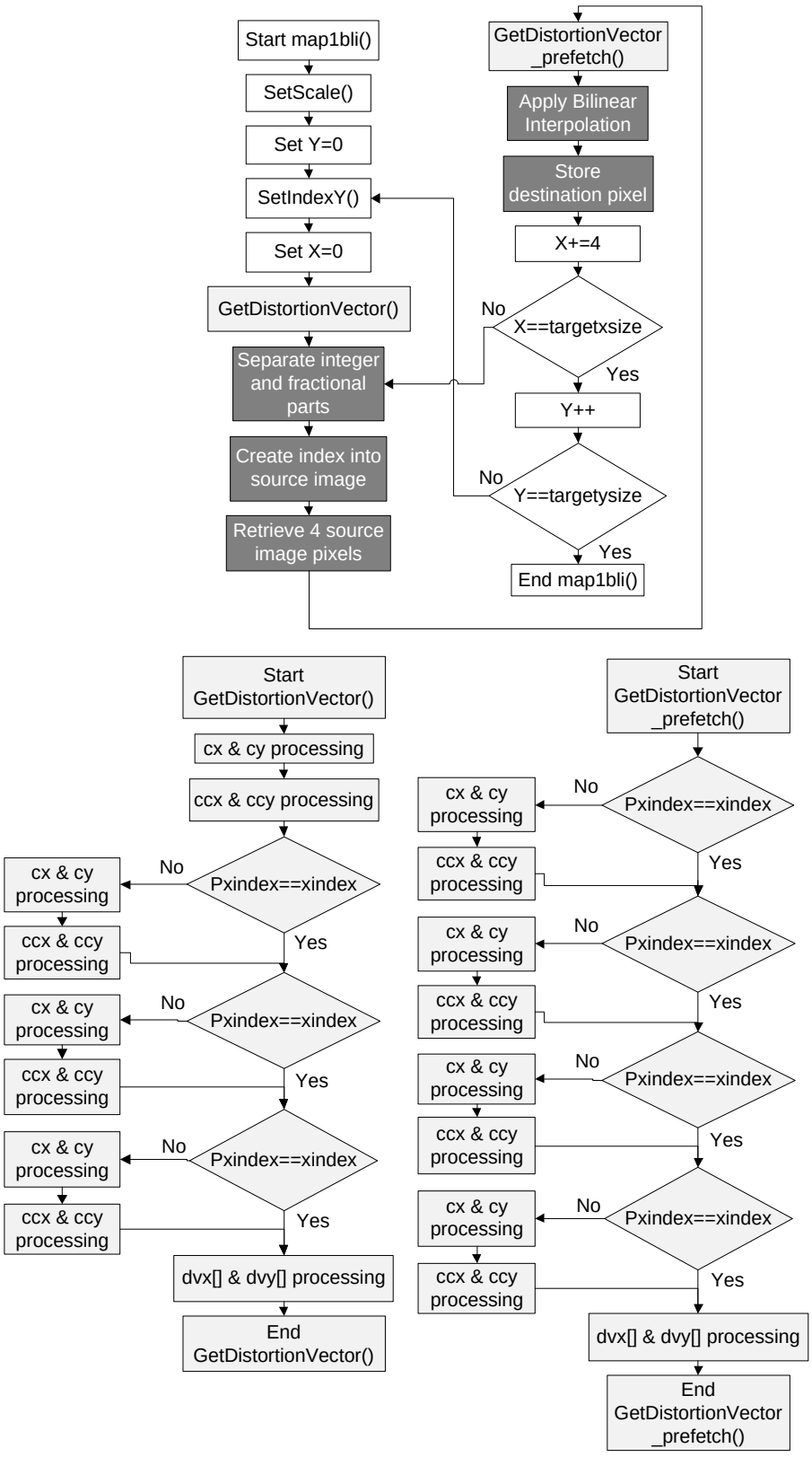


Figure 4.8: Distortion Algorithm's Program Flow for the ASM1 Test

values in the image processing do not change with each loop iteration. Intially, these values are loaded from memory or computed during each loop iteration as needed. This test case keeps the constant variables in the NEON registers. This change makes processing the image more difficult because there are less registers available to keep data. Another change exploits the option of dual issuing instructions. The NEON and ARM processors can issue two instructions at a time if one instruction is a load or store and no dependencies exist. The compiler attempts to accomplish this, but manually altering the code exploits this possibility even more. The code is modified to put load and store instructions near other instructions and to remove data dependencies between instructions.

4.2.6 ASM2 Test

The ASM2 test case uses the assembly from the ASM1 test, and vectorizes the calculation of the distortion vectors, *dvx* and *dvy*, in both the *GetDistortionVector* and *GetDistortionVector_prefetch* functions. Figure 4.9 shows the program flow for this test with the NEON vectorized parts shown in dark grey. The static variables *ccx* and *ccy* used for this calculation are either calculated with ARM instructions and transferred to NEON registers, or loaded from memory into NEON registers. The distortion vectors are then calculated based on these variables, and kept in NEON registers until they are separated into their integer and fractional parts in the image processing part of the code. This saves an extra store from ARM to memory and load from memory to NEON, and processes the vectors in parallel. The image processing part of the code is identical to the ASM1 test.

4.2.7 ASM3 Test

This final assembly test builds on the ASM2 test, but processes eight pixels instead of four pixels per iteration. In previous tests, the NEON registers were not fully utilized during the image processing. These extra registers are now used to process twice the number of pixels per iteration which can increase performance. Calculating more pixels can help limit the data dependency stalls between the instructions and reduce the number of branches. Stalls from structural dependencies may arise, but because the NEON functional units are pipelined, the effect should be minimal. The calculation of distortion vectors are unrolled by a factor of eight to correspond with the eight pixels being processed. The preload cache instruction (*PLD*) is removed to see the effect of not

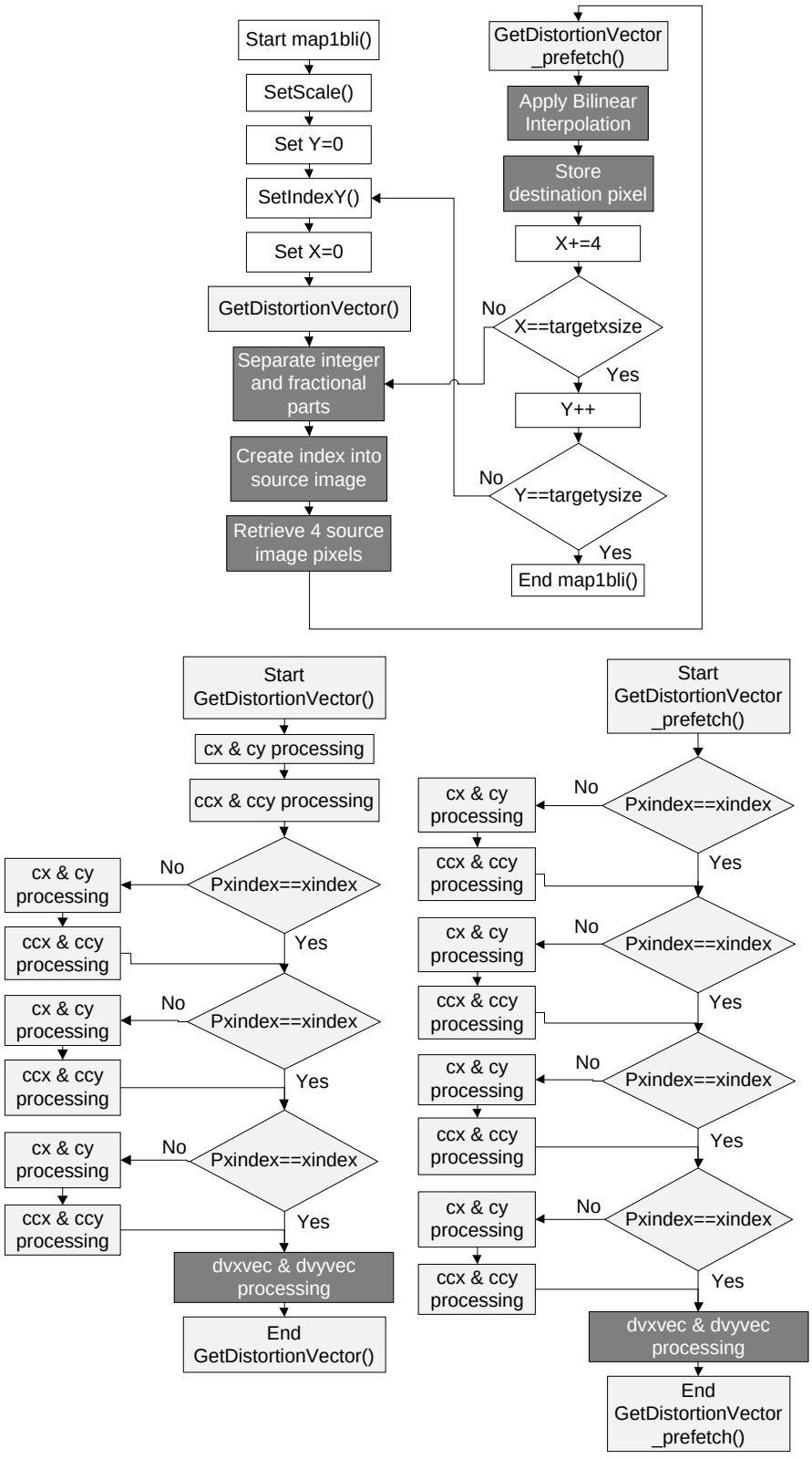


Figure 4.9: Distortion Algorithm's Program Flow for the ASM2 Test

preloading the cache has on the performance. Although the preload instruction could minimize cache misses, the instruction does take time to execute.

4.2.8 32-bit to 16-bit test

This test takes the baseline code and alters it by using 16-bit operands instead of 32-bit operands with non-SIMD code. For the distortion vectors, the 16-bit operands require changing from 16-bit integer and fractional parts to 8-bit parts. The rest of the image processing occurs with 16 bit values. Truncating and rounding will be likely during the multiplication and addition of variables. Using 16 bits doubles the amount pixels that can be in a NEON register which should increase performance. However, losing half the precision could cause undesirable errors in the destination image.

4.2.9 Integer and Floating Point Test

This test uses the integer and floating point functional units of the NEON coprocessor in parallel. The NEON coprocessor has an integer ALU, multiplier, and shifter and a floating point adder and multiplier. The test uses the NEON intrinsic functions, with integer and floating point data types, and processes four pixels using integer calculations in parallel with four pixels using floating point calculations. The only portion of the code tested is the bilinear interpolation in the image processing, but the test could be expanded to the rest of the code. The source pixels and distortion vectors are converted to floating point numbers and stored in NEON registers. The code has shift left operations which are not able to be processed with the floating point functional units. So instead of shifting left, the floating point numbers are multiplied by a power of two corresponding to the shift. A shift right operation is also present in the algorithm. Because a floating point shifter or divider are not available, the shift right is accomplished in the integer part of the coprocessor. The conversion between integer and floating point numbers takes two cycles to complete for the NEON coprocessor. The initial conversion and converting for shift right instructions will likely cause an increase in the number of cycles and therefore decreased performance. The test uses single precision floating point numbers which reserve 23 bits for the fractional part. Moving from 32-bit operations to 23-bit operations may produce errors in the destination image due to the truncation of values. The added cycles along with image errors may cause this test to perform insufficiently.

4.2.10 ARM and NEON Test

This test uses the NEON4 test case, and adds the processing of one pixel per iteration with the ARM processor to the four pixels per iteration with the NEON coprocessor. The ARM and NEON coprocessors can run in parallel and this test attempts to exploit this feature. First, the loop in the *GetDistortionVector* function is changed to produce five value *dvx* and *dvy* distortion vectors. Four values are used for SIMD and one value is used for ARM processing. Processing five pixels per iteration should cause a small increase in performance because structural hazards will not be present between the ARM and NEON pipeline. However, the ARM and NEON coprocessors will have to access the same cache block which could cause some stalls due to ordering issues. This test could also be expanded to ten pixels per iteration with eight pixels being processed with SIMDs and two pixels being processed without SIMDs.

4.2.11 Revised Baseline Test

This test converts the baseline code without NEON instructions to assembly, and applies the same non-NEON optimizations that are present in the ASM1 test case. First, the *GetDistortionVector* function is moved before the *x-loop* and in the middle of the image processing so the distortion vectors are prefetched. Second, unneeded comparisons and branches in the *fyindex* and *pfyindex* are removed. Third, the loads and stores of the static variables are changed to store and load directly to the stack instead of the address pointed to by the stack. This test only processed one pixel per iteration, but could be expanded to process four pixels per iteration. Four pixels per iteration would better match the NEON tests, but would likely not increase the speedup up due to insufficient number of ARM registers.

This test is used as another baseline to see how the NEON SIMD instructions improved the performance. The test can be compared to the best performing assembly test. If the performance increase of the baseline and assembly is the same, then SIMD instructions do not provide a performance benefit. Most likely, the performance increase of the baseline will be less than that of the assembly test. This can help show that SIMD instructions are very valuable in increasing the performance of this and other algorithms.

Chapter 5

Results and Discussions

Both algorithms demonstrate increased performance when using the NEON SIMD instructions. Using SIMD instructions alone doubled the speed of both algorithms, and altering the assembly code of the distortion algorithm tripled the speed. Section 5.1 and section 5.2 discuss the results of the bilinear interpolation algorithm and the distortion algorithm, respectively. Section 5.4 concludes with the contributions this work can provide to others.

5.1 Bilinear Interpolation Results

The highest performing bilinear interpolation test is nearly twice as fast when compared to the baseline. The NEON1 test case processes one pixel per iteration and the three color channels in parallel. The NEON2 test case processes four pixels per iteration and the three color channels separately. The theoretical maximum speedups for the NEON1 and NEON2 test cases are three and four, respectively. Figure 5.1 shows the actual

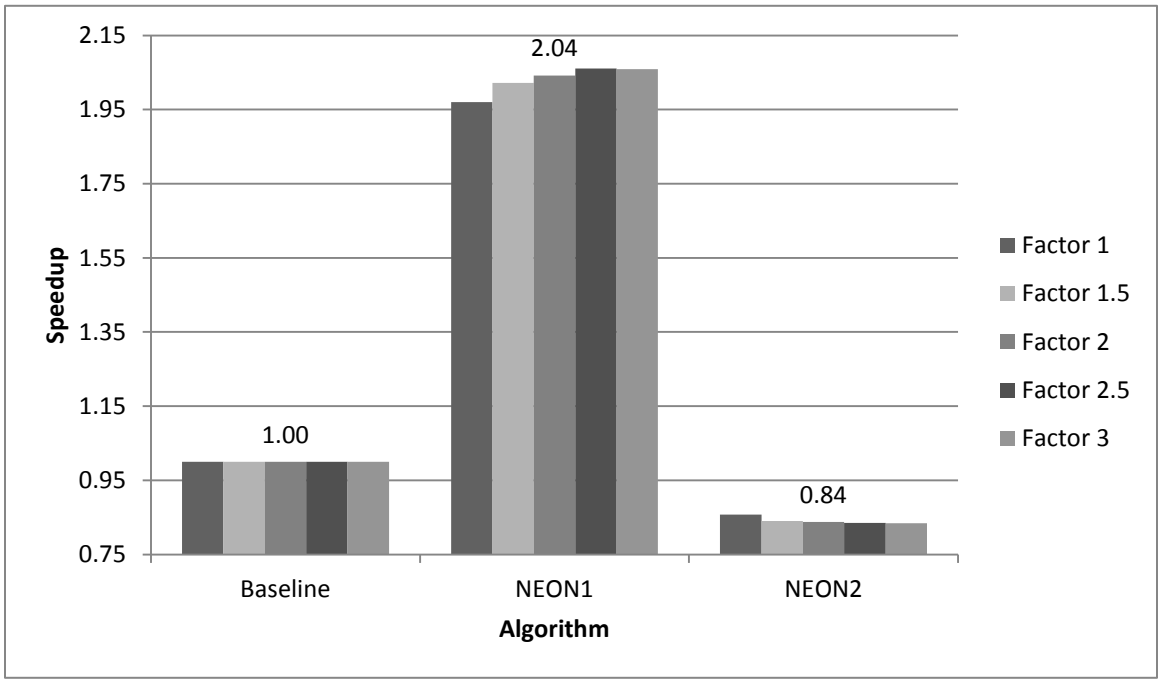


Figure 5.1: Bilinear Interpolation's Speedup with Different Interpolation Factors

speedup of the three test cases relative to the baseline (5.06 seconds). Five different interpolation factors are chosen to interpolate a one million pixel image. For example, an interpolation factor of two doubles both the width and the length of the image. When using an interpolation factor of one, the image does not change size, but the interpolation still occurs on this image. The speedup is not affected much by the interpolation factors, but is affected by the algorithm used. The NEON1 test case has the highest speedup which ranges from 1.97 for a factor of one to 2.06 for a factor of three. The NEON2 test case is slower than the baseline with speedups ranging from 0.86 for a factor of one to 0.83 for a factor of three.

The low speed-up in the NEON2 test could be caused by the increased number of instructions and data dependencies. The NEON1 test is able to load two pixels from memory with one instruction because of how the pixels are stored in memory. The NEON2 test uses one instruction for each pixel because the algorithm may not select sequential pixels from the source image array. Also, the NEON2 test case has more instructions due to the shift, AND, and multiply operations, and these instructions can cause more stalls due to data dependencies. The NEON2 test has four more shift and twelve more AND operations per four pixels, when compared to the NEON1 test case. These 16 additional instructions can require about 16 million more cycles to complete when interpolating a four million pixel image (interpolation factor of two). Although, most instructions take one cycle to complete, the NEON multiply instruction takes four cycles. The NEON1 test has 16 multiply instructions for every four pixels, and the NEON2 test has 18 multiply instructions for every four pixels. For example, a four million pixel target image would require two million extra multiplies for the NEON2 test. This translates to up to eight million extra cycles, assuming that each multiply has a data dependency. The NEON2 test has about a quarter the instructions of the baseline. However, there are more cache accesses because of how the *wc1vec* and *wc2vec* variables are loaded. Also, when a cache access does occur, the slower L2 cache is accessed rather than the faster L1 cache. With other instructions included, the NEON2 test case requires many more cycles to complete, which can be attributed to the low speed-up.

The baseline test has fewer L2 cache accesses than the SIMD tests. Figure 5.2 compares the L2 cache accesses and misses of the three test cases when interpolating an image by a factor of two. The baseline has the least number of L2 cache accesses most likely because the data it needs is loaded into the L1 cache, and the high L2 cache miss rate is due to the data only being used once. Initially, when the algorithm needs

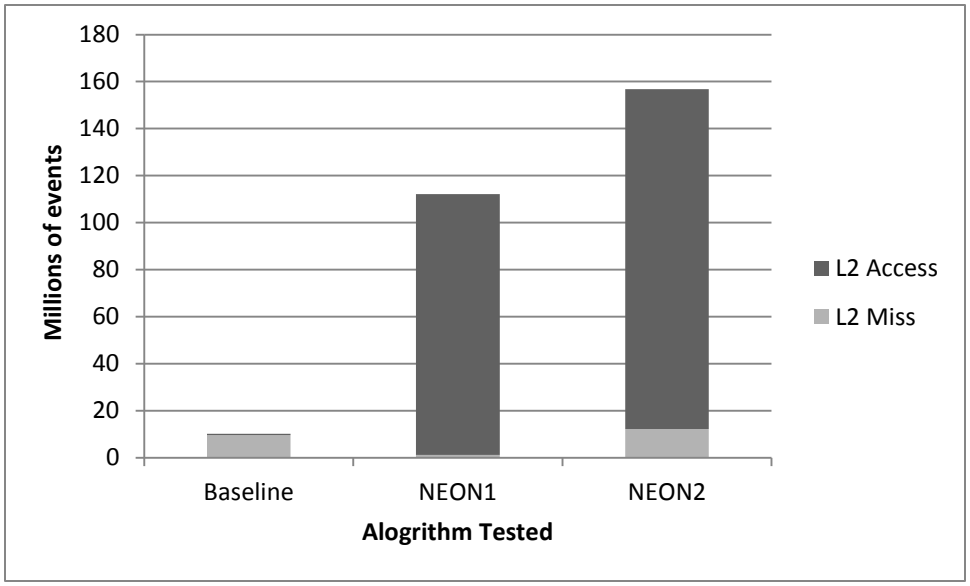


Figure 5.2: Bilinear Interpolation's L2 Cache Events

either source or destination image data, it will be written to the L1 and L2 caches, resulting in misses for both caches. After the data is written to or read from, it is unlikely to be accessed again, and will then be removed from the caches when more space is needed. So, although spatial locality will cause a high hit rate in the L1 cache, the L2 cache will have a high miss rate. The NEON tests have more L2 accesses than the baseline because the NEON load and store instructions can access the L2 cache directly without using the L1 cache. When source or destination image data is needed, the NEON coprocessor will load the L2 cache from memory and bypass the L1 cache. The NEON2 test case has more L2 cache accesses than the NEON1 test case because the increased number of instructions likely requires the intermediate values to be saved to memory due to insufficient number of registers. The cache preload instruction decreases the L2 miss rate for the NEON tests, and would likely have similar results for the baseline test. The miss rate went from 29.5% to 1.0% and 8.1% to 7.9% for the NEON1 and NEON2 tests, respectively. This is a fairly large change in miss rate for the NEON1 test, which may also contribute to the large performance improvement.

Mispredicted branches can also decrease the runtime performance of code. Each branch misprediction causes the pipeline to empty and this incurs a 13 cycle penalty. The number of branches must be kept low to minimize the impact of mispredictions. Also, the branches should have a predictable pattern so the program flow prediction hardware can guess the direction of branches with greater accuracy. Figure 5.3 shows

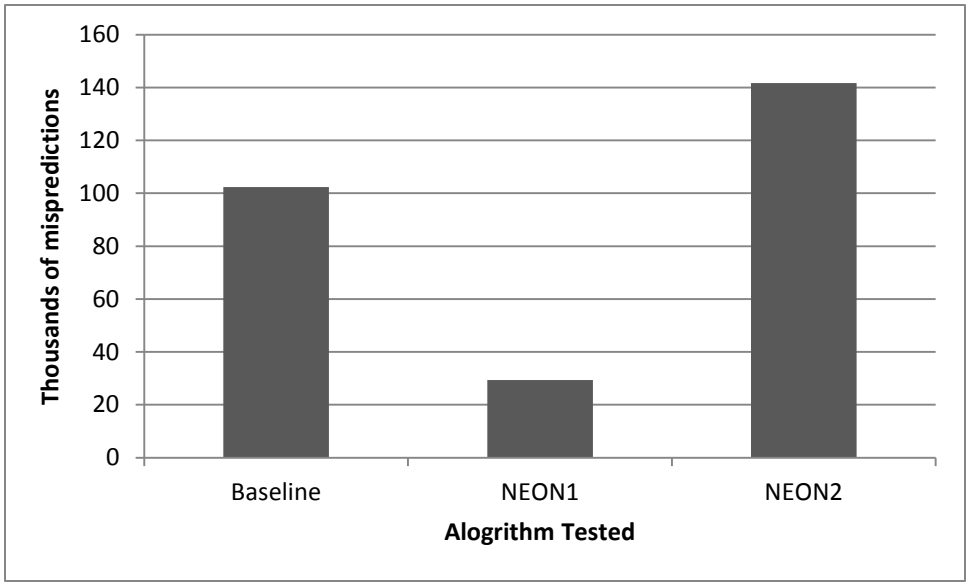


Figure 5.3: Bilinear Interpolation's Branch Mispredictions

the number of mispredictions from the three bilinear interpolation tests. The NEON1 test case has fewer mispredictions than both the baseline and the NEON2 test. The NEON2 test case has a loop to create the *wc1* and *wc2* variables which could increase the number of branches, and therefore increase the number of mispredictions. The branches for the NEON1 and baseline tests are identical. The low mispredictions in the NEON1 test could be attributed to the branch prediction hardware. The branch predictor, implemented as a branch target buffer (section 3.1), may work better for smaller loops, which is the case for the NEON1 test case. The high number of branch mispredictions could be the reason for the slower performance of NEON2 when compared to the baseline and NEON1.

For optimum performance, a balance between ARM and NEON instructions must be found. First, the number of SIMD instructions executed in a row must be kept to a minimum to ensure the NEON instruction or memory queue is not filled. When a queue is filled, no more instructions can be issued from the ARM processor to the NEON coprocessor, and a stall occurs. Figure 5.4 shows the number of cycles the processor stalls as a result of a full NEON queue. The baseline does not show any stalls because SIMD instructions are not used here so the NEON queues are not filled. The NEON2 test showed many more stalls because more SIMD instructions are used here, and the number of load and stores are greater than the NEON1 test. For this metric, the NEON1 test outperforms the NEON2 test, which results in its higher speedup. Second, for optimum performance the ARM and NEON coprocessors should be active for as many

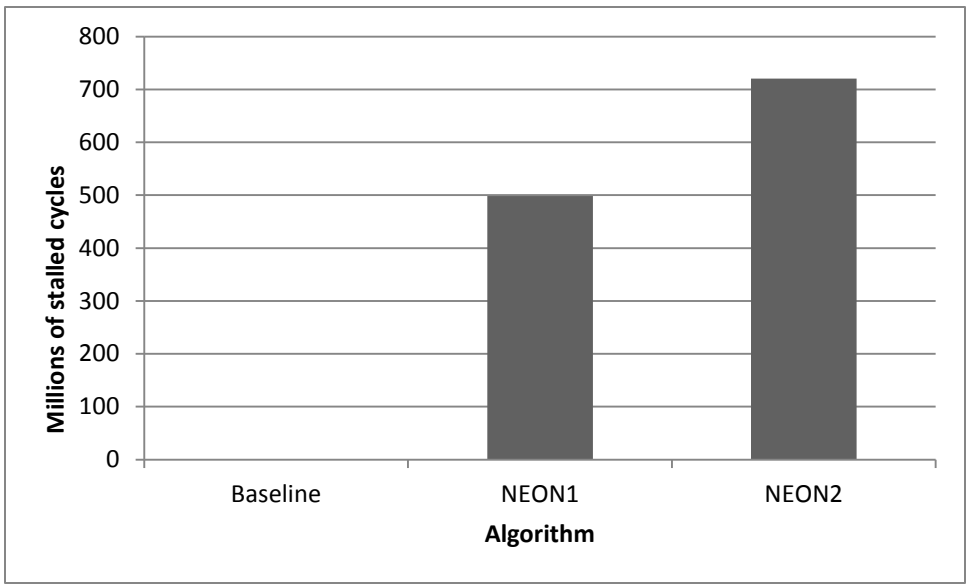


Figure 5.4: Bilinear Interpolation's Full NEON Queue Stalls

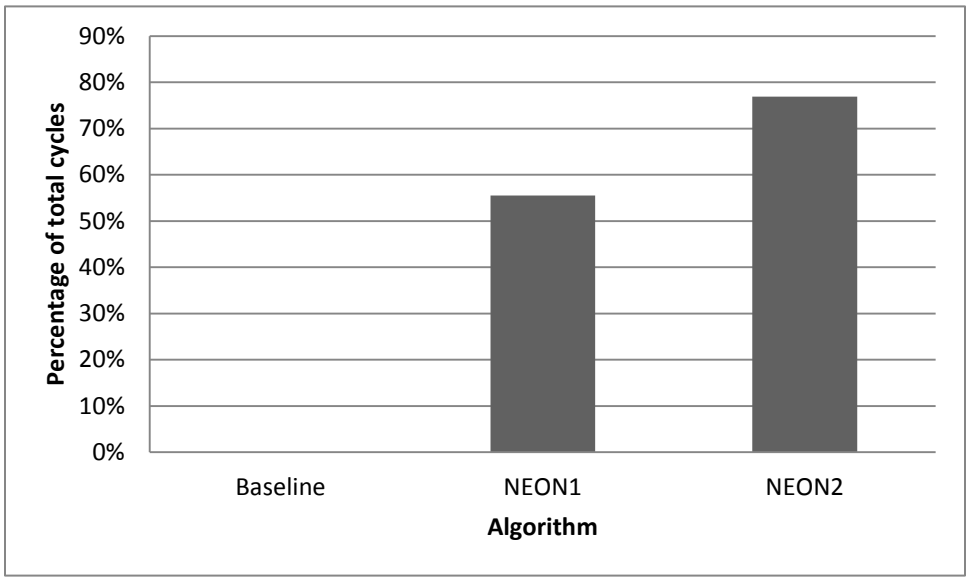


Figure 5.5: Bilinear Interpolation's Both Processors Active

cycles as possible. Figure 5.4 shows the percentage of total cycles in which both processors are actively executing instructions. This is computed by dividing the number of cycles the processors work in parallel by the total cycles the algorithm takes. Ideally, this number should be close to 100% to show that the ARM and NEON coprocessors are always working in parallel. Again, the baseline does not show any concurrent cycles because the NEON coprocessor is not executing instructions. The NEON2 test has a higher percentage of cycles where the processors work in parallel. Loading pixels from

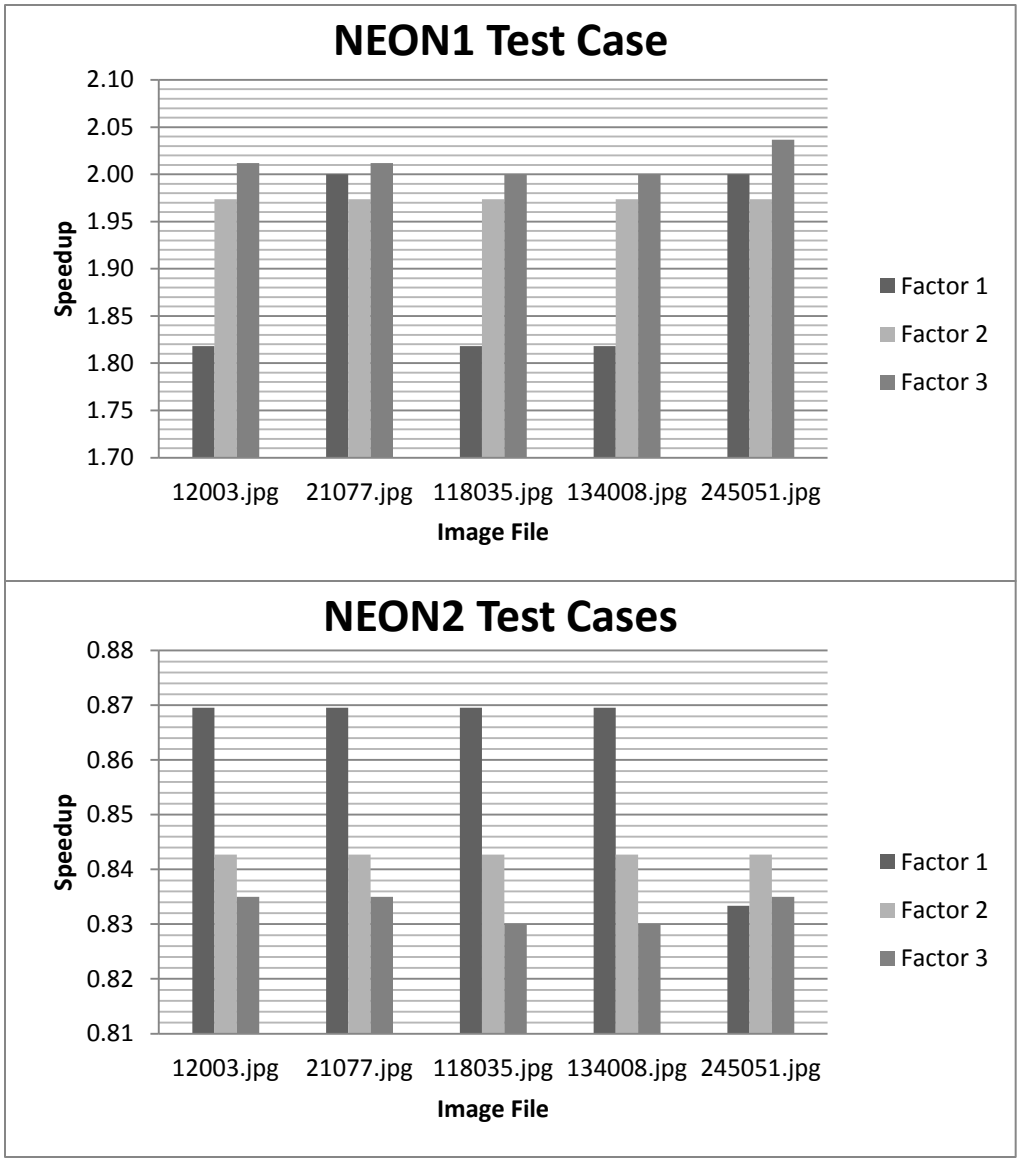


Figure 5.6: Bilinear Interpolation Speedup with Five Different Images

memory uses both ARM and NEON instructions, and therefore, the larger number of load operations in NEON2 can cause the processors to work more concurrently. In both cases, the interpolating of the destination pixel uses SIMD instructions, not ARM instructions. Therefore, the theoretical maximum of 100% concurrent activity cannot be achieved with the bilinear interpolation algorithm.

The same three test cases are applied to five different images from the Berkley image database . These images contain 154,401 pixels per color channel and three color channels. Because the five images contain the same number of pixels, the speedup should be approximately the same for all five images. The speedups for the

NEON1 and NEON2 test cases when compared to the baseline are shown in Figure 5.6 for three interpolation factors. As expected, the speedups are independent of the image, but are dependent on the interpolation factor. The previous results showed the NEON1 test case to have a speedup of between 1.97 and 2.06 times, which is approximately the speedups with this test. The NEON2 test case showed a similar pattern with the previous test having speedups between 0.83 and 0.86 times. This test shows the speedups are independent on the image size or content, but are dependent on the interpolation factor.

Although, neither test reached its theoretical maximum speed-up, the NEON1 test case shows the greatest speedup. The speedup of the test can be mostly attributed to smaller code due to the use of SIMD instructions. The use of SIMD instructions significantly reduces the number of instructions to be executed during the processing. The speedup can also be attributed to low cache misses, low branch mispredictions, and concurrent use of the ARM processor with the NEON coprocessor.

5.2 Distortion Results

The distortion algorithm shows similar speedup results to the bilinear interpolation algorithm. The distortion algorithm uses the SIMD intrinsic functions as with the bilinear interpolation algorithm, but it also uses assembly code for an even larger speedup. Section 5.2.1 discusses the main results of the NEON and ASM tests. Section 5.2.2 discusses the results from other attempts to fully utilize the processor.

5.2.1 Main Results

Figure 5.7 shows the speedup of the different test cases relative to the baseline test (10.01 seconds). Because the test cases process four pixels per iteration of the inner loop, the theoretical maximum speedup should be four. However, the maximum speedup obtained using only the SIMD intrinsic functions is 2.195, and using modified assembly code is 3.090. Modifying the assembly code significantly increases the speedup of the algorithm, but the speedup does not approach the theoretical maximum. As shown, the NEON2 test case does not show an increased speedup compared to the NEON1 test case, and therefore, its code is not used in any subsequent test cases. Several performance metrics are obtained for the tests and are shown in the remainder of this section.

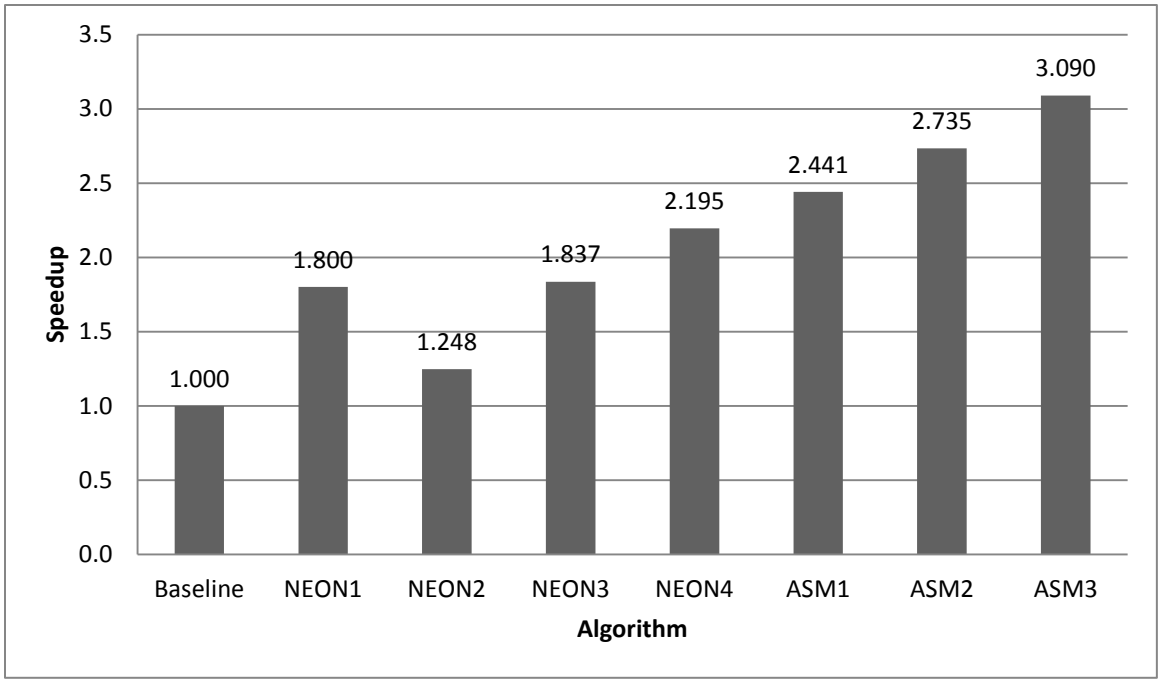


Figure 5.7: Distortion Algorithm's Speedup Relative to Baseline

L2 cache accesses and L1 cache misses should be kept at a minimum to achieve optimum performance. Each cache access or miss can stall the processor, which decreases the performance of the algorithm. The NEON coprocessor is the main cause of L2 cache accesses because it can read/write data from/to the L2 cache directly without updating the L1 cache. Approximately 65% of all L2 cache accesses in the SIMD test cases are from the NEON coprocessor. For the SIMD intrinsic functions test cases the NEON coprocessor has a miss rate of approximately 5%. Figure 5.8 shows the total L2 cache accesses and misses for the distortion algorithm test cases. All the tests except the baseline and NEON2 tests have relatively the same amount of cache accesses. The baseline test shows the least amount of cache accesses because it does not contain any NEON instructions, and the ARM processor primarily uses the L1 cache. The NEON2 test uses SIMD instructions in the *GetDistortionVector* function in an effort to increase performance. This function has many static variables which need to be loaded from memory on a function call and stored to memory on a return. These variables will likely be saved to the L2 cache by the NEON coprocessor. Saving and loading the static variables likely results in higher cache accesses for the NEON2 test. All the tests have relatively low miss rates (3.4% to 9.1%). Adding the cache preload instruction in the NEON4 test caused the miss rate to change from 8.9% to 3.4% when compared to the NEON3 test. The preload instruction is used to preload the L2 data

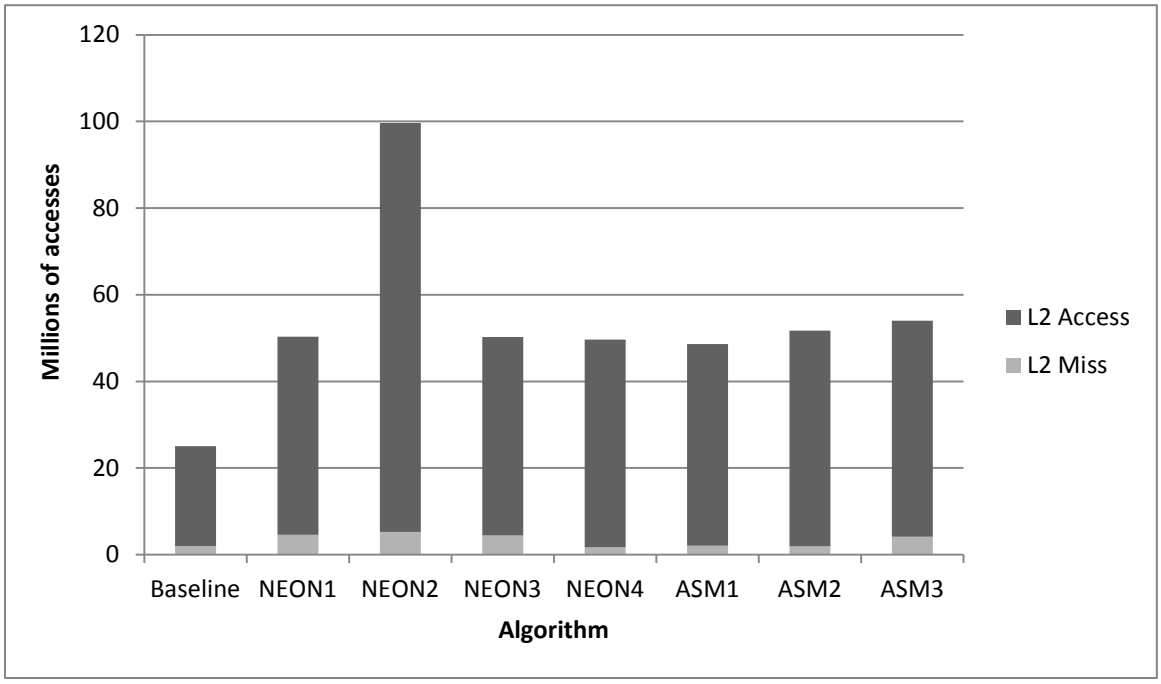


Figure 5.8: Distortion Algorithm's L2 Cache Events

cache with the data expected to be accessed next. This can save a cache miss and increase performance. With ASM3, the cache preload instruction is removed resulting in a slight increase in speedup and increase in L2 cache misses. Most likely the overhead in issuing that instruction is much greater than the performance increase it provided. In all test cases, the miss rate was kept below 10% which means that the faster L2 cache is accessed more frequently than the slower external memory.

Branch mispredictions can also have a major impact on the performance of the algorithm. With the Cortex-A8 processor, each branch misprediction causes the pipeline to empty which incurs a 13 cycle penalty. Figure 5.9 shows the number of branch mispredictions for the various test cases of the distortion algorithm. The NEON2 test case shows the most branch misses most likely due to the vectorization of the *GetDistortionVector* function. The other test cases show relatively the same amount of branch misses. In the assembly based tests the code is altered to remove unneeded branches to help reduce mispredictions. As the figure shows the assembly tests have approximately the same number of misses as the other tests. Most likely the branch prediction hardware is able to correctly guess the direction a branch takes, and therefore, the alternations do not affect the branch mispredictions. The main flow of the program does not change much between the different tests. The same number of

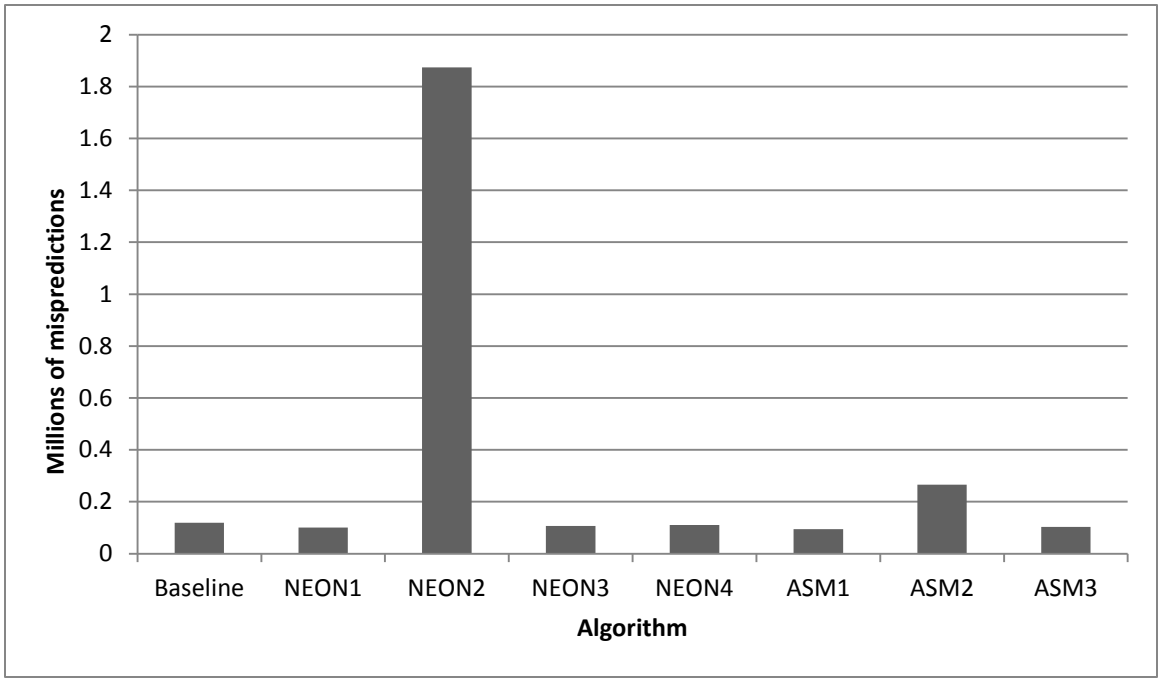


Figure 5.9: Distortion Algorithm's Branch Mispredictions

branches and the direction of branches are about the same during all the tests. The small fluctuations of the branch misses can be due to how the hardware implements the branch prediction unit.

For optimum performance, both processors should be concurrently executing instructions at all times. The ARM and NEON coprocessors are separate from each other, and therefore, the have the ability to operate in parallel. Issuing a mix of ARM and NEON instructions is a way in which this option can be exploited. For example, SIMD load and store operations use both ARM and NEON instructions. For a SIMD load or store, the address is calculated with the ARM processor and then passed to the NEON coprocessor where the memory access occurs. The processing of an image is mostly done with SIMD instructions. So, image processing does not use the ARM and NEON coprocessor in parallel. Figure 5.10 shows the percentage of cycles that both processors are active for the various distortion test cases. The theoretical maximum is 100% which corresponds to both processors always being active. The baseline does not show any concurrent cycles because the NEON coprocessor was inactive during this time. The tests have instances where more NEON instructions are used than ARM, and instances where the opposite occurs. The NEON2 and ASM3 tests have about the same percentage of concurrent cycles. In the NEON2 test, NEON and ARM instructions are used in the *GetDistortionVector* function which helps operate the processors in parallel.

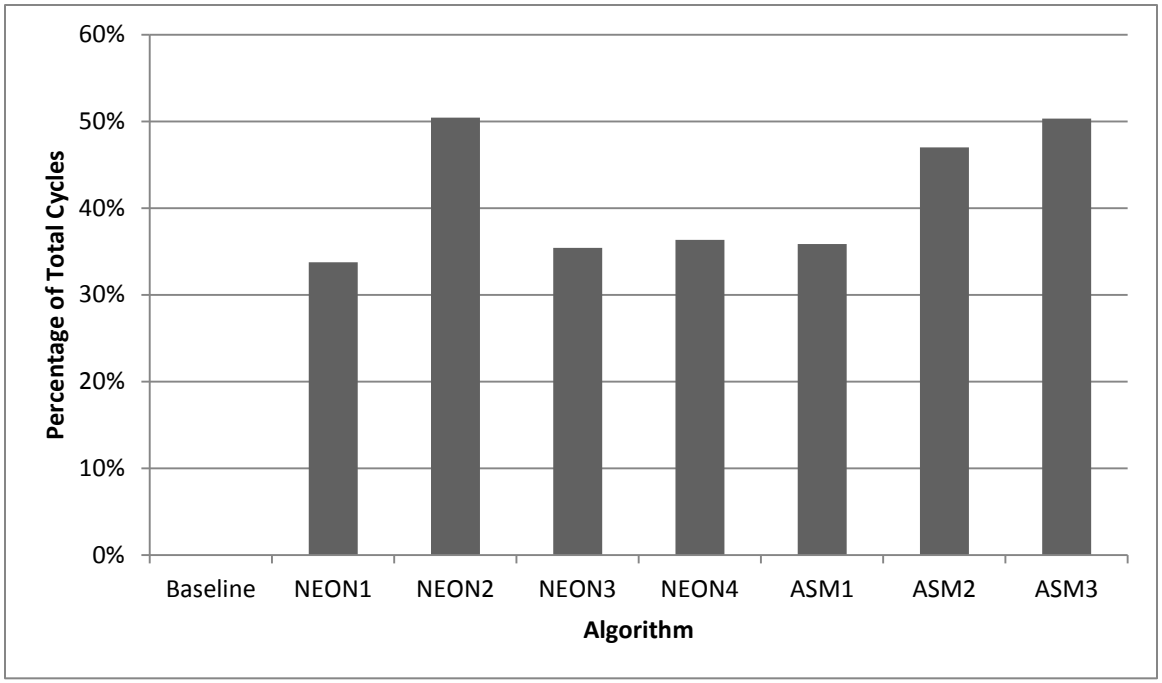


Figure 5.10: Distortion Algorithm's Both Processors Active

In the ASM2 and ASM3 test cases, some ARM instructions are replaced with NEON instructions and the image is processed at eight pixels per inner loop iteration. This resulted in more NEON instructions and more cycles that the processors are concurrently active. Moving from the NEON1 test to the NEON3 test made a small improvement in this metric. The mainly ARM instruction based *GetDistortionVector* function is placed in the middle of the mainly NEON instruction based image processing. The result is more concurrent activity of the two processors and a small increase in speedup. Although, the theoretical maximum cannot be reached, it is still important to run the processors concurrently when possible to help increase performance.

ARM and NEON instructions should also be mixed to avoid stalls to the NEON coprocessor from either a full instruction queue or a full load and store queue. Normally, one or two instructions are issued every cycle. If an instruction takes longer than one cycle to complete, the next instruction will be added to queue. Once the queue is filled, no more instructions can be issued and the processor stalls. The same occurs if too many memory accesses are requested. Figure 5.11 shows the number of cycles the NEON coprocessor stalls as a result of a full instruction or load and store queue. For best performance, this metric should be kept to a minimum. Again, the baseline does not have any stalled cycles because the NEON coprocessor is not active. The most significant change is shown between NEON1 and NEON3. Moving the

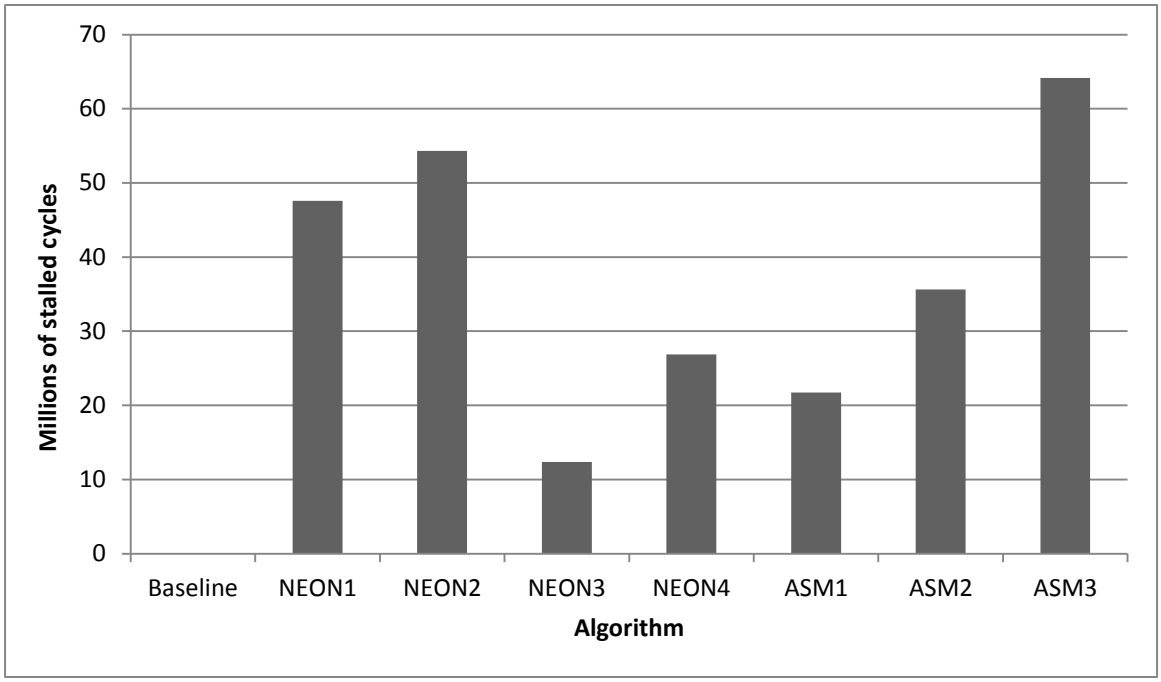


Figure 5.11: Distortion Algorithm's Full NEON Queue Stalls

GetDistortionVector function into the main image processing part of the code mixes the NEON and ARM instructions. This results in fewer stalls from a full queue because less NEON instructions are issued sequentially. In subsequent tests, ARM instructions are removed and NEON instructions are added, which causes more stalls in the NEON coprocessor. Although, measuring the stalls from a full ARM instruction or load and store queue is not possible, the ARM processor likely shows an inverse relationship to the NEON processor's queue stalls. With less ARM and more NEON instructions, the ARM processor's queue should not fill as quickly and stalls should be less prevalent. When comparing the ASM2 test to the ASM3 test, many more sequential NEON instructions are added. The increase in NEON instructions causes the queues to fill up faster and therefore the ASM3 test shows many more stalled cycles. Increasing performance can be achieved by mixing the ARM and NEON instructions which will help to reduce the number of stalls from a full NEON coprocessor queue.

Moving data from a coprocessor register, such as a NEON register, to an ARM register is a costly process. The move takes 20 cycles to complete and stalls the ARM pipeline while the data is being transferred. Figure 5.12 shows the number of cycles the ARM processor stalls while waiting for data from a coprocessor. The distortion algorithm avoids these stalls by not directly transferring data from the NEON coprocessor to the ARM processor. If a transfer is needed, the data is stored by the NEON coprocessor to

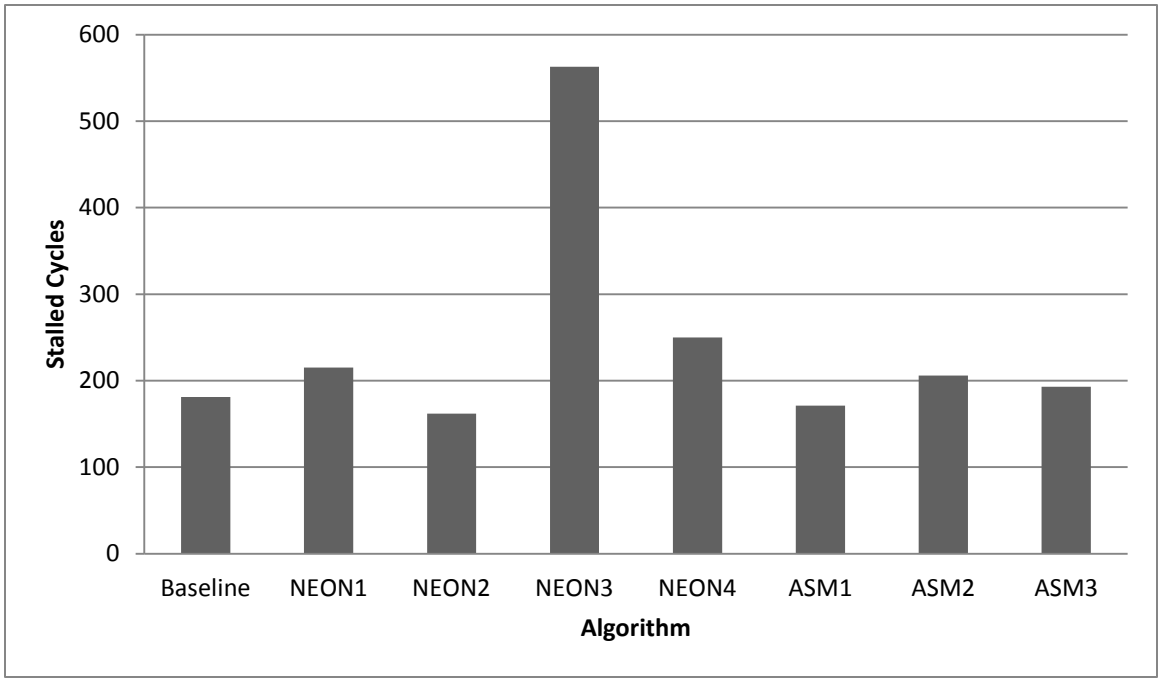


Figure 5.12: Distortion Algorithm's Coprocessor to Processor Transfer Stalls

memory and the ARM processor then loads this data. This process can still cause stalls, but it eliminates the 20 cycle penalty from a direct register to register transfer. The stalls for all the tests are very low in comparison to stalls previously discussed. The metric is likely measuring the transfers from the performance counter registers to the ARM registers. The distortion algorithm's test cases have no NEON coprocessor to ARM processor transfers to keep stalls to a minimum.

5.2.2 Other Considerations

Using 16-bit operands instead of 32-bit operands does not produce an acceptable resulting image. This doubles the amount of data that can be packed into a NEON register which should double the performance. Figure 5.13 shows the resulting image from this test, which is unacceptable. Normally, the algorithm packs the integer and fractional parts of the distortion vectors into the two halves of a 32-bit register. When using a 16-bit register, the integer and fractional parts are truncated to 8 bits and lose much of their precision. For this particular image processing algorithm, the 16-bit operands do not provide enough precision to produce an accurate result.

Using the integer and floating point functional units in parallel also do not produce an acceptable result. This test case should remove some structural hazards related to insufficient functional units. It uses the integer ALU, shift, and multiply units for four

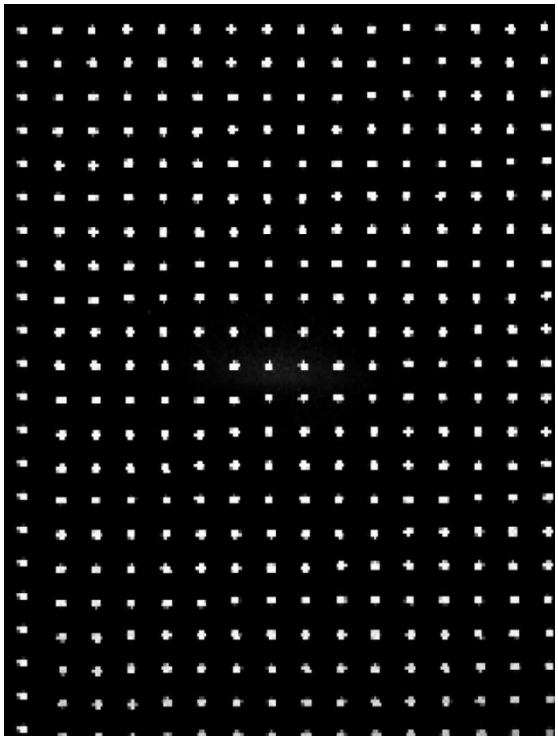


Figure 5.13: 16-bit Distortion Test Result

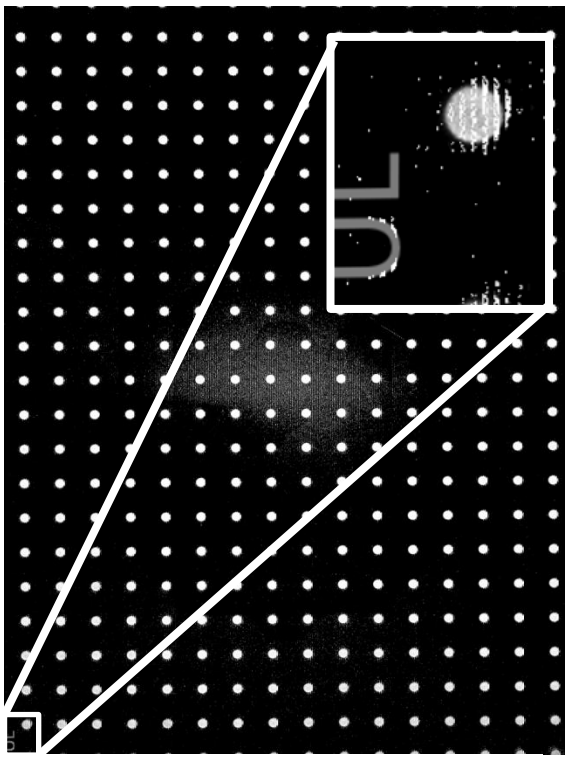


Figure 5.14: Integer and Floating Point Distortions Results

pixels, and the floating point ALU and multiply units for four pixels. The resulting image, shown in Figure 5.14, shows many artifacts from this test. Looking at the image as whole, not many faults can be seen, but when the image is enlarged, the faults become evident. The pixels processed with the integer functional units appear to be accurate, but the pixels processed with floating point functional units show many artifacts that are not acceptable for a resulting image. These are likely due to the loss in precision when moving from 32-bit integer operands to 32-bit floating point operands. Although, both operands are the same size, floating point numbers only reserve 23 bits for the fractional part. The remaining 9 bits are reserved for the sign and exponent, and therefore, the entire 32-bit floating point register cannot be used to its full precision. Also, the overhead when converting from integer to floating point may cause slowdowns in the processing and therefore the speedup may be negligible. Using both the integer and floating point functional units produces an unacceptable image and is unlikely to produce any increase in performance.

Another way to full utilize the processor is to process the image using both the ARM and NEON processors. This test processed four pixels of the image with the NEON coprocessor and one pixel with the ARM processor. So this test should be 1.25 times faster than the NEON only approach. The resulting image matches the expected result obtained by the baseline test. However, the speedup went from 2.195 in the NEON4 test to 1.468 in this test, which is a significant decrease in speed. The decreased performance is likely caused by the 20 cycle stall occurring when the ARM and NEON coprocessors access the same cache block. This occurs when the source image is loaded from memory, or the destination image is stored to memory. One alternative would be to have the ARM processor process one part of the image while the NEON coprocessor processes another part of the image. For the distortion algorithm, this is not possible due to the *GetDistortionVector* function which cannot process the image out of order. Although the resulting image is correct, this test case provided a slowdown in speed.

The final test speeds up the baseline algorithm by applying the same non-SIMD assembly based optimizations that were applied to the assembly tests. As expected, the resulting image matches the expected image from the baseline test. This test did show a speedup of 1.52 over the original baseline test. The branch miss-predictions decreased by 15% and the L2 cache miss rate decreased by 4.3%. This test shows the importance of optimizations at the assembly level because the compiler can only optimize the

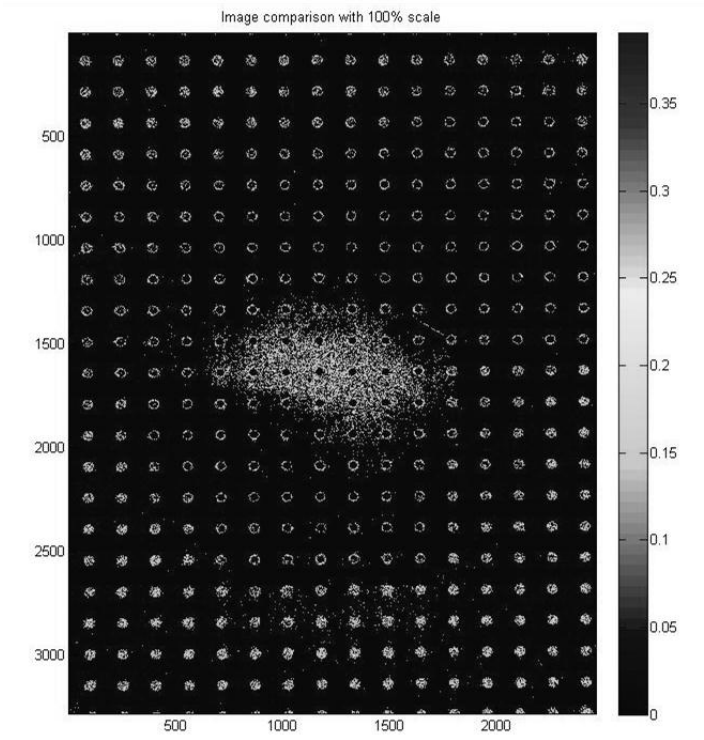


Figure 5.15: Distortion's Image Error Compared to Baseline Image

program to a certain level. Further optimization must be done manually. In this case the manual optimization shows an acceptable performance increase when modifying the assembly code. This test also shows that the NEON SIMD instructions have a major impact on the overall speedup of the algorithm. Performing the same optimizations with and without SIMD instructions shows that the SIMD instructions provide a greater performance improvement over the non-SIMD version.

The resulting images from the test cases differ from the resulting image from the baseline test. First, the error was computed by subtracting the pixel values and scaling to a 100% scale. Figure 5.15 shows the error of the output image from the NEON4 test when compared to the baseline test. The error for most of the image is zero, but parts of the image, especially where a transition occurs, have error. The maximum error is 0.391%, which is acceptable and not visible on the image. The error can likely be attributed to the way the ARM and NEON coprocessors differ in truncating or rounding of register values. The correlation coefficient can also be used to compare two images. Coefficients of greater than 0.95 are sufficient to conclude the images match . For these tests, the MATLAB *corr2* function is applied to the baseline image and the image under

test. The function is applied to each color channel and the three results are then averaged. For the main results, the computer coefficient is 0.99999 which is greater than 0.95, and therefore the images match. The 16-bit test case produced a coefficient of 0.30913 which is less than 0.95. Therefore the images are uncorrelated and which explains why the resultant image is unacceptable. The integer and floating point test case produced a higher coefficient of 0.86914, but this number is still less than 0.95 and the resultant image is unacceptable.

With a speedup of 3.090, the distortion algorithm approaches the theoretical speedup factor of four. The use of SIMD instructions and modification of the assembly code are important factors to achieve this speedup. Some test cases attempt alternate methods to fully utilize the processor, but these methods can either produce an inaccurate result image or show a decrease in performance. The results of this test show that the performance of this and possibly other image processing algorithms can significantly benefit from the use of SIMD instructions.

5.3 Power Assessment

The power is measured using both the on-board method and the external power supply method. Table 5.1 shows the current and power measurements from the idle, NEON, and non-NEON distortion algorithm tests. The NEON and non-NEON test results are from the ASM3 and baseline test cases, respectively. The table shows the two methods produced non-similar results. The on-board method shows the current to be about double the expected value, and far exceeds the 502 mA expected maximum when processing. Therefore, these results are considered not valid. The external method's current results are within the range of expected values. The processing with the NEON instructions uses about 3.1% more power than the baseline processing. This can be attributed to the NEON coprocessor being in a low power mode when no NEON instructions are issued. Although, the NEON processing requires more power, the energy used during the entire image processing time is less because the execution time is shorter.

The power results from the bilinear interpolation, shown in Table 5.2, are similar to the distortion algorithm's results. In both cases using the NEON and ARM coprocessors uses more power than the ARM processor only. The NEON1 and NEON2 test cases use 120% and 40% more power than the baseline test, respectively. Although, the NEON1 test case uses more power, it completes in less time. Therefore, the overall energy

Table 5.1: Distortion Algorithm's Power Consumption

Test	Method	Current (mA)	Power (W)
Idle	On-board	494	2.39
Processing with NEON	On-board	677	3.20
Idle	External	270	1.35
Processing with NEON	External	330	1.65
Processing without NEON	External	320	1.60

Table 5.2: Bilinear Interpolation Algorithm's Power Consumption

Test	Method	Current (mA)	Power (W)
Idle	External	330	1.65
Baseline	External	380	1.9
Processing with NEON1	External	440	2.2
Processing with NEON2	External	400	2.0

consumed will be equal or less in the baseline test case.

Both algorithms show enabling the NEON coprocessor uses more power than not enabling it. However, the decreased processing time should keep the overall power consumption approximately the same.

5.4 Contributions

The results show that using SIMD instructions can provide a significant speedup to image processing algorithms. The speedup can only be reached when processing multiple pixels or colors at a time, which is possible in many algorithms. The use of SIMD instructions was tested on a BeagleBone prototyping board containing a TI AM3359 Cortex-A8 processor with a NEON SIMD coprocessor. The bilinear interpolation and distortion algorithms were chosen for testing because they are able to process multiple pixels or colors simultaneously. Using SIMD intrinsic functions for the GNU ARM compiler, the speed up both algorithms were increased by a factor of about two over the

non-SIMD test cases. Furthermore, the distortion algorithm achieved a speedup of over three times after modifications were made to the assembly code. When an algorithm normally takes ten or more seconds to complete, this speedup can be significant and provide a faster experience to the end user.

Although, neither of the algorithms achieved its theoretical maximum speedup, many lessons were learned about implementing NEON SIMD instructions. First, correctly using SIMD instructions is important to maximize the speedup. Some algorithms may be difficult to parallelize, and therefore, additional instructions may have to be used to move the data between registers and lanes. These added instructions may cause the SIMD code to perform slower than the non-SIMD code. This was shown in both the bilinear interpolation algorithm's NEON2 and distortion algorithm's NEON2 test cases. Second, a mix of ARM and NEON instructions should be used when possible. This will help avoid stalls related to a full NEON coprocessor instruction or memory queue and it will run the ARM and NEON coprocessors more concurrently. The former must be avoided so instructions can keep being issued and stalled cycles avoided resulting in more processing time. Running the coprocessors concurrently can double the number of instructions issued each clock cycle, which should decrease the time needed to execute the algorithm. Thus, the use of SIMD instructions must be done carefully so a performance benefit can be achieved.

Lessons were also learned about increasing the performance of the algorithms with non-SIMD techniques. First, cache accesses and cache misses must be kept to a minimum. Each cache access means the data is not in the processor's registers and must be loaded from the cache, which takes time. Each cache miss means the cache does not have the requested data and must access it from a higher hierarchical memory level, which requires even more time. Second, cache preload instructions can be used to reduce cache miss rates. The L2 cache preload instruction was able to decrease the cache miss rate in both algorithms, which should increase performance. The tests using the intrinsic functions showed an increased speed, but the assembly tests showed a decreased speed, likely due to the time required to issue the instruction. Therefore, the cache preload instruction can be beneficial in some cases, but harmful in others. Finally, branches should be eliminated when possible to help reduce the number of branch mispredictions. With the ARM processor, each branch misprediction incurs a 13 cycle penalty. Although not all mispredictions can be eliminated, minimizing them can greatly increase the speed.

Other techniques to increase the speedup were unsuccessful for the distortion algorithm, but may work for other image processing algorithms. The first attempt was to use 16-bit operands instead of 32-bit operands. Halving the precision of the calculations caused an incorrect resulting image, but in other algorithms where 16-bit operations are permitted, this technique can provide a speedup. The second attempt was to use the integer and floating point functional units in parallel. Again, the reduced precision resulted in an incorrect image, and the conversion between floating point numbers and integers caused a decrease in speed. This technique may work with other algorithms, but one must consider the reduced precision and time for conversions. The final attempt was to concurrently use the ARM and NEON coprocessors to process the image. A correct resulting image was created, but the attempt showed a decrease in speed, which is likely due to the coprocessors accessing the same cache line. Other algorithms would likely see the same results from this technique. Therefore, this technique shouldn't be used.

Using SIMD instructions can benefit image processing algorithms, but they are difficult to implement. To achieve some speedup, the SIMD intrinsic functions can be used within an existing code. However, these functions require time to implement and an understanding of how the code has to be parallelized. If higher speedup is needed, then the code can be modified at the assembly level. Modifying the assembly code requires more time and a greater understanding of how the algorithm works. The use of vectorizing compilers can reduce the time and understanding level required. Both tested algorithms used the automatic vectorization, but the compiler was not able to find any vectorizable loops. Specifically, the bilinear interpolation algorithm can likely be sped up more, but the distortion algorithm is close to its maximum speedup. The bilinear interpolation algorithm only used SIMD instructions with intrinsic functions and no modified assembly code. Many of the lessons learned from the distortion algorithm's assembly test cases could be applied to it. Minor improvements can likely be achieved by reordering instructions and by register renaming to reduce data dependencies. For both algorithms, using alternative compilers may result in a better optimization and produce even greater speedups.

Figure 5.16 shows the steps we believe one should follow when attempting to use SIMD instructions to speedup other image processing algorithms. At the end of each step the speedup should be checked to ensure an increase has occurred. Each step is self-explanatory.

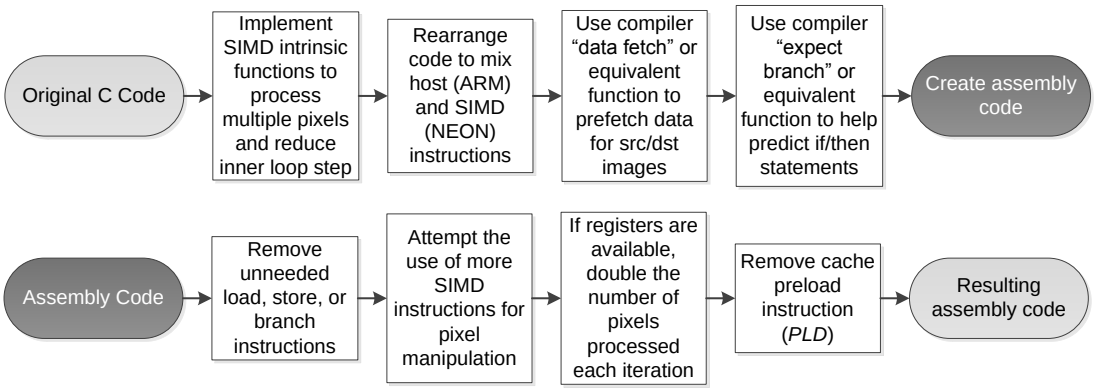


Figure 5.16: Recommended steps to follow in the use of SIMD Instructions

Chapter 6
Conclusions

This thesis has shown that through the proper use of SIMD instructions and assembly coding, image processing algorithms can be sped up by a factor of more than three. Previous works have only achieved speedups of up to 2.7 times with simpler algorithms.

The results and methods presented can be extrapolated to other image processing algorithms. The speedup can only be reached when processing multiple pixels or colors at a time, which is possible in a majority of image processing algorithms.

References

[1] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Waltham, Massachusetts: Mogan Kaufmann, 2012, pp. 282-288.

[2] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong and D. A. Padua, "An Evaluation of Vectorizing Compilers," in *International Conference on Parallel Architectures and Compilation Techniques*, Urbana, Illinois, 2011.

[3] ARM, "Cortex-A Series Overview," 2012. [Online]. Available: http://www.arm.com/products/processors/cortex-a/index.php.

[4] ARM, "Cortex-A Series: Programmer's Guide," August 2011. [Online]. Available: https://silver.arm.com/download/Software/BX100-DA-98001-r0p0-01rel2/DEN0013C_cortex_a_series_PG.pdf.

[5] S. Antao and L. Sousa, "Exploiting SIMD Extensions for Linear Image Processing with OpenCL," in *International Conference on Computer Design*, Lisbon, 2010.

[6] ARM, "Overview of NEON Technology," 2012. [Online]. Available: http://www.arm.com/products/processors/technologies/neon.php.

[7] T. Rintaluoma and O. Silven, "SIMD Performance in Software Based Mobile," in *International Conference on Embedded Computer Systems*, Oulu, 2010.

[8] Intel, "Using MMX Instructions to Implement Bilinear Interpolation of Video RGB Values," Intel Corporation, 1996.

[9] P. Larsson and E. Palmer, "Image Processing Acceleration Techniques using Intel Streaming SIMD Extensions and Intel Advanced Vector Extensions," Intel, 2009.

[10] BeagleBoard, "BeagleBone System Reference," February 2012. [Online]. Available: http://beagleboard.org/static/beaglebone/latest/Docs/Hardware/BONE_SRM.pdf.

[11] Texas Instruments, "AM335x Technical Reference Manual," June 2012. [Online]. Available: http://www.ti.com/lit/ug/spruh73f/spruh73f.pdf.

[12] ARM, "Cortex-A8 Technical Reference Manual," May 2010. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf.

[13] GNU, "GNU Compiler User Guide," 2010. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/index.html.

[14] E. Sobole, "Bilinear enlarge with NEON," May 2011. [Online]. Available: http://pulsar.webshaker.net/2011/05/25/bilinear-enlarge-with-neon/.

[15] P. Arbelaez, C. Fowlkes and D. Martin, "The Berkeley Segmentation Dataset and Benchmark," Berkley EECS, June 2007. [Online]. Available: http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/.

[16] E. Yen and R. Johnston, "The Ineffectiveness of the Correlation Coefficient for Image Comparisons," Los Alamos National Laboratory, Los Alamos, 2007.

Appendix A

Performance Counter Code

```
1   #define OUT_FILE "perf.csv"
2
3   #include <stdlib.h>
4   #include <stdio.h>
5   #include <time.h>
6   #include "bilin.h"
7
8   //intialize globals
9   FILE *perf;
10  clock_t perf_start;
11  uint32 event[4];
12  bool enable = FALSE;
13
14  void perf_init(uint32 g_perf[4]){
15          for(int i = 0; i < 4; i++){
16                  event[i] = g_perf[i];
17                  if(event[i] != 0) enable = TRUE;
18          }
19          if(enable){
20                  //reset counters and overflow
21                  asm("MCR p15, 0, %0, c9, c12, 0" :: "r"(0x41002007));
22                  asm("MCR p15, 0, %0, c9, c12, 3" :: "r"(0x8000000f));
23
24                  //setup output file
25                  perf = fopen(OUT_FILE, "w");
26                  fprintf(perf,"%s, %s, %d, %d, %d, %d,
    %s\n","Name","time",event[0],event[1],event[2],event[3],"V Status");
27
28                  //setup events
29                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000000));
30                  asm("MCR p15, 0, %0, c9, c13, 1" :: "r"(event[0]));
31                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000001));
32                  asm("MCR p15, 0, %0, c9, c13, 1" :: "r"(event[1]));
33                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000002));
34                  asm("MCR p15, 0, %0, c9, c13, 1" :: "r"(event[2]));
35                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000003));
36                  asm("MCR p15, 0, %0, c9, c13, 1" :: "r"(event[3]));
37
38                  //start counters
39                  asm("MCR p15, 0, %0, c9, c12, 1" :: "r"(0x8000000f));
40
41                  //get start time
42                  perf_start = clock();
43          }
44
45  }
46
47  void perf_checkpoint(char* name, uint32 disable){
48          //function will only run when disable is 0
49          if(disable == 0 && enable){
50                  unsigned int value[5];
51                  float time_dif;
52
53                  //get current time
54                  time_dif=(float)(clock() - perf_start)/CLOCKS_PER_SEC;
55
56                  //get perf counter values including CC and overflow
57                  //asm("MRC p15, 0, %0, c9, c13, 0" : "=r"(value[0]));
58                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000000));
59                  asm("MRC p15, 0, %0, c9, c13, 2" : "=r"(value[0]));
60                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000001));
61                  asm("MRC p15, 0, %0, c9, c13, 2" : "=r"(value[1]));
```

```
62                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000002));
63                  asm("MRC p15, 0, %0, c9, c13, 2" : "=r"(value[2]));
64                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000003));
65                  asm("MRC p15, 0, %0, c9, c13, 2" : "=r"(value[3]));
66                  asm("MRC p15, 0, %0, c9, c12, 3" : "=r"(value[4]));
67
68                  //Print values to file
69                  fprintf(perf,"%s, %.3f, %u, %u, %u, %u,
    0x%X\n",name,time_dif,value[0],value[1],value[2],value[3],value[4]);
70          }
71
72  }
73
74  void perf_exit(){
75          if(enable){
76                  unsigned int value[5];
77                  float time_dif;
78
79                  //get current time
80                  time_dif=(float)(clock() - perf_start)/CLOCKS_PER_SEC;
81
82                  //get perf counter values including CC and overflow
83                  //asm("MRC p15, 0, %0, c9, c13, 0" : "=r"(value[0]));
84                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000000));
85                  asm("MRC p15, 0, %0, c9, c13, 2" : "=r"(value[0]));
86                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000001));
87                  asm("MRC p15, 0, %0, c9, c13, 2" : "=r"(value[1]));
88                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000002));
89                  asm("MRC p15, 0, %0, c9, c13, 2" : "=r"(value[2]));
90                  asm("MCR p15, 0, %0, c9, c12, 5" :: "r"(0x00000003));
91                  asm("MRC p15, 0, %0, c9, c13, 2" : "=r"(value[3]));
92                  asm("MRC p15, 0, %0, c9, c12, 3" : "=r"(value[4]));
93
94                  //Print values to file
95                  fprintf(perf,"%s, %.3f, %u, %u, %u, %u,
    0x%X\n","END",time_dif,value[0],value[1],value[2],value[3],value[4]);
96
97                  //close file
98                  fclose(perf);
99          }
100
101 }
```

Appendix B

Bilinear Interpolation Baseline Code

```
1   //*************************************************************************
2   //                              Stretch function
3   //                     Algorithm taken from and adapted:
4   //    http://pulsar.webshaker.net/2011/05/25/bilinear-enlarge-with-neon/
5   //*************************************************************************
6
7   int stretch_c(unsigned int *bSrc, unsigned int *bDst, int wDst, int hDst, bool
    test, int mult)
8   {
9    unsigned int wSrc = INPUT_SIZEy;
10   unsigned int hSrc = INPUT_SIZEx;
11   unsigned int *Dst;
12   unsigned int wStepFixed16b, hStepFixed16b, wCoef, hCoef, x, y;
13   unsigned int pixel1, pixel2, pixel3, pixel4;
14   unsigned int pixela, pixelb;
15   unsigned int hc1, hc2, wc1, wc2, offsetX, offsetY;
16   unsigned int c, b, a, i;
17   unsigned int a1, a2, a3;
18   unsigned int hca, wca;
19   unsigned int error = 0;
20   bool passed = 1;
21
22   wStepFixed16b = ((wSrc - 1) << 16) / (wDst - 1);
23   hStepFixed16b = ((hSrc - 1) << 16) / (hDst - 1);
24
25   for(i=mult;i>0;i--){
26    Dst=bDst;
27    hCoef = 0;
28
29
30    for (y = 0 ; y < hDst ; y++) //begin y-loop
31    {
32     hc2 = (hCoef >> 9) & 127;
33     hc1 = 128 - hc2;
34     offsetY = (hCoef >> 16);
35     wCoef = 0;
36
37     for (x = 0 ; x < wDst ; x++) //begin x-loop
38     {
39      offsetX = (wCoef >> 16);
40      wc2 = (wCoef >> 9) & 127;
41      wc1 = 128 - wc2;
42
43      //Each pixel is 24 bits with 3 color channels of 8 bits
44      pixel1 = *(bSrc + offsetY * wSrc + offsetX);
45      pixel2 = *(bSrc + (offsetY + 1) * wSrc + offsetX);
46      pixel3 = *(bSrc + offsetY * wSrc + offsetX + 1);
47      pixel4 = *(bSrc + (offsetY + 1) * wSrc + offsetX + 1);
48
49      a = ((((pixel1 >> 0) & 255) * hc1 + ((pixel2 >> 0) & 255) * hc2) * wc1 +
50      (((pixel3 >> 0) & 255) * hc1 + ((pixel4 >> 0) & 255) * hc2) * wc2) >> 14;
51      b = ((((pixel1 >> 8) & 255) * hc1 + ((pixel2 >> 8) & 255) * hc2) * wc1 +
52      (((pixel3 >> 8) & 255) * hc1 + ((pixel4 >> 8) & 255) * hc2) * wc2) >> 14;
53      c = ((((pixel1 >> 16) & 255) * hc1 + ((pixel2 >> 16) & 255) * hc2) * wc1 +
54      (((pixel3 >> 16) & 255) * hc1 + ((pixel4 >> 16) & 255) * hc2) * wc2) >> 14;
55
56      *Dst++ = (c << 16) + (b << 8) + (a);
57      wCoef += wStepFixed16b;
58     } //end x-loop
59     hCoef += hStepFixed16b;
60    } //end y-loop
61   }
62
```

```
63  //    Check for calculation to match expected result form data2.h
64   if(test){
65    Dst=bDst;
66    for(i=0;i<wDst*hDst-1 && error < 10;i++){
67     if(expected_data[i] != *Dst){
68      printf("ERROR at [%d] (%X != %X)\n",i,*Dst,expected_data[i]);
69      error++;
70     }
71     Dst++;
72    }
73    printf("%d error(s) occured.\n", error);
74    if(error>0) passed=0;
75   }
76   return(passed);
77  }
78
```

Appendix C

Bilinear Interpolation NEON1 Code

```
1   //*************************************************************************
2   //                      Stretch function of NEON1 test case
3   //                       Algorithm taken from and adapted:
4   //    http://pulsar.webshaker.net/2011/05/25/bilinear-enlarge-with-neon/
5   //*************************************************************************
6
7   int stretch_neon(unsigned int *bSrc, unsigned int *bDst, int wDst, int hDst, bool
    test, int mult)
8   {
9    unsigned int wSrc = INPUT_SIZEy;
10   unsigned int hSrc = INPUT_SIZEx;
11   unsigned int *Dst;
12   unsigned int wStepFixed16b, hStepFixed16b, wCoef, hCoef, x, y;
13   unsigned int hc1, hc2, wc1, wc2, offsetX, offsetY;
14   unsigned int i;
15   unsigned int error = 0;
16   bool passed = 1;
17   uint16x8_t hc2vec, hc1vec;
18   uint16x4_t wc2vec, wc1vec;
19   uint32x4_t res1, res2;
20   uint16x8_t pixelavec, pixelbvec;
21   uint32x2_t destvec;
22
23   wStepFixed16b = ((wSrc - 1) << 16) / (wDst - 1);
24   hStepFixed16b = ((hSrc - 1) << 16) / (hDst - 1);
25
26   for(i=mult;i>0;i--){
27    Dst=bDst;
28    hCoef = 0;
29
30
31    for (y = 0 ; y < hDst ; y++) //begin y-loop
32    {
33     hc2 = (hCoef / 512) & 127;
34     hc1 = 128 - hc2;
35     hc2vec = vdupq_n_u16(hc2);
36     hc1vec = vdupq_n_u16(hc1);
37     offsetY = (hCoef / 65536);
38     wCoef = 0;
39
40     for (x = 0 ; x < wDst ; x++) //begin x-loop
41     {
42      offsetX = (wCoef / 65536);
43      wc2 = (wCoef / 512) & 127;
44      wc1 = 128 - wc2;
45      wc1vec = vdup_n_u16(wc1);
46      wc2vec = vdup_n_u16(wc2);
47
48      //Each pixel is 24 bits with 3 color channels of 8 bits
49      //load pixel3|pixel1
50      pixelavec = vmovl_u8(vreinterpret_u8_u32(vld1_u32(bSrc + offsetY * wSrc +
    offsetX)));
51      //preload next likely source into cache
52      __builtin_prefetch(bSrc + offsetY * wSrc + offsetX + 2, 0, 2);
53      //load pixel4|pixel2
54      pixelbvec = vmovl_u8(vreinterpret_u8_u32(vld1_u32(bSrc + (offsetY + 1) * wSrc
    + offsetX)));
55      //preload next likely source into cache
56      __builtin_prefetch(bSrc + (offsetY + 1) * wSrc + offsetX + 2, 0, 2);
57
58      pixelavec = vmulq_u16(pixelavec, hc1vec);
59      pixelbvec = vmulq_u16(pixelbvec, hc2vec);
60      pixelavec = vaddq_u16(pixelavec, pixelbvec);
```

```
61      res1 = vmull_u16(vget_high_u16(pixelavec), wc2vec);
62      res2 = vmull_u16(vget_low_u16(pixelavec), wc1vec);
63      res1 = vaddq_u32(res1, res2);
64      pixelavec = vcombine_u16(vshrn_n_u32(res1, 14), vshrn_n_u32(res1, 14));
65
66      destvec = vreinterpret_u32_u8(vmovn_u16(pixelavec));
67      vst1_lane_u32(Dst++, destvec, 0);
68
69      wCoef += wStepFixed16b;
70     } //end x-loop
71     hCoef += hStepFixed16b;
72    } //end y-loop
73   }
74
75   if(test){
76    Dst=bDst;
77    for(i=0;i<wDst*hDst-1 && error < 10;i++){
78     if(expected_data[i] != *Dst){
79      printf("ERROR at [%d] (%X != %X)\n",i,*Dst,expected_data[i]);
80      error++;
81     }
82     Dst++;
83    }
84    printf("%d error(s) occured.\n", error);
85    if(error>0) passed=0;
86   }
87   return(passed);
88  }
```

Appendix D

Bilinear Interpolation NEON2 Code

```
1   //*************************************************************************
2   //                      Stretch function of NEON2 test case
3   //                       Algorithm taken from and adapted:
4   //    http://pulsar.webshaker.net/2011/05/25/bilinear-enlarge-with-neon/
5   //*************************************************************************
6
7   int stretch_neon2(unsigned int *bSrc, unsigned int *bDst, int wDst, int hDst, bool
    test, int mult)
8   {
9    unsigned int wSrc = INPUT_SIZEy;
10   unsigned int hSrc = INPUT_SIZEx;
11   unsigned int *Dst;
12   unsigned int wStepFixed16b, hStepFixed16b, wCoef, hCoef, x, y;
13   unsigned int hc1, hc2, wc1[4], wc2[4], offsetX[4], offsetY;
14   unsigned int i;
15   unsigned int error = 0;
16   bool passed = 1;
17   uint32x4_t hc2vec, hc1vec;
18   uint32x4_t wc2vec, wc1vec;
19   uint32x4_t pixel1vec, pixel2vec, pixel3vec, pixel4vec;
20   uint32x4_t destvec;
21   uint32x4_t avec, bvec, cvec;
22   uint32x4_t FFmask = vdupq_n_u32(255);
23
24   wStepFixed16b = ((wSrc - 1) << 16) / (wDst - 1);
25   hStepFixed16b = ((hSrc - 1) << 16) / (hDst - 1);
26
27   for(i=mult;i>0;i--){
28    Dst=bDst;
29    hCoef = 0;
30
31
32    for (y = 0 ; y < hDst ; y++) //begin y-loop
33    {
34     hc2 = (hCoef / 512) & 127;
35     hc2vec = vdupq_n_u32(hc2);
36     // hc1 = 128 - hc2;
37     hc1vec = vdupq_n_u32(128 - hc2);
38     offsetY = hCoef / 65536;
39     wCoef = 0;
40
41     for (x = 0 ; x < wDst ; x+=4) //begin x-loop
42     {
43      for(int z=0; z<4; z++) //begin i-loop
44      {
45       offsetX[z] = (wCoef / 65536);
46       wc2[z] = (wCoef / 512) & 127;
47       wc1[z] = 128 - wc2[z];
48       wCoef += wStepFixed16b;
49      } //end i-loop
50
51      wc2vec = vld1q_u32(wc2);
52      wc1vec = vld1q_u32(wc1);
53
54      //Each pixel is 24 bits with 3 color channels of 8 bits
55      //preload next likely source into cache
56      __builtin_prefetch(bSrc + offsetY * wSrc + offsetX[3] + 2, 0, 2);
57      pixel1vec = vld1q_lane_u32(bSrc + offsetY * wSrc + offsetX[0], pixel1vec, 0);
58      pixel1vec = vld1q_lane_u32(bSrc + offsetY * wSrc + offsetX[1], pixel1vec, 1);
59      pixel1vec = vld1q_lane_u32(bSrc + offsetY * wSrc + offsetX[2], pixel1vec, 2);
60      pixel1vec = vld1q_lane_u32(bSrc + offsetY * wSrc + offsetX[3], pixel1vec, 3);
61
```

```
62      pixel3vec = vld1q_lane_u32(bSrc + offsetY * wSrc + offsetX[0] + 1, pixel3vec,
    0);
63      pixel3vec = vld1q_lane_u32(bSrc + offsetY * wSrc + offsetX[1] + 1, pixel3vec,
    1);
64      pixel3vec = vld1q_lane_u32(bSrc + offsetY * wSrc + offsetX[2] + 1, pixel3vec,
    2);
65      pixel3vec = vld1q_lane_u32(bSrc + offsetY * wSrc + offsetX[3] + 1, pixel3vec,
    3);
66
67      //preload next likely source into cache
68      __builtin_prefetch(bSrc + (offsetY + 1) * wSrc + offsetX[3] + 2, 0, 2);
69      pixel2vec = vld1q_lane_u32(bSrc + (offsetY + 1) * wSrc + offsetX[0],
    pixel2vec, 0);
70      pixel2vec = vld1q_lane_u32(bSrc + (offsetY + 1) * wSrc + offsetX[1],
    pixel2vec, 1);
71      pixel2vec = vld1q_lane_u32(bSrc + (offsetY + 1) * wSrc + offsetX[2],
    pixel2vec, 2);
72      pixel2vec = vld1q_lane_u32(bSrc + (offsetY + 1) * wSrc + offsetX[3],
    pixel2vec, 3);
73
74      pixel4vec = vld1q_lane_u32(bSrc + (offsetY + 1) * wSrc + offsetX[0] + 1,
    pixel4vec, 0);
75      pixel4vec = vld1q_lane_u32(bSrc + (offsetY + 1) * wSrc + offsetX[1] + 1,
    pixel4vec, 1);
76      pixel4vec = vld1q_lane_u32(bSrc + (offsetY + 1) * wSrc + offsetX[2] + 1,
    pixel4vec, 2);
77      pixel4vec = vld1q_lane_u32(bSrc + (offsetY + 1) * wSrc + offsetX[3] + 1,
    pixel4vec, 3);
78
79      avec =
    vshrq_n_u32(vaddq_u32(vmulq_u32(vaddq_u32(vmulq_u32(vandq_u32(pixel1vec, FFmask),
    hc1vec), vmulq_u32(vandq_u32(pixel2vec, FFmask), hc2vec)), wc1vec),
    vmulq_u32(vaddq_u32(vmulq_u32(vandq_u32(pixel3vec, FFmask), hc1vec),
    vmulq_u32(vandq_u32(pixel4vec, FFmask), hc2vec)), wc2vec)), 14);
80
81      bvec =
    vshrq_n_u32(vaddq_u32(vmulq_u32(vaddq_u32(vmulq_u32(vandq_u32(vshrq_n_u32(pixel1ve
    c, 8), FFmask), hc1vec), vmulq_u32(vandq_u32(vshrq_n_u32(pixel2vec, 8), FFmask),
    hc2vec)), wc1vec), vmulq_u32(vaddq_u32(vmulq_u32(vandq_u32(vshrq_n_u32(pixel3vec,
    8), FFmask), hc1vec), vmulq_u32(vandq_u32(vshrq_n_u32(pixel4vec, 8), FFmask),
    hc2vec)), wc2vec)), 14);
82
83      cvec =
    vshrq_n_u32(vaddq_u32(vmulq_u32(vaddq_u32(vmulq_u32(vandq_u32(vshrq_n_u32(pixel1ve
    c, 16), FFmask), hc1vec), vmulq_u32(vandq_u32(vshrq_n_u32(pixel2vec, 16), FFmask),
    hc2vec)), wc1vec), vmulq_u32(vaddq_u32(vmulq_u32(vandq_u32(vshrq_n_u32(pixel3vec,
    16), FFmask), hc1vec), vmulq_u32(vandq_u32(vshrq_n_u32(pixel4vec, 16), FFmask),
    hc2vec)), wc2vec)), 14);
84
85      destvec = vaddq_u32(vaddq_u32(vshlq_n_u32(cvec, 16), vshlq_n_u32(bvec, 8)),
    avec);
86      vst1q_u32(Dst, destvec);
87      Dst += 4;
88
89     } //end x-loop
90     Dst -= 4-(wDst%4);
91     hCoef += hStepFixed16b;
92    } //end y-loop
93   }
94
95
96   if(test){
97    Dst=bDst;
98    for(i=0;i<wDst*hDst-1 && error < 10;i++){
99     if(expected_data[i] != *Dst){
100     printf("ERROR at [%d] (%X != %X)\n",i,*Dst,expected_data[i]);
101     error++;
102    }
103    Dst++;
104   }
105   printf("%d error(s) occured.\n", error);
```

```
106   if(error>0) passed=0;
107  }
108  return(passed);
109 }
```