EE 282

Project Report

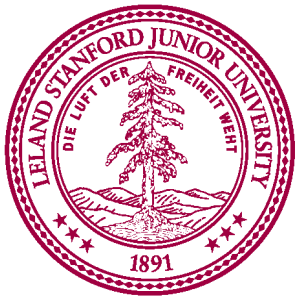
ARM / NEON / DSP Processor Programming

Spring 2010

Students:
Milad Mohammadi

CJ Steuernagel

Yu-Shiuan Huang



Introduction

The course project for EE 282 was designed to provide us the opportunity to investigate the relationship between application software and the hardware that executes it by developing an application for a modern smartphone. Our assignment was to develop software to run on the 3 execution units within the Nokia N900's multi-processor—the ARM Cortex-A8 core, the NEON vector core, and the TI DSP core—by adapting an implementation of a CJPEG algorithm to execute on each of the 3 cores. The CJPEG program we used is a complex program with a diverse set of algorithmic content, which allowed us to examine the relative strengths and weaknesses of each execution unit by observing how they compared to one another when used to execute different algorithms. It also gave us the opportunity to adapt the code in a way that took advantage of each of the 3 cores' strengths in order to optimize the overall performance. From the project we learned about how the right hardware can be beneficial for achieving software performance, and we gained insight into the techniques that can be used to optimize the performance of modern smartphone applications.

This report documents the steps we took to complete the assignment specifications and the results we achieved during the various phases of the project. The following outline indicates the structure of the report.

Contents

Nokia N900 Hardware

The Nokia N900 uses a TI OMAP3430 multiprocessor chip that contains an ARM Cortex-A8 processor with a NEON core and a separate TI DSP. Since we will be developing programs to run on each one of the execution units, it is important to understand the interconnect structure and cache hierarchy of the chip. This section summarizes the memories, caches, and execution units comprising the N900's hardware. The following table shows information about the on-chip caches:

Unit	Cache	Capacity	Block Size	Associativity	Write Policy	Eviction Policy
ARM	Level 1: Instruction	16KB	64B	4-way set associative		Random
	Level 1: Data	16KB	64B	4-way set associative		Random
	Level 2: Shared I/D	256KB	64B	8-way set associative		Random
DSP	Level 1: Instruction	0KB - 32KB	32B	Direct-mapped	N/A	N/A
	Level 1: Data	0KB - 32KB	4B	2-way set associative	Write-back	LRU
	Level 2: Shared I/D	0KB - 64KB	128B	4-way set associative	Write-back	LRU

Table 1. TI OMAP3430 chip caches

The following figure is a diagram of the interconnect structure on the chip:

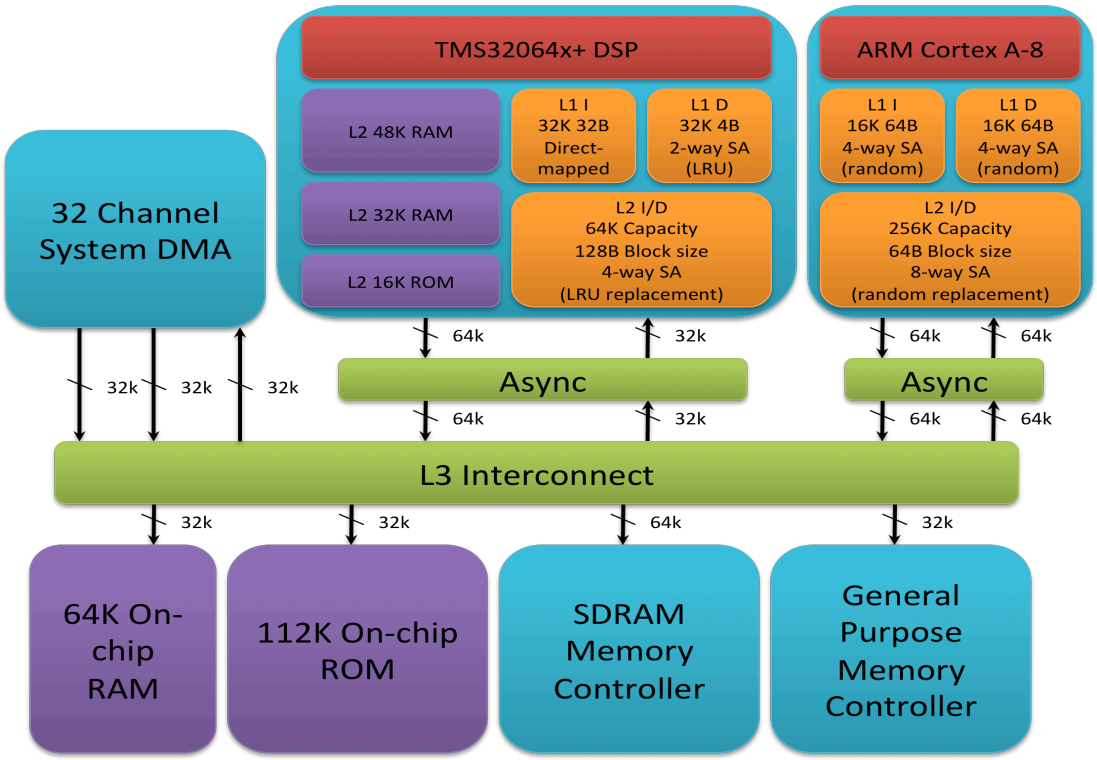


Figure 1. Block diagram of chip memories and execution units

ARM

The first part of the project required us to observe the performance of the CJPEG program when run on the ARM core and to optimize its performance for this execution unit. In the course of this undertaking, we became familiar with the CJPEG algorithm as well as the structure of the individual kernels, which prepared us to embark on the subsequent parts of the project. We also explored several optimizations to the program, including compiler optimizations, manually editing source code, and implementing software prefetching. Observing the performance of the optimized program provided us with some insight into the efficiency of the kernels, which helped us translate the code to run efficiently on the NEON core and the DSP in parts 2 and 3.

The first step we took was to characterize the performance of the un-optimized CJPEG code as it was provided to us when run on the ARM core. Using the provided time measurement utility, we collected measurements for the execution time of the algorithm and of the individual kernels when operated on images of various sizes. The following graph shows the execution time for three different image sizes:

Figure 2. Graph of execution time per kernel for different image sizes

From the graph, we observe that the execution time is linearly related to the image size, which tells us that all of the kernels have computational complexity O(*N*). The following figures show the execution time of the individual kernels for a single image:

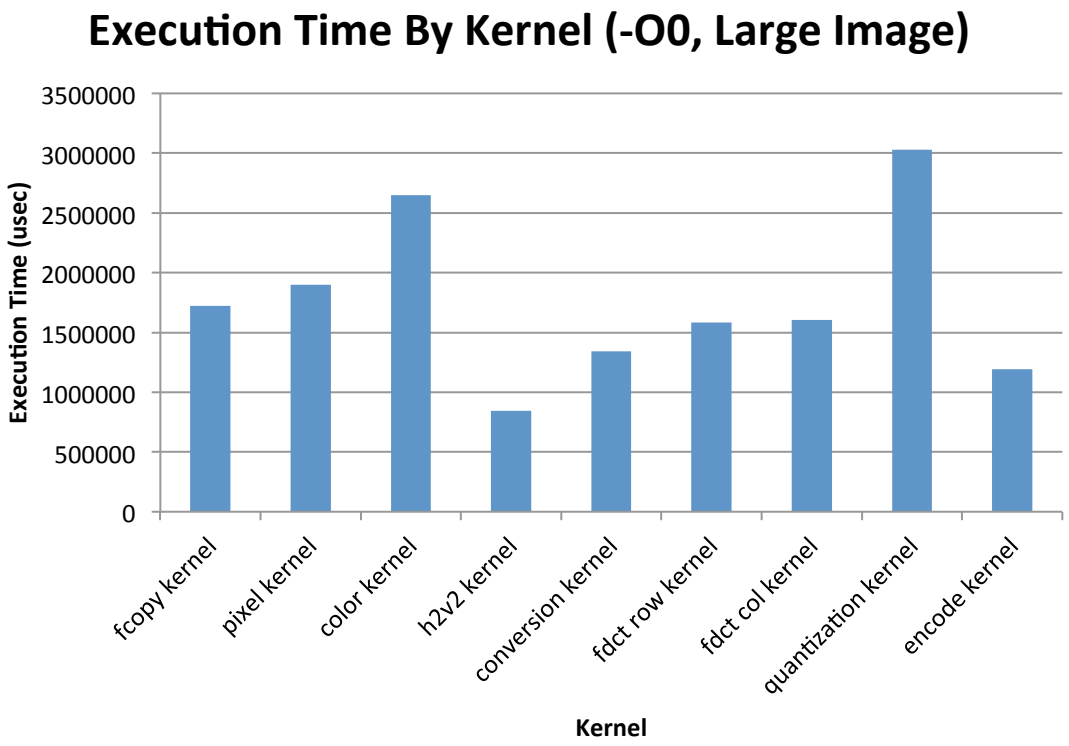


Figure 3. Execution time per kernel for a large image

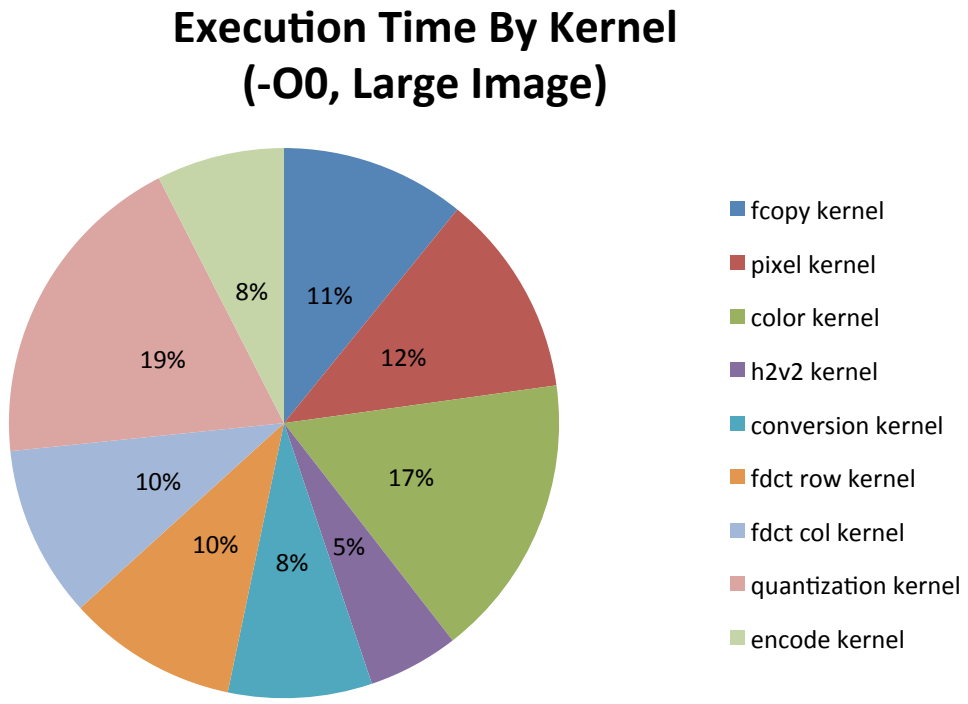


Figure 4. Breakdown of execution time by kernel

The previous chart shows that the execution times of the individual kernels are similar to one another for the most part, with the exception of a few that compose the largest fractions of the total execution time.

The next step we took was to make use of the built-in compiler optimizations to improve CJPEG's performance on ARM. We compiled the provided code under several different levels of optimization in gcc. Our results are shown in the following figures:

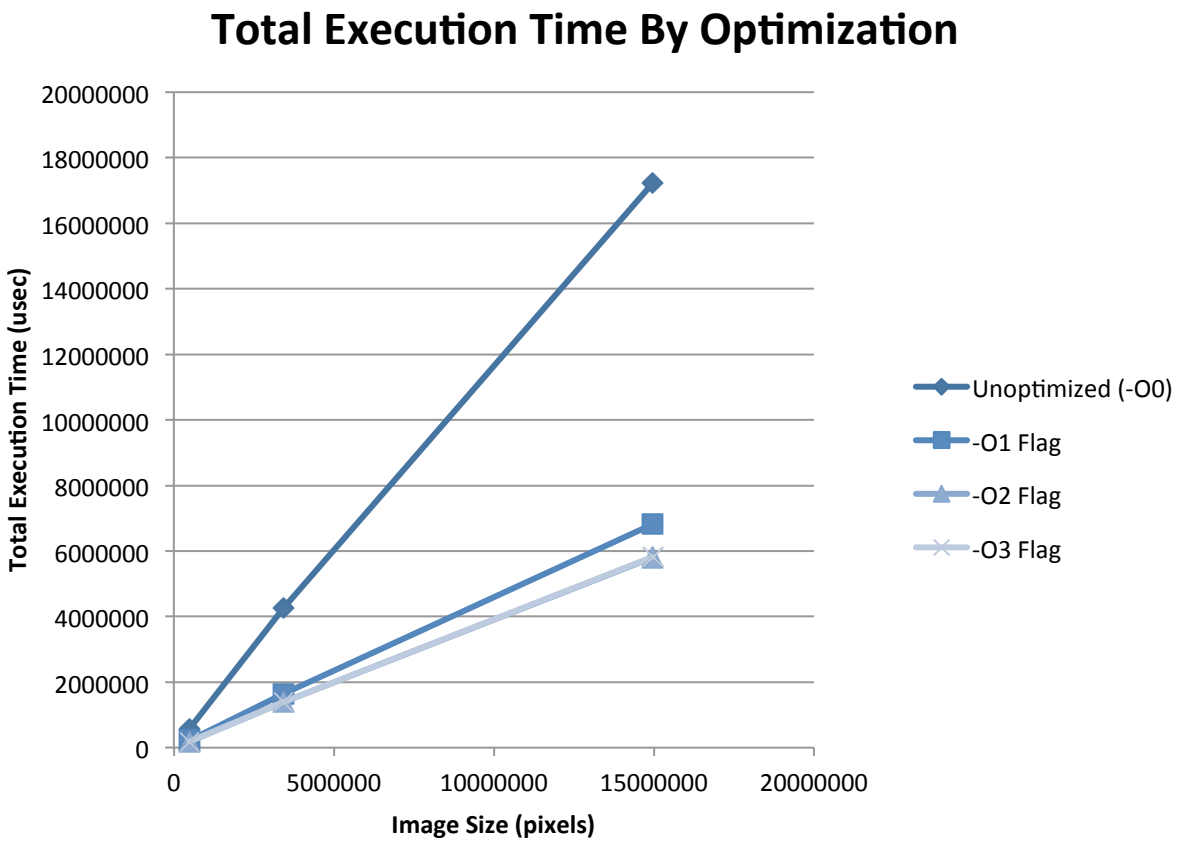


Figure 5. Execution time by optimization for three different image sizes

The above graph shows us that the performance increases significantly when compiled with the gcc optimization flags. We observe a large improvement over the un-optimized code when compiled using the –O1 flag and marginal improvements thereafter when compiled under –O2 and –O3.

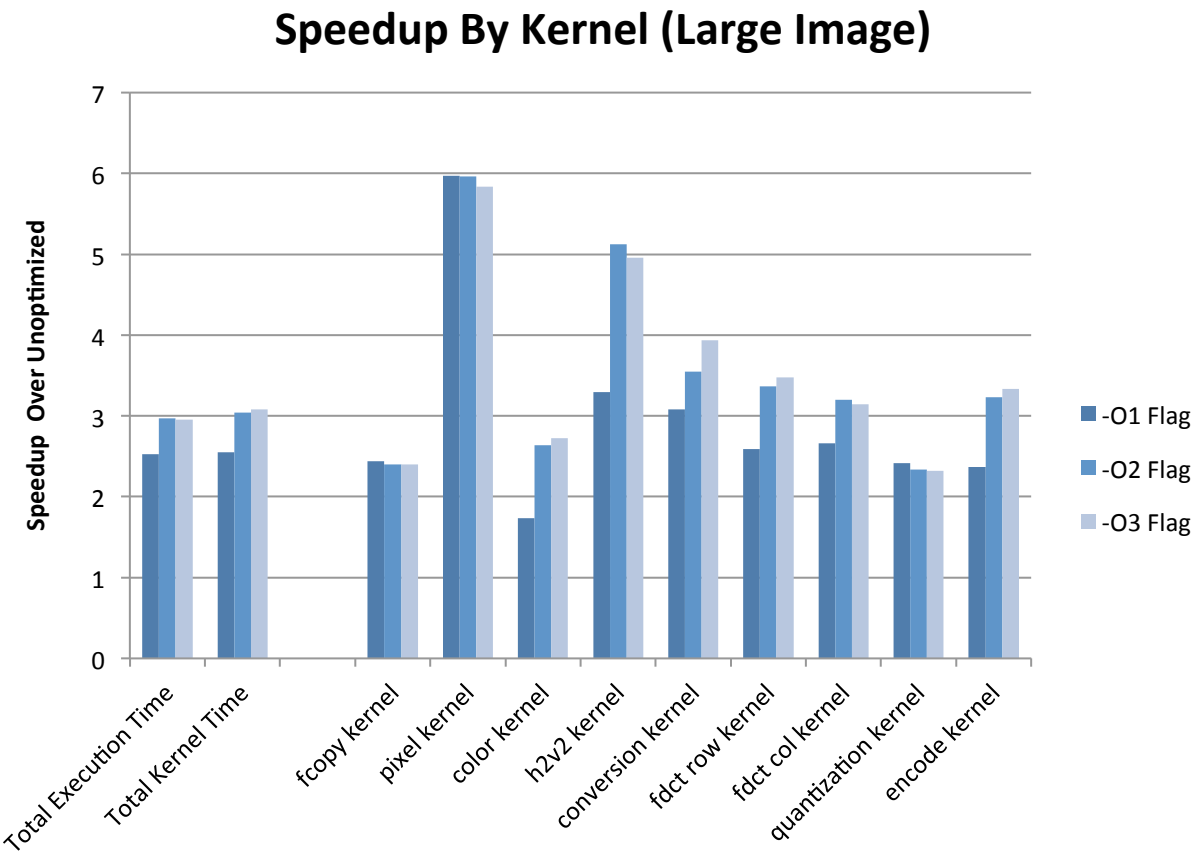


Figure 6. Speedup over the un-optimized code for different compiler optimizations for a large image

The above chart demonstrates the speedup over the un-optimized code for each kernel. We observe that every kernel experienced a significant speedup (>2) when compiled using optimization. In particular, the pixel kernel and h2v2 kernel experienced the largest speedup. This makes sense because the code for both of these kernels contained several computational inefficiencies, such as redundant arithmetic, that limited their performance when un-optimized.

We also observed that the total execution time and total kernel time was minimized when compiled using the –O3 flag in gcc. Therefore, we chose to continue to compile the code under –O3 optimization and *use this as a baseline for comparison in the subsequent parts of the project*.

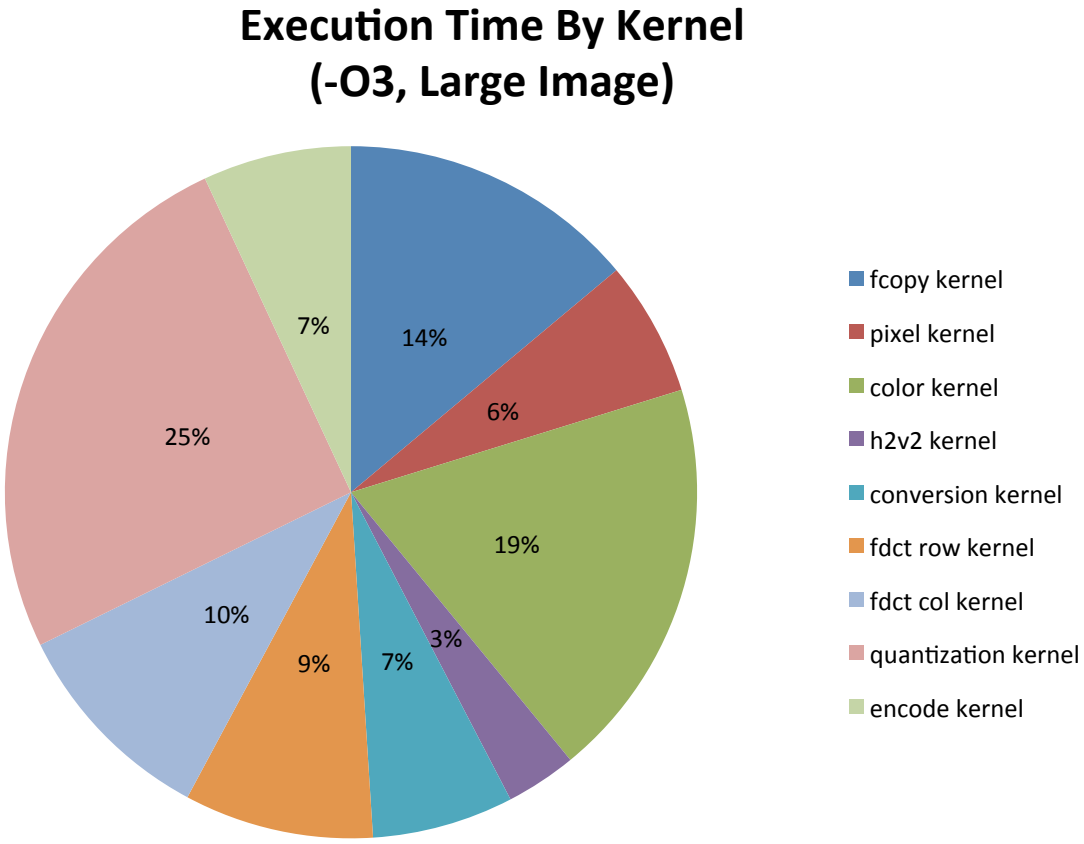


Figure 7. Breakdown of execution time by kernel for the compiler-optimized code applied to a large image

The above chart demonstrates the new breakdown of execution time by kernel. From the chart, we observe that the fcopy, color, and quantization kernels represent the major bottlenecks in performance after compiler optimization. We therefore selected these kernels for further optimization using software prefetching, which is explained later.

The third step we took was to edit the CJPEG source code to apply optimizations that the compiler may not have been able to recognize. The first part of this task involved making small edits throughout the code that targeted various inefficiencies that existed. The edits we made included the following:

- Condensing redundant arithmetic – by performing common arithmetic expressions only once instead of repeatedly, we could reduce the number of total computations
- Traversing loops in reverse order – by decrementing the iteration variable and comparing to 0 instead of incrementing and comparing to a positive number, we could reduce the number of instructions per loop iteration by using a less costly comparison
- Using the memcpy() library function in fcopy kernel – since fcopy kernel merely copies data from one memory location to another, using memcpy() instead of a less efficient loop could reduce the overhead involved in the transfer

The following figure shows the results of these optimizations:

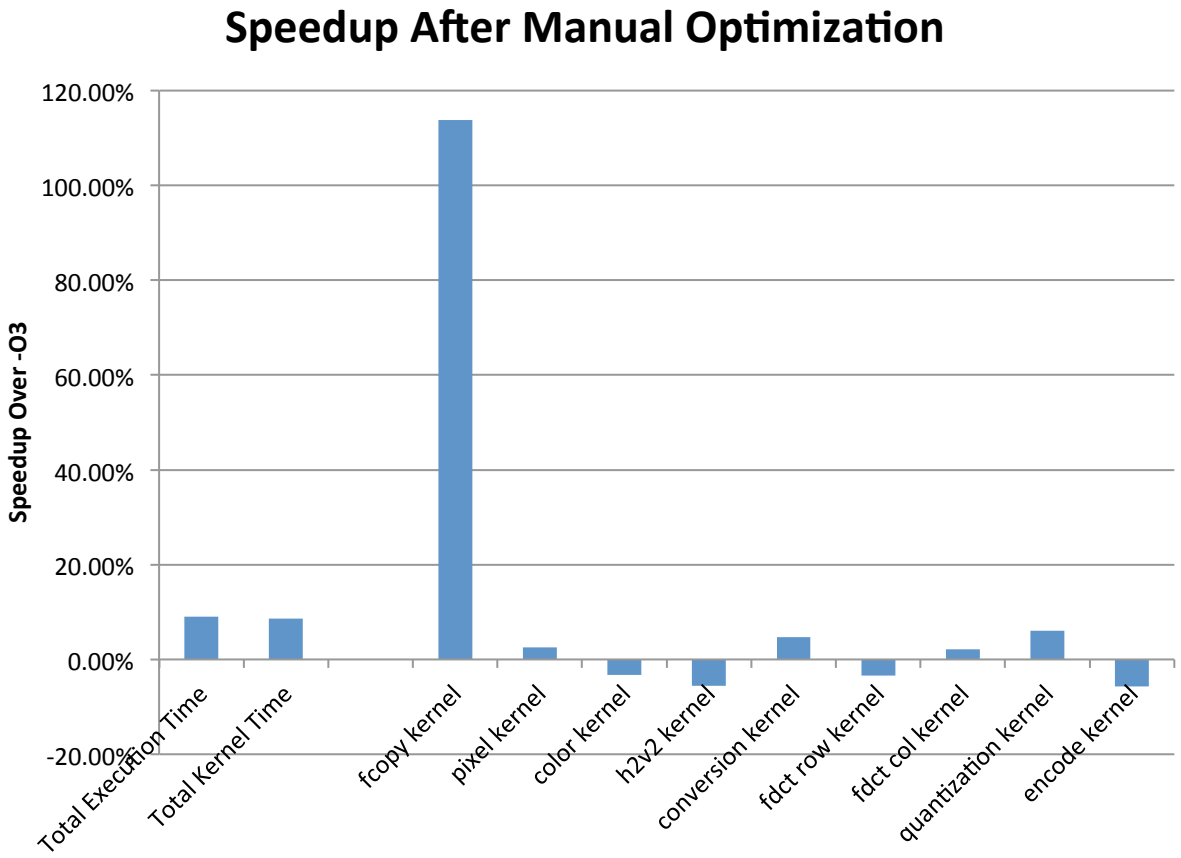


Figure 8. Speedup by kernel for manual optimizations

From the above graph, we observed negligible differences in performance from the –O3 optimized code for most kernels with the exception of the fcopy kernel, for which we achieved a large performance gain by using the memcpy() library function instead of a loop to copy data. It is sensible to us that the other two manual optimizations did not produce a significant change in the performance, because the optimizing probably performs optimizations that are similar if not identical to the changes we made manually. From the above graph we see a **9.1%** improvement in the total execution speedup.

The second manual optimization we explored was using software prefetching. Since most of the kernels perform memory accesses to arrays of data in order, we believed that prefetching data would yield a performance increase by reducing the miss rate of memory accesses in the data cache. We implemented prefetching by modifying the CJPEG source code to include inline ARM assembly code. The ARM instruction set includes a valuable instruction called PLD, which can be used to provide a "hint" to the processor that a piece of data is likely to be accessed from a specified location in the future. The following code from getpixel kernel is an example of how the PLD instruction can be used to prefetch data from a memory location pointed to by the pointer variable `intptr`and the address offset `pldaddr`:

```
/* Prefetch first 6 cache blocks */
intpld, pldaddr;
for (pld=0; pld<6; pld++) {
```

```
        pldaddr = pld*BLOCK_SIZE;
        asm volatile("pld [%[src], %[ofst]]" :: [src] "r"
        (inptr),  [ofst] "r" (pldaddr));
        asm volatile("pld [%[dst], %[ofst]]" :: [dst] "r"
        (outptr), [ofst] "r" (pldaddr));
}
```

We implemented prefetching in this way for the fcopy, getpixel, color, and quantization kernels. We chose this combination of kernels because they represented the largest performance bottlenecks. The following figure shows the performance after prefetching compared with performance of the –O3-optimized code without the manual optimizations applied:

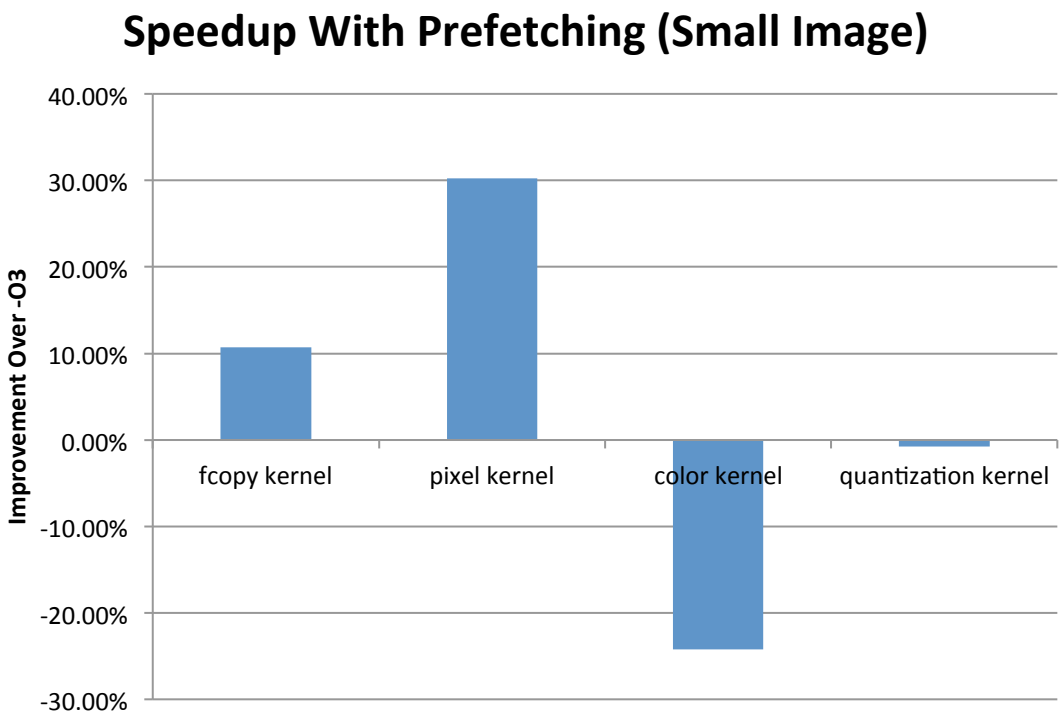


Figure 9. Speedup per kernel with prefetching implemented

From the graph, we observe significant performance gains using prefetching for the fcopy and pixel kernels but reductions in performance for the color and quantization kernels. The performance gains for fcopy and pixel were consistent with our expectations, since we assumed that prefetching data would mitigate misses in the data cache. We attribute performance reduction in the color and quantization kernels to the fact that the sizes of the data sets on which these kernels operate are smaller than a 64-Byte cache block, so performing prefetching in the kernels themselves cannot help reduce the miss rate. We can conclude from this experiment that prefetching can be beneficial when it is implemented in kernels that operate on large data sets but that it simply wastes clock cycles and can even reduce performance when applied to kernels that operate on data sizes smaller than a cache block. Therefore, we chose not to implement prefetching on the other kernels because most of them operated on small data sets.

NEON

The second part of the project required us to observe the performance of the CJPEG program when run on the NEON core and to optimize its performance for this execution unit. In the course of this undertaking, we became familiar with the intrinsic functions available in NEON for vectorizing the CJPEG kernels. During this process we learned the advantages and disadvantages of this processor for doing different operations. We will discuss them in detail throughout this section. We studied the effect of compiler optimization flags and explored inline assembly optimization to experiment its effect on the speedup in NEON.

Standalone Performance

The first step we took was to vectorize the performance of the CJPEG code. Using the provided time measurement utility, we collected measurements for the execution time of the algorithm and of the individual kernels when operated on images of various sizes. The following graph shows the vectorized CJPEG execution time for three different image sizes running on the NEON core with the –O3 flag enabled:

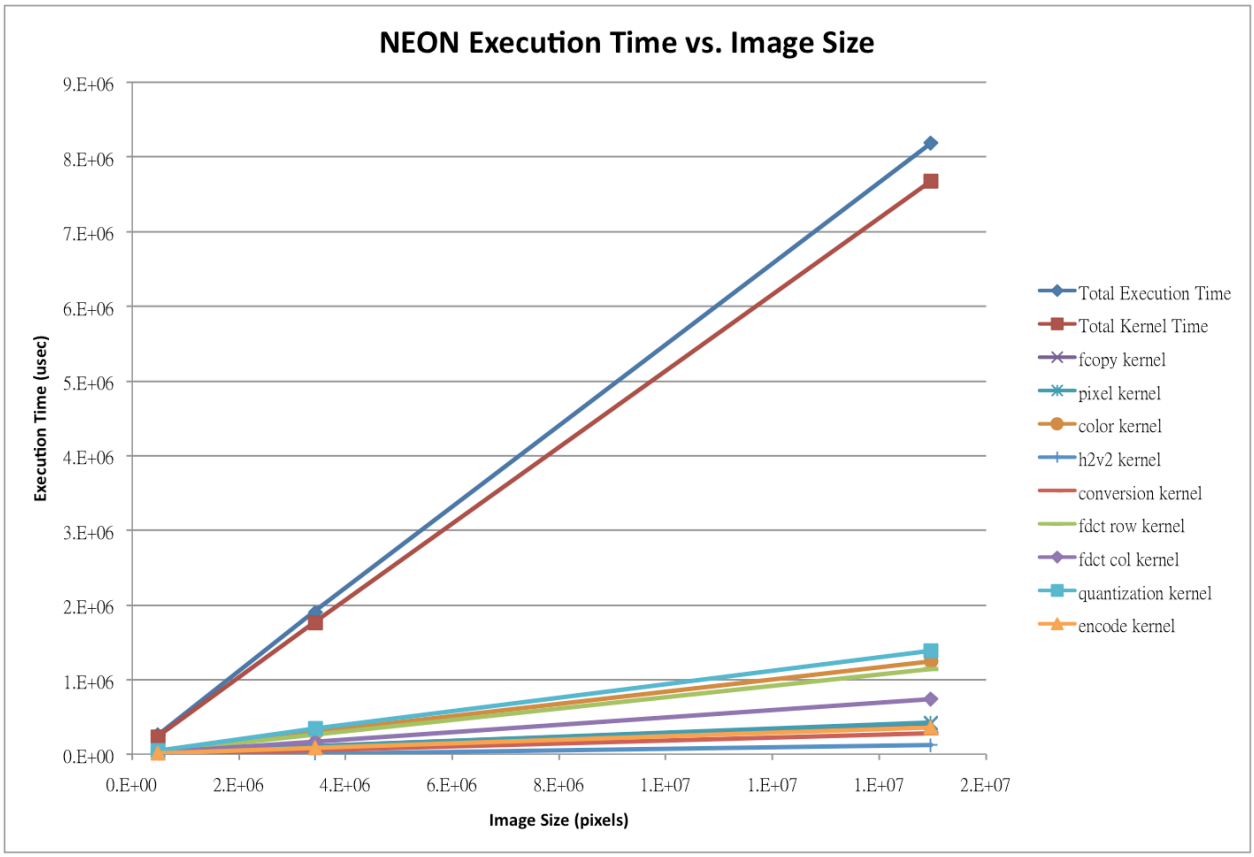


Figure 10. Graph of execution time per kernel for different image sizes

From the graph, we observe that the execution time is linearly related to the image size, which makes sense because the algorithm has computational complexity O(*N*).

The following figures show the execution time of the individual vectorized kernels for a single image:

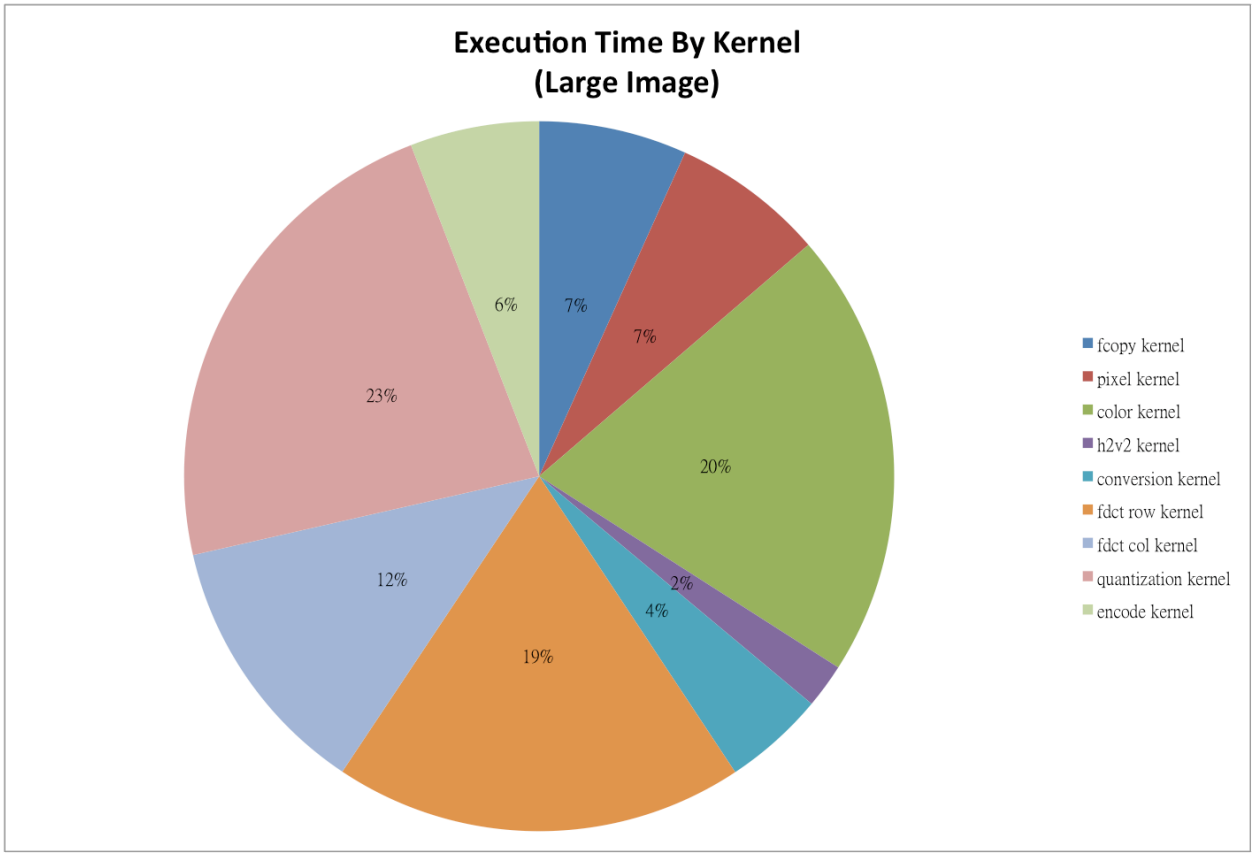


Figure 11. Breakdown of execution time by kernel

The above chart demonstrates vast differences in the execution times of each of the kernels. The main performance bottlenecks after vectorization are in color kernel, fdct row kernel, and quantization kernel.

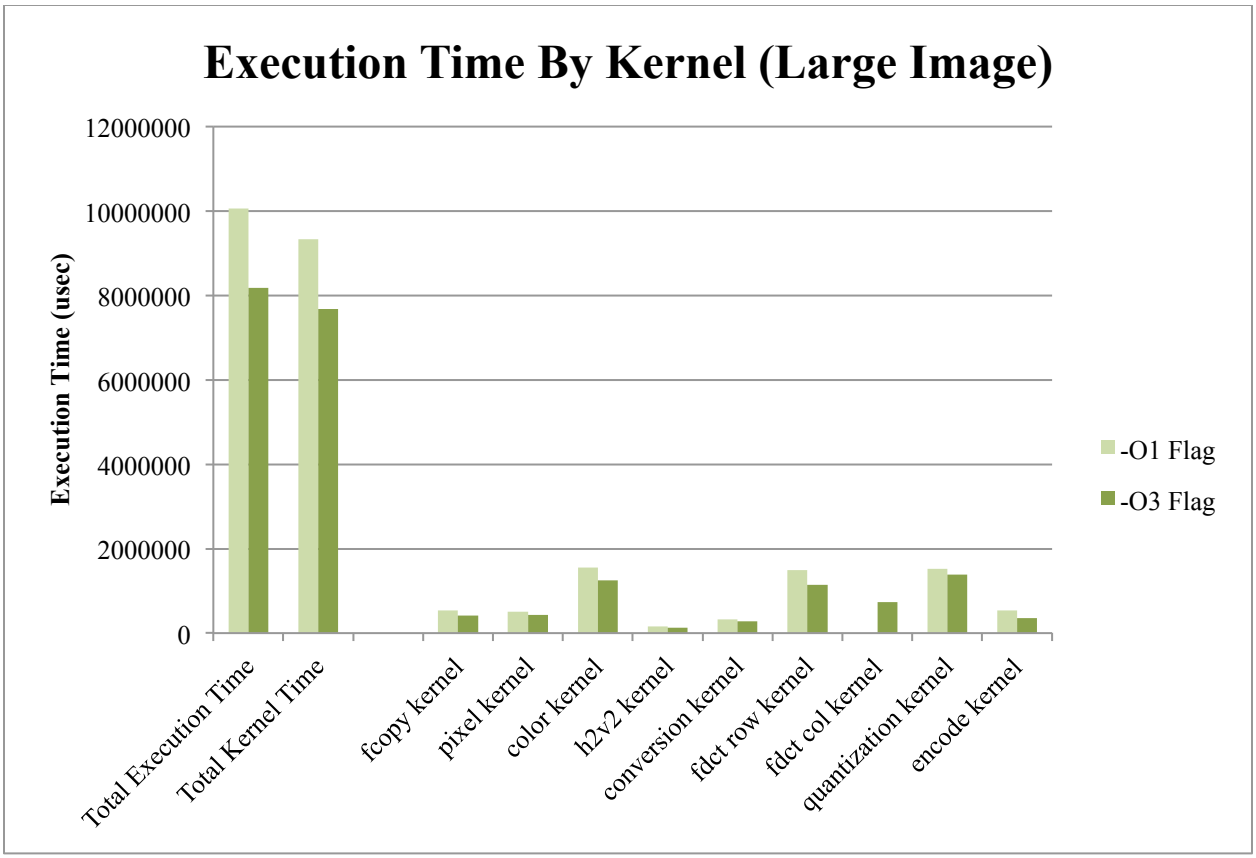


Figure 12. Execution time over the un-optimized code for different compiler optimizations for a large image

The above chart demonstrates the execution time over the O1 optimized code for each kernel. We observe that every kernel experiences execution time reduction when compiled using the O3 optimization as expected. We also observed that the total execution time and total kernel time was minimized when compiled using the –O3 flag in gcc.

However, the improvement of Neon using the compiler optimization flag is not as effective as the Arm code. In ARM, we get a 20% improvement when we change compiler optimization flag from O1 to O3, but in the NEON part, we only get 15% improvement. This probably means that the compiler optimization does not optimize NEON code as well as optimizing the ARM code, and that is probably the reason why in some kernels the NEON is worse than ARAM when we are using the O3 optimization flag: although the vectorization in NEON could get a better performance, but the compiler for ARM could get an even better performance with its smart optimization, which does not work well for NEON.

Comparison with ARM

For each of the three image sizes, the following graph (Figure 13) illustrates the speedup of the vectorized kernels over the non-vectorized kernels. We calculated the speedup with respect to the execution time of the CJPEG program as it was provide to us compiled with the –O3 compilation flag enabled. We used this as a baseline for comparison between NEON and ARM because it allowed us enforce consistency between the results for the two execution units without implementing software prefetching and manual optimizations on the NEON code. Explanation of the performance of each kernel is included in the following section.

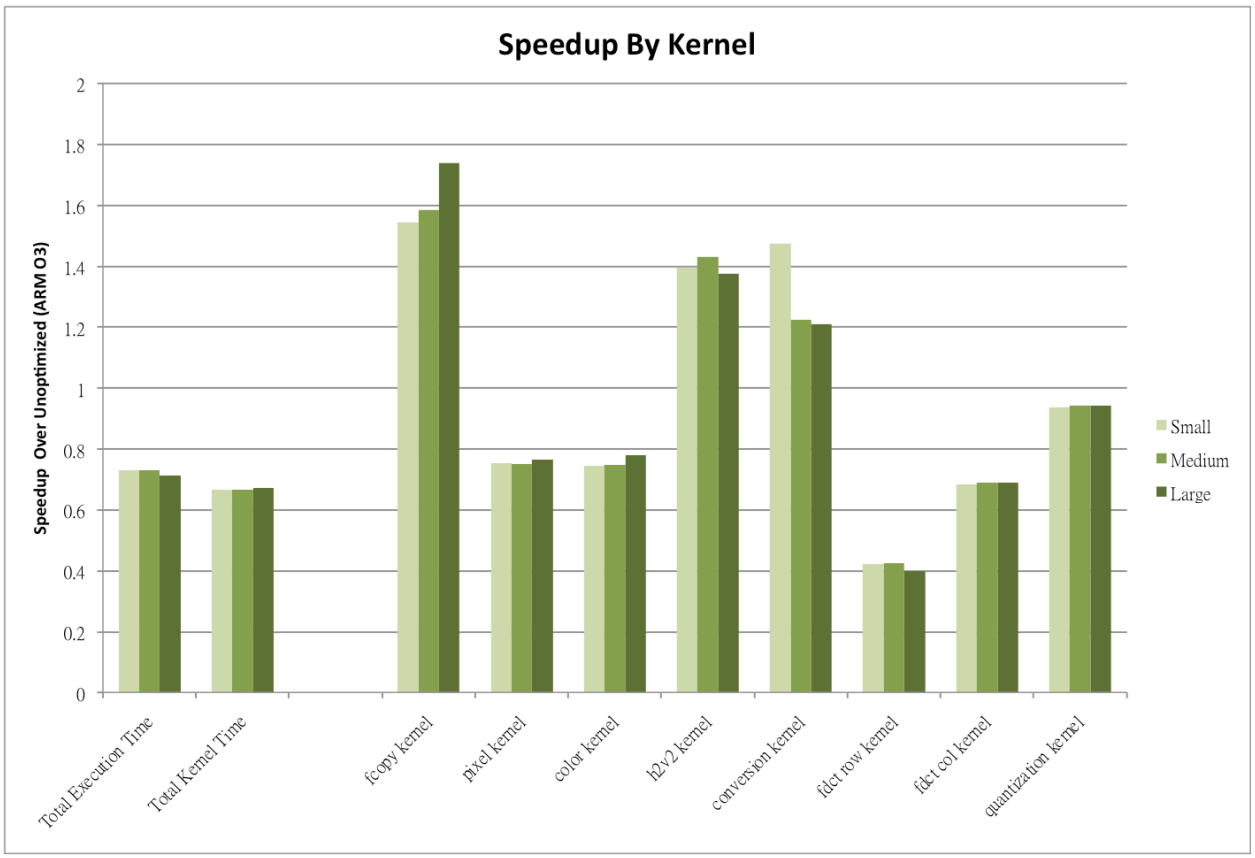


Figure 13.Speedup of the individual kernels for different image sizes (O3 compilation flag is enabled)

Figure 14 shows the highest achievable speedup of the our CJPEG implementation collected by running only the kernels with > 1 speedups on NEON and running the rest on ARM. The Total Execution Speedup is now 13% higher than running the entire code on ARM. We are happy about this! The total execution speedup is *almost* equal for all image sizes. This makes sense as, *on average*, increasing the image size introduces equal amount of load to all ARM kernels and all NEON kernels.

Note that the NOEN speedup is computed in reference to the execution time of ARM with –O3 flag enabled. This is why 6 of the kernel speedup values are 1.

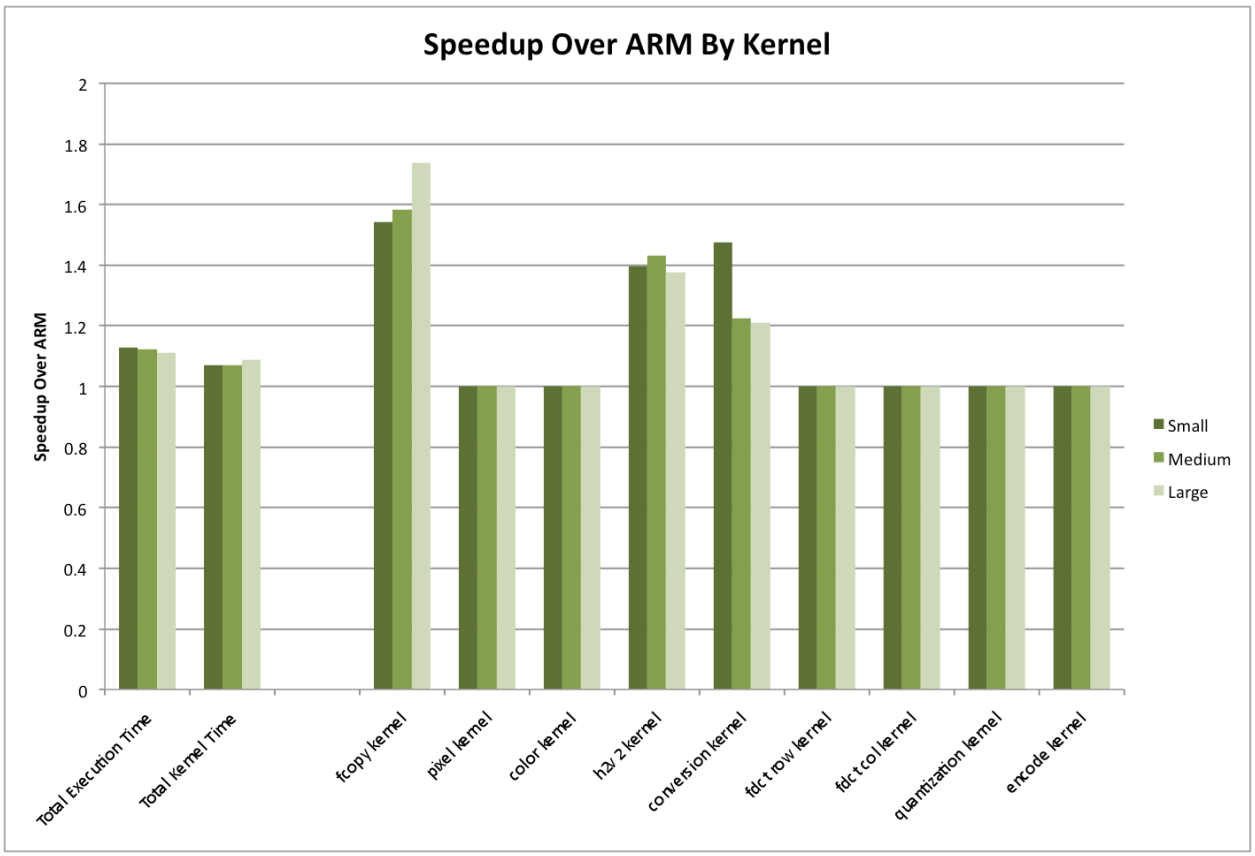


Figure 14.The best per-kernel-Speedup for ech kernel, combining ARM and NEON execution times.

Kernels

The following sections describe, in detail, the individual kernels we implemented on NEON. Each section describes how we vectorized the kernel, the performance we expected to achieve, the performance we actually achieved, and why we got the results we observed.

fcopy_kernel

fcopy_kernel is the most easily vectorizable kernel as it merely copies 8-bit data between two arrays. On each vector we packed 16 data-values and reduced the number of loop iterations by a factor of about 16. As mentioned earlier, a smaller number of loop iterations means a smaller number of branch predictions and arithmetic operations (i.e. loop overhead time) and consequently a shorter execution time. Since larger arrays bear larger loop iterations, it is expected to observe even more improvement in speedup when moving form small image sizes to larger ones. In fact, in Figure 13, fcop_kernel has the highest speedup compared to the ARM (with -O3 flag) execution time. Also, fcopy_kernel can be considered the reference data-set for determining how long a simple data transfer between registers and memory takes in NEON and attribute the speedup differences between fcopy_kernel and other kernels (see Figure 13) to the additional arithmetic/logical operations other kernels perform.

In addition, Figure 11 indicates that fcopy_kernel accounts for only 5% of the total execution time, making it a very efficient kernel for NEON compared to other kernels.

getpixel kernel

This kernel transfers the raw image pixels, represented in 8-bit BGR color, from one memory location to another, re-ordering the color components into an RGB arrangement in the process. The kernel lends itself well to vectorization, because the sequential loads and stores it performs in the copying process can be easily translated into vector loads and stores using the NEON intrinsic functions.

To implement the vector load and store, I used the vld3q() and vst3q() intrinsics—which implement the NEON VLD3 and VST3 vector instructions—to copy the pixel data to and from memory and also package them into groups of red, green, and blue pixels. Re-ordering the pixels was slightly tricky, as I had to swap data between the Q registers that contain the red and blue pixels. I expected that the extra instructions needed to swap the data could diminish the performance gain I saw.

I expected to see a large reduction in execution time over the non-vector implementation due to the parallelization I exploited by using the NEON vector instructions. The VLD3 and VST3 instructions load and store 16 8-bit values in parallel, allowing me to reduce the total number of loop iterations in getpixel kernel by a factor of 16. It was my hope that the parallelization would directly result in a reduction in execution time by a factor of 16, or perhaps something smaller considering the potentially detrimental register swap.

To our surprise, the execution time actually increased comparing to the non-vectorized implementation. We observed an average speedup of only about **0.75** for various image sizes (see Figure 13). To investigate why this is the case, I examined the assembly code generated by the compiler, and discovered that the compiler implements the loop using a total of 53 instructions inside the loop. It appears to us that the compiler is generating vastly inefficient assembly code, which is clearly responsible for the bad performance result. To obtain an accurate performance measurement of this kernel, I eventually resorted to vectorizing it using inline ARM assembly, which is explained later in this section.

color kernel

The purpose of the color kernel is to transform the pixel from RGB representation to YCrCb representation, which means that we need to use the R, G, B values in each kernel to calculate the new Y, Cr, Cb value in the same pixel. Each Y, Cr, and Cb in the new representation is the linear combination of the R, G, and B in the original representation. Thus, this kernel is pretty suitable for vectorization since linear operation is exactly what the vector could do. In the beginning, we use an approach that vectorizes each pixel as an element in the vector, which means that each vector element would be 24 bits since R, G, and B are all 8 bits. After the vectorization, the vector will look like this:

Then we could do operations within the vector elements to derive the Y, Cr, and Cb. However, as you could see, since there is no data type that uses 24 bits, we need to use a 32-bits integer as a vector element to accommodate one pixel, and this means that we are wasting computation

power of the CPU. Therefore, instead of using this approach, we use another way to vectorize the data: vectorize each R, G, and B of each pixel into three different vectors, which could be done by using vld3_u8 to get three arrays. Therefore the new vector will look like this:

8bits	8bits	8bits	8bits	8bits	8bits	8bits	8bits
8bits	8bits	8bits	8bits	8bits	8bits	8bits	8bits
8bits	8bits	8bits	8bits	8bits	8bits	8bits	8bits

After this step, we could directly do linear operations on these vectors and get eight Y, Cr, Cb values at a time, which should be able to use the full power of the CPU.

On the other hand, there is another issue in implementing vectorization in this kernel, which is the fix point operation. Because the fix point operations need a space that is more than eight bits, we are not able to directly do the linear operation on each element because there will be an overflow if we do operations on these 8-bit space. Therefore we need to separate each of the vectors above into two vectors, and make every element 32-bits so that there is enough space for the fix point operations, and the vector will become like this:

32bits	32bits	32bits	32bits
32bits	32bits	32bits	32bits
32bits	32bits	32bits	32bits

Finally, after the operations are done, we could now shrink the element size back to 8 bits, and store the Y, Cr, and Cb value to the output pointer, so the final output will be:

Y 8bits	8bits	8bits	8bits	8bits	8bits	8bits	8bits
Cr 8bits	8bits	8bits	8bits	8bits	8bits	8bits	8bits
Cb8bits	8bits	8bits	8bits	8bits	8bits	8bits	8bits

This kernel occupies 16% of the total execution time, so it would be worthwhile if we could optimize this kernel by using the Neon intrinsic. At the first glance, we were expecting the color kernel could have better performance in Neon than Arm because in this kernel the operations are all linear operation, which should be able to be executed more efficiently by vetorizing the data. However, according to the “Speedup By Kernel” plot, the performance does not become better, and only get 0.8 of the original performance in Arm. We believe that this is because there are too much overhead when vectorizing the data in this kernel. Since there are fix point operations, we are not able to do calculations in the original data after vectorizing the data, and need to make the size of vector elements bigger to accommodate the output data. Then after the operations, we need to shrink the vector element size back to the original size, so that the data could be stored back with correct size. These lengthening and shrinking actions might add a lot of overheads

when vectorizing data. Therefore we could not see performance gain in this kernel. If there is a way to load 8-bit data into a 32-bit vector element efficiently, we believe the Neon should be able to give a better performance than Arm.

h2v2 kernel

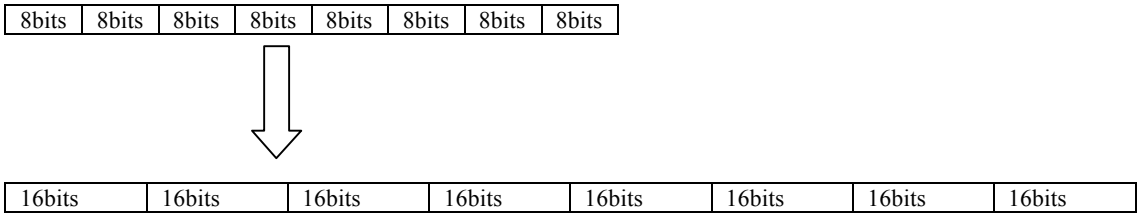
This kernel down-samples the image pixels. Like getpixel kernel, it is easily vectorizable because it consists mainly of a loop containing a single load, store, and some intermediate arithmetic. Modifying the kernel to exploit the vector instructions required simply translating the sequential arithmetic operations into analogous vector operations using the NEON intrinsic functions.

Like getpixel kernel, I expected to observe a large reduction in execution time over the non-vector implementation. By exploiting the vector instructions, I was able to perform operations on 8 operands in parallel withina single loop iteration, hence, reducing the total number of loop iterations by 8. Because nearly all of the required computations are completely vectorizable, I expected to see a subsequent reduction in execution time by a factor of 8.

The speedup we observed was about **1.4 (**Figure 13). The difference between the speedup we expected and what we observed could be a result of a number of inefficiencies. As explained previously, the gcc compiler seems to be due to the generation of inefficient code when compiling the NEON intrinsics. Also, the efficiency of some of the intrinsics is ambiguous. To implement vector operations, I made use of a few intrinsics that did not have a clear sequential translation. For instance, I used the vpadalq() instrinsic—which implements the ARM VPADAL pairwise-add, single-opcode widen and accumulate instruction—to condense several sequential operations into a single instruction. It is unclear as to how this instruction and other unusual instructions like it improve or degrade performance. It is also unclear as to how gcc implements them in ARM assembly. It is most likely that the gcc compiler is producing vastly inefficient code that is degrading the performance, as was the case in getpixel kernel.

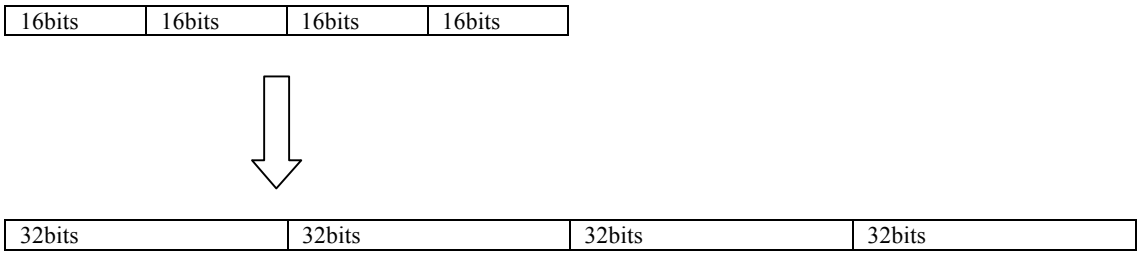
conversion kernel

The purpose of conversion kernel is to convert an unsigned integer into a signed integer, which means that we are going to convert 8-bit data into 32-bit data. In order to achieve this, we need to lengthen single vector elements, which is done by using the vmovl_u8 two times to make the element size quadruple. Therefore the first thing after loading the vector is doubling the element size:



In this process, one issue is that we do not need to worry about the sign bit because we are transforming from the unsigned to the signed number, so the zero extension would be

automatically correct. Finally, we cut this vector into two smaller vectors, and double them again to get the final output:



In the "Speedup By Kernel" plot we could observe that the there is an 20 percent jump in performance. In this kernel we could see that in each loop the old code extract a number out and then cast it into integer. Then deduct a number from it and save it back. In the new code, we extract eight numbers at a time in each loop. Since lengthening an element (same as casting here) and deducting a number are all linear operations, we could use vector to parallelize these operations in Neon without much overhead, so in this kernel we are processing eight numbers at a time without paying much effort vectorizing data. Therefore we are able to see this performance improvement. However, this kernel only occupies 5 percent of the total execution time, so we might not be able to perceive an obvious improvement in the total execution time.

fdct row kernel and fdctcol kernel

This kernel performs the forward discrete cosine transform on the columns of the image. It is a complex algorithm that is not easily vectorizable. There are sections of the source code containing similar operations performed sequentially that lend themselves well to vectorization. Other sections, however, contain code that operates on scalar variables and are not easily vectorizable. For these sections, we were faced with the choice between trying to implement them as best we could using the vector instructions or simply implementing them using scalar arithmetic. It was usually unclear as to which method would yield the largest benefit to performance. On the one hand, by using the vector instructions, we could implement a few arithmetic computations in parallel. On the other hand, the overhead required to package the scalar variables into vectors to perform the parallel operations was detrimental to performance.

In the interest of attempting different approaches to vectorizing the kernels, two of our group members vectorized each kernel independently using differing strategies. For fdct row kernel, we vectorized pieces of the code and performed a little scalar arithmetic in the sections that were not easilyvectorizable. For fdccol kernel, we implemented it entirely using vector intrinsics with no scalar arithmetic. From examining the code, we could not gain too much intuition into what kind of performance we could expect from vectorizing the kernels. Discussions with other project groups led us to believe that there was room for a 5-10% reduction in execution time.

We observed an increase in execution time over the non-vector implementations for both kernels. For fdct row, the average speedup was about **0.4**, and for fdctcol, the average speedup was about **0.7**. There are numerous potential sources of performance lag within the vectorized kernels, and we believe that the most probable cause of this poor performance is that we tried to vectorize as many things as possible. Therefore we put in a lot of effort in vectorizing data,

which is not worth it. For example, we vectorized a vector with only two elements, and then do only one vector operation, and then store it back to the memory. This kind of actions cause a lot of overhead and does not gain benefits from the vector operations because there is few parallel operations conducted and the overhead of storing and loading vectors is too big. Therefore we believe that only when the vector operation number is above a certain number of operations could we get benefit from vectorizing data. In the case of DCT transform, we think it would be better to use the sequential scalar operation in most parts of the code.

quant_kernel

The function of the quant kernel basically is to divide one number to another one. However, since the input number could be negative, and we need to do rounding up after dividing one number to another one, there are a couple of operations that we need to do to achieve this. Although there is a non-linear operation, which is getting the absolute value of the element, fortunately the Neon vector operation provides an intrinsic that gets the absolute value for each element in the vector. Therefore this kernel seems pretty suitable for vectorizing. We then proceed by vectorizing two integer arrays into two vectors so that we could deal with four numbers at a time, instead of only one number in the original code. After loading the vectors, we could use vabsq_s32 to get a new vector with absolute values of the original vector and then do the operation we need. However, since we need to change each negative element back to negative after the operations are done, we need to find a way to record each element's sign bit so that we know which element to be converted back to negative. Our approach is to use the Neon intrinsic vcltq_s32, which is "less than" function. It will return an unsigned vector with element bits full of one if the corresponding element satisfies the condition (less than zero). For example, if a vector looks like this:

-10	3	-5	17

Then the vctltq will return this "sign vector" with each element is a 32-bit unsigned int:

111111111111…11111111	0000000000…..000000000	1111111111….1111111111	00000000…00000000000

Although this sign vector could record which element is negative, it is hard to be used to convert element back to negative. Therefore we are going to do a trick here. We want to make this "sign vector" element to have value -1 when the corresponding element value is negative, and value 1 when the corresponding element value is positive, so that we could simply multiply the output vector to this sign vector to get the original sign back. Our trick is like this: since the negative element is now filling with 1, and we know that all 1s in a signed integer format means -1, we could simply "cast" this unsigned vector to signed vector, which makes all-one-element become -1, and 0 remain 0. Therefore the new sign vector looks like this:

-1	0	-1	0

Then we multiply this vector by two, and plus one to every element, which makes this vector become:

-1	1	-1	1

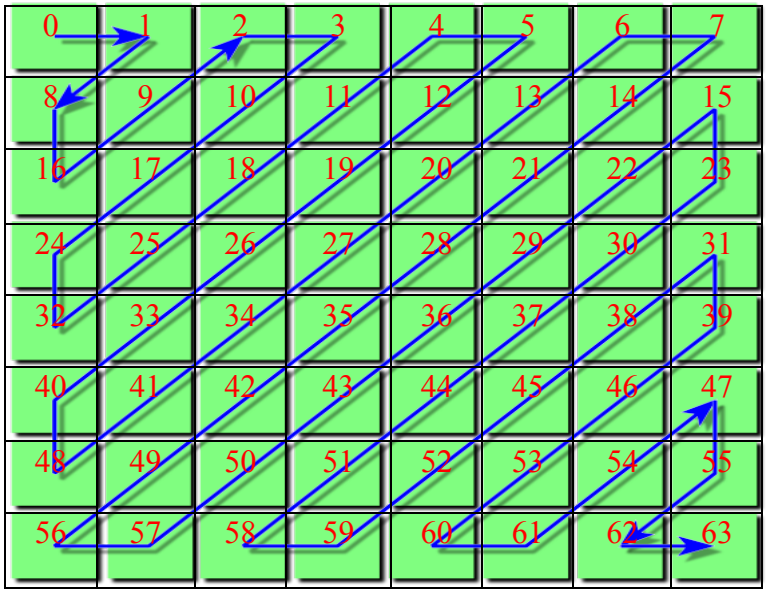
Then we could use this vector to multiply the output vector to get the original sign later.

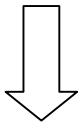
The other issue in this kernel is that we are not able to have division operation, because the Neon intrinsic does not provide division. Therefore, we solve this problem by storing this vector back and do the division in the normal code sequentially. After the division, we reload the vectors again and do the rest of the operation. Finally we shrink the vector to short size so that it could conform to the output pointer.

This kernel occupied 23% of the total execution time, so it would be very helpful if the Neon could improve the performance. However, we are only getting 0.9 of the original speed, and we are guessing that this is due to the division operation causing a lot of overheads. In order to perform the division, we need to store the vector back to memory, and then reload it back to vector after the division. This is very inefficient, plus that the division is done sequentially instead of parallelism, so the benefit of Neon on other vector operations is compensated by this overhead. As a result, we are not able to see a performance gain in this kernel. If the Neon could support the division operation for vector, we believe we could see an improvement by reducing the overhead produced by the sequential division and the vector store and reload.

encode kernel

The purpose of the encode kernel is to put things in zig-zig order. Since there are a lot of non-linear operations inside this kernel, we did not vectorize this kernel to make it run on Neon, which should probably be very hard and would not produce good performance. The first non-linear operation we had was to put numbers in zig-zag order. Apparently this is an operation that is pretty hard to achieve using the vector, but we did try to consult all the Neon intrinsic to see if there is any function could apply to this and help us vectorize this kernel. We know the zig-zag natural order is as follows:





0	1	8	16	9	2	3	10
17	24	32	25	18	11	4	5
12	19	26	33	40	48	41	34
27	20	13	6	7	14	21	28
35	42	49	56	57	50	43	36
29	22	15	23	30	37	44	51
58	59	52	45	38	31	39	46
53	60	61	54	47	55	62	63

As you could see from this matrix, the zig-zag order goes from left to right, then going down row by row. This means that the in our new output matrix, the first element should be the first element in the input matrix, and the second element in output matrix should be the 2nd element in the input matrix, and the third element in the output matrix should be the 8th element in the input matrix, etc. At first, we tried to use vector table look up to implement this non-linear operation. However, since the longest vector is only 8 elements long, we would not be able to extract the 16th element of a vector when we need to output the fourth data element in our first run. On the other hand, there is another non-linear operation, which is discarding the zero data, which would also be hard to be done in vector efficiently. Since we need to examine every element in a vector and put the zero away, which would not utilize the efficient linear operation of vector at all. Therefore we rather do these two operations in original way and did not vectorize this kernel.

Since this kernel only occupies 6% of the total execution time, it would not affect much. Note: this kernel does not appear on the "Speedup By Kernel" plot because we did not change the code in this kernel and there is no reason to compare it with the one in Arm.

Inline Assembly Code

While investigating the performance results we obtained for our vectorized kernels, we observed that the compiler seemed to be producing inefficient ARM assembly code when we used the NEON intrinsic functions. We believe that this caused us to fall far short of our expectations for performance gain from vectorizing and, in some cases, caused us to experience diminished performance.

Disappointed by the lackluster performance of our vector kernels, we decided to experiment with implementing NEON vector instructions using inline assembly code. As a test case, we implemented the getpixel kernel. The following code implements a vector load from the `inptr`array to 3 D registers, a D register swap, and store to the `outptr`array with NEON vector instructions using inline assembly:

```
asm volatile("vld3.8 {d0, d2, d4}, %[in]" :: [in] "m" (
inptr[j]));
asm volatile("vmov.64 d3, d0");
asm volatile("vmov.64 d1, d4");
asm volatile("vst3.8 {d1, d2, d3}, %[out]" :: [out] "m"
(outptr[j]));
```

The following graph compares the performance of the getpixel kernel using the inline assembly to that using only the intrinsic functions:

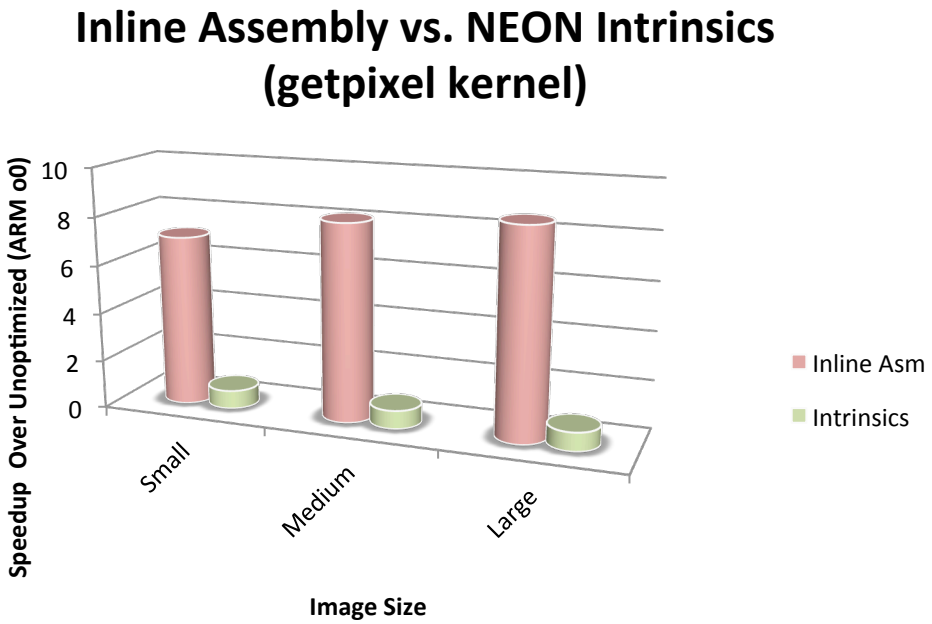


Figure 15. Performance of getpixel kernel implemented using inline assembly and NEON intrinsic functions

From the graph, it is clear that the inline assembly implementation is vastly more efficient than the NEON intrinsic implementation. We observe an average speedup of about **8** over the non-vectorizedgetpixel kernel, which is exactly what we initially expected from our back-of-the-envelope calculation. We can generalize from this result and conclude that, if vectorized properly using inline ARM assembly, most of the other kernels in the CJPEG that clearly lend themselves to vectorization—such as fcopy, color, h2v2—should exhibit significant speedup over the non-vector equivalent.

DSP

The third and final phase of our project involved porting the individual CJPEG kernels to run on the DSP. Achieving this objective required significant changes to the structure of the CJPEG program and forced us to think carefully about our implementation strategy, specifically with regard to the kernels we chose to implement and how to handle the communication between the DSP and the ARM core. This part of the project proved to be the most challenging by far and instilled in us an appreciation for the great pains that engineers must take in the name of achieving potential improvements in the performance of their software.

This first step we took was to determine which kernels we would implement on the DSP. The main performance advantage of using the DSP to run the kernels is that it is potentially faster at performing numeric computations than the ARM core. With this in mind, we chose to focus on kernels that contained heavy computation and to bypass kernels that with no significant computation, such as fcopy kernel and getpixel kernel, which merely copy data.
We also considered how the communication between the DSP and the ARM core would affect our choice of kernels. Due to the significant overhead involved in transferring data between the two cores, it is most advantageous to minimize the amount of communication. We decided to target kernels that we could implement back-to-back on the DSP so that we could avoid transferring data between the cores after the end of one kernel and before the beginning of another.

Out of the 5 kernels that we believed we could potentially improve, 4 of them are implemented in succession with little intermediate computation. Only the color kernel must be implemented independently of the other 4, which means that the code must perform two data transfers for this kernel in addition to the two performed for the other 4. We decided to port the color kernel to the DSP first and test its performance to determine whether or not it would be to our benefit to implement this kernel in addition to the other 4. After running the CJPEG code while executing the color kernel on the DSP, we observed an execution time of **759138.29 microseconds** for the color kernel when performed on a small image. This was significantly greater than the kernel execution time when run on the ARM. Considering the large overhead required for transferring data for the color kernel to operate on, we decided to omit it from our implementation. We therefore decided to implement only conversion kernel, fdct row kernel,

fdctcol kernel, and quant kernel on the DSP, because we could easily call these kernels back-to-back and transfer data only once at before start of conversion kernel and again at the end of quant kernel.

The next step we took was to decide how to implement the data transfer between the cores. After performing some initial experiments on the communication overheads involved in transferring data, we observed that it would be most advantageous to transfer the image data in blocks of the maximum transferrable data size, 4KB, to minimize the impact of the overhead. Given that the CJPEG code was already structured to operate on 2-dimensional arrays of 8x8 pixels—which were represented by data blocks of 256B—we found it most convenient to implement the code such that it transfers data of this size to and from the DSP and subsequently maximizes designer sanity (a rare commodity throughout the course of this project section).

We tried two main approaches for transferring data between ARM and DSP. We first tried transferring 2D arrays of 8x8 blocks across to the DSP core. We realized that doing so involves extra coding complexity that would merely complicate our coding approach and consequently consume our time debugging the code. Also,we found more than one set of data that must be sent to the DSP every time, making it even more complicated to talk to the DSP without running the risk of making any mistakes. In the end, we decided to aggregate all our data into a 1D array and transfer them across.Once the data are at their destination, they are broken down into proper variables and arrays.

Standalone Performance

The following figures characterize the performance we observed after implementing conversion kernel, fdct row kernel, fdctcol kernel, and quant kernel to run on the DSP:

Figure 16. Execution time for three different image sizes

The above graph demonstrates the execution time is linearly related to image size, which is consistent with our observations on the previous parts. The following graph shows the breakdown of execution time by kernel:

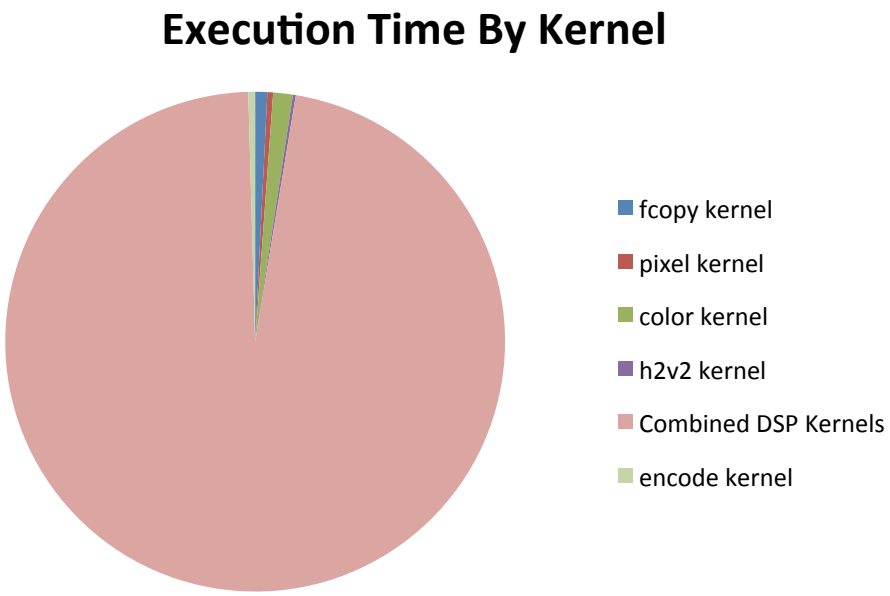


Figure 17. Breakdown of execution time by kernel

The previous chart clearly shows that the kernels implemented on the DSP, including the time consumed in the data transfer, dominated the total execution time. We believe that a large portion of the execution time recorded for these kernels is due to the data transfer, which is quantified later in this section.

Comparison with ARM

The following graph shows the speedup achieved over the CJPEG implementation that runs solely on ARM for three different image sizes (compilation flag O3):

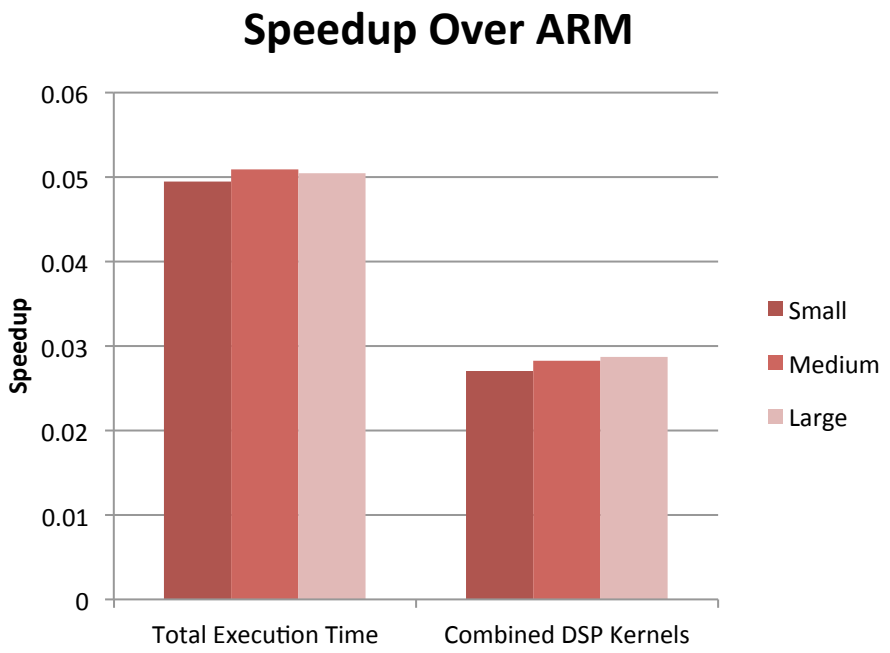


Figure 18. Speedup over ARM

The graph shows that we clearly failed to achieve any performance gain by executing kernels on the DSP and that we in fact reduced the total performance by a factor of about **20**. We attribute this performance reduction, in part, to the enormous amount of time needed to transfer data between the DSP and the ARM core.

As part of our initial characterization of the DSP's performance, we experimented transferring data of different sizes to and from the DSP and observed the latency. The following graph shows the relationship between latency and data size:

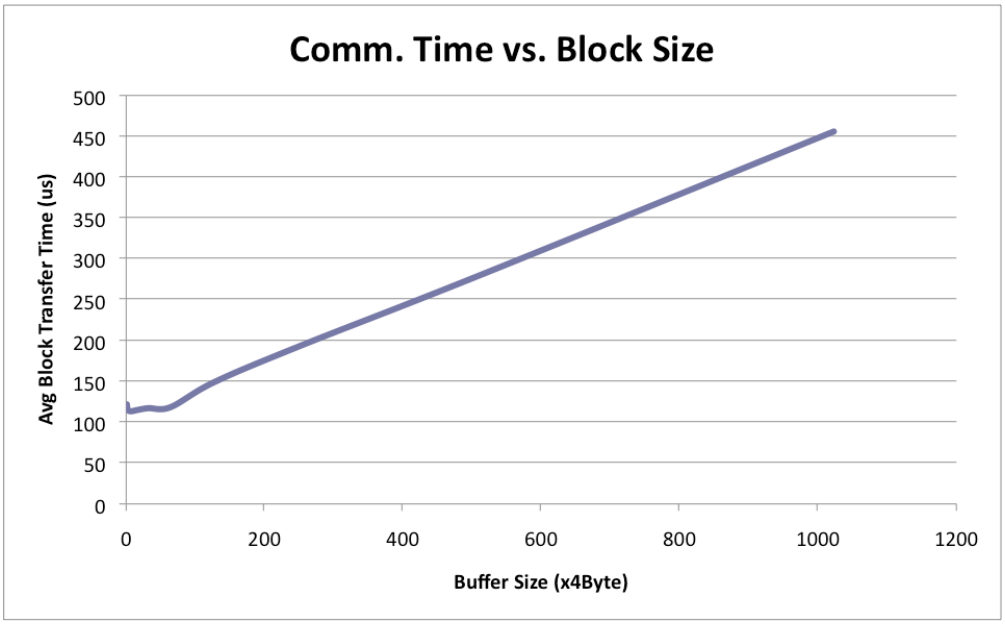


Figure 19. Communication latency vs. input buffer size

The graph demonstrates that the communication latency is linearly related to the input buffer size. We also observe that there is an additional fixed overhead of about 100us. The following graph shows how the number of bytes transferred per unit time changes with the data size:

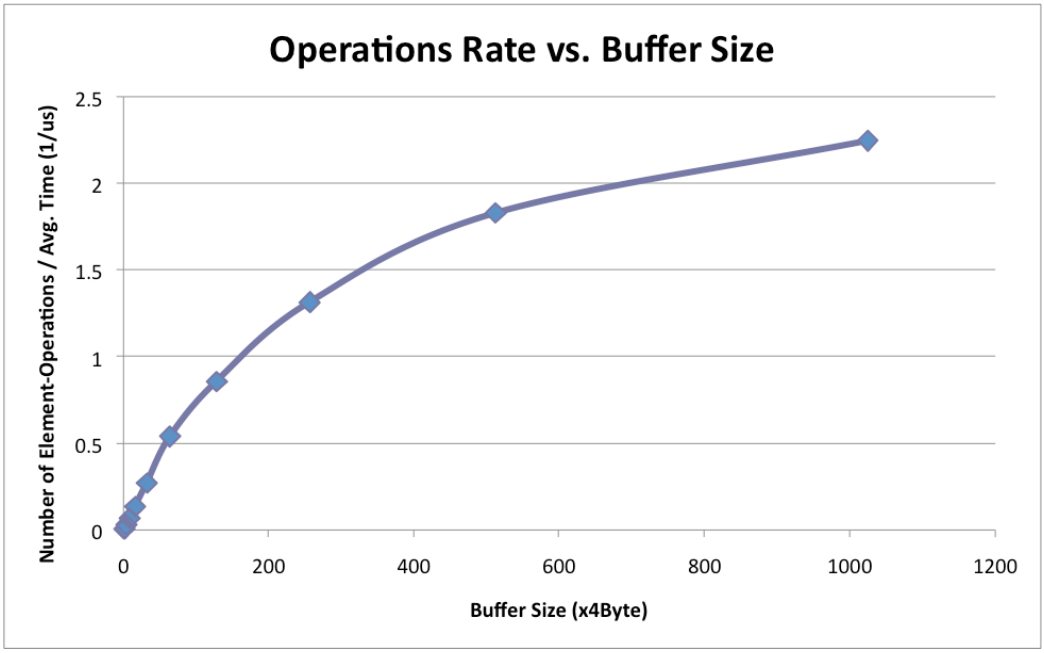


Figure 20.Communication latency per byte.

The graph demonstrates that the number of bytes transferred per microsecond increases with increasing buffer size and is maximized when the buffer size is equal to the largest transferrable size. It makes sense that the efficiency of the data transfer should increase with the buffer size because the fixed overhead is amortized over a larger data set. This result shows that, by transferring image data between the DSP and ARM core in 256B blocks instead of the maximum 4KB blocks, we are sacrificing performance in the transfer. From the graph, we observe that our CJPEG implementation operates with an efficiency of about **1.4** operations per microsecond when transferring data in 256B blocks and that the maximum achievable efficiency is about **2.2** for 4KB blocks. Therefore, if we were to transfer data in 4KB blocks, we could theoretically achieve an additional performance gain of about 2.2/1.4 = **1.6** over the performance we observed. Thus, unless we can implement a better mechanism for communicating data between the two processors, using DSP for doing any of the CJPEG kernels is disadvantageous.

To quantify the effect of the data transfer on the execution time, we performed an experiment in which we measured the time taken by simply transferring data to and from the DSP for an entire image in 256B blocks *without* operating the 4 DSP kernels on the data. We then subtracted the transfer time we measured from the kernel execution time we measured previously to determine the true execution time of all the kernels. The following figure shows the breakdown of execution time by kernel when transfer time is ignored:

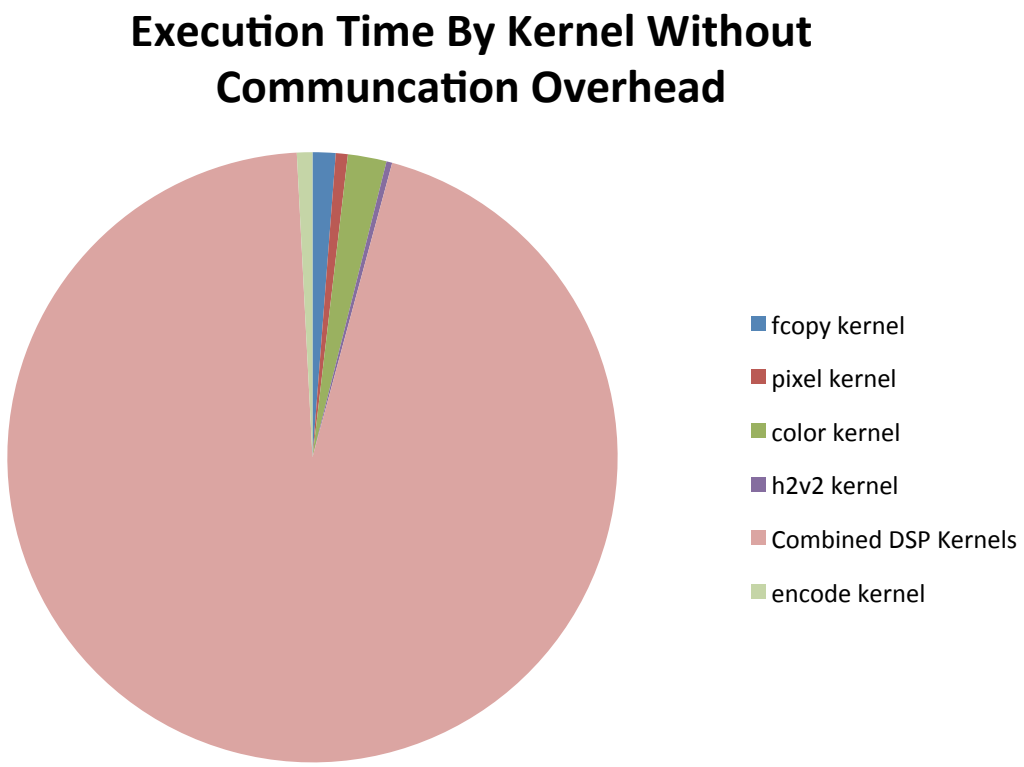


Figure 21. Execution time by kernel with communication overhead removed

The above chart demonstrates a somewhat surprising result: the 4 kernels that run on the DSP still make up for the overwhelming majority of the execution time even when the transfer time is ignored. The following graph shows the speedup over the ARM implementation:

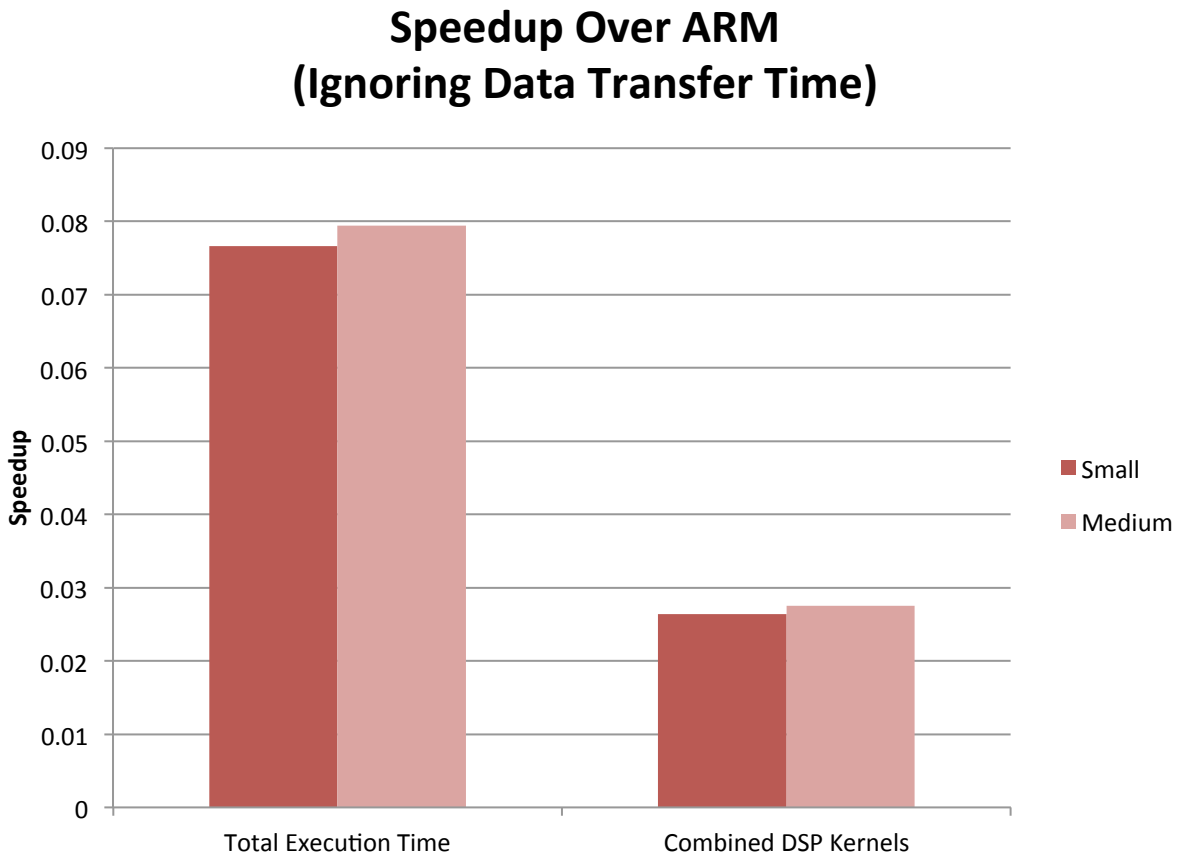


Figure 22. Speedup over ARM ignoring data transfer time

From the chart, we observe that the speedup we achieved for the 4 DSP kernels was about **0.03**, while the total speedup was about **0.8**, which shows that the CJPEG program that used the DSP was about **12.5** times slower than the program that used only the ARM core.

We conclude that the algorithms performed by the 4 kernels that we run on the DSP simply do not benefit from the potential advantages of the DSP's capabilities for executing numeric computations when implemented in straight C code and compiled using gcc. Thus, the only way for us to obtain time-efficient results is by utilizing DSP intrinsics which we believe can significantly improve the performance of each kernel (not considering the data communication overhead).

DSP Intrinsics

DSP Intrinsicfunctions provide great support for 8bit and 16bit values. This is because DSP has 64 32-bit registers in which multiple data can be packed (i.e. vectorized). This allows parallel computation of a larger number of data at once.

As mentioned earlier, our goal for this part of the project, was to first collect performance data on the currenct runtime of CJOEG kernels on DSP. Once we obtained those results, we decided to start using DSP intrinsics in the kernels transferred to the DSP. We realized, however, that since all the data-types in all the abovementioned four kernels utilize 32-bit data we have no way utilizing the DSP intrinsics in our kernels. However, we realized that the color kernel operates on 8-bit data and has a potential for utilizing the intrinsics. As a result, we decided to spend some time learning the DSP intrinsics by attempting them on this kernel. We made this attempt knowing that it will eventually not provide us with any significant speedup as we know the DSP speedup is greatly affected by transferring data between the ARM and DSP. Additionally, we tried to work with different type identifiers such as `const` and `restrict` that we knew help producing more time efficient kernels binary code for the DSP.

The code snippet below is one of the three lines in color_kernelthat are vectorizable using the approach explained below.

```
outptr0[col] = (JSAMPLE) ((
                                FIX(0.29900) * r +
                                FIX(0.58700) * g +
                                FIX(0.11400) * b +
                                ONE_HALF
                                )>> SCALEBITS
                            );
```

The steps taken to vectorize the above code is listed below and illustrated on next page. The corresponding DSP instrinsic commands are also included inthe graph.

1. Four 8-Byte numbers are loaded into a 32-bit register
2. The least 16 significant bits are packed into a 32-bit register and the most 16 significant bits are packed into another 32-bit register
3. The fourth element of the original array is redundant for our purposes. Hence, it is replaced by zeros.
4. Each FIX value can be represented as a 16-bit number. Pack the three FIX numbers, associated with each line of the code, into a pair of 32-bit registers.
5. Perform a dot product operation on the FIX vectors and the RGB vectors created above, and place the results into a pair of 32-bit registers.
6. Sum the pair of generated 32-bit registers in step 6 with each other. Then, sum the result with the corresponding “constant value” in the code (e.g. ONE_HALF).
7. Perform a shift-right operation by SCALEBITS on the result of step 7
8. Cast the result of 7 into an 8-bit number (i.e. `JSAMPLE)`.

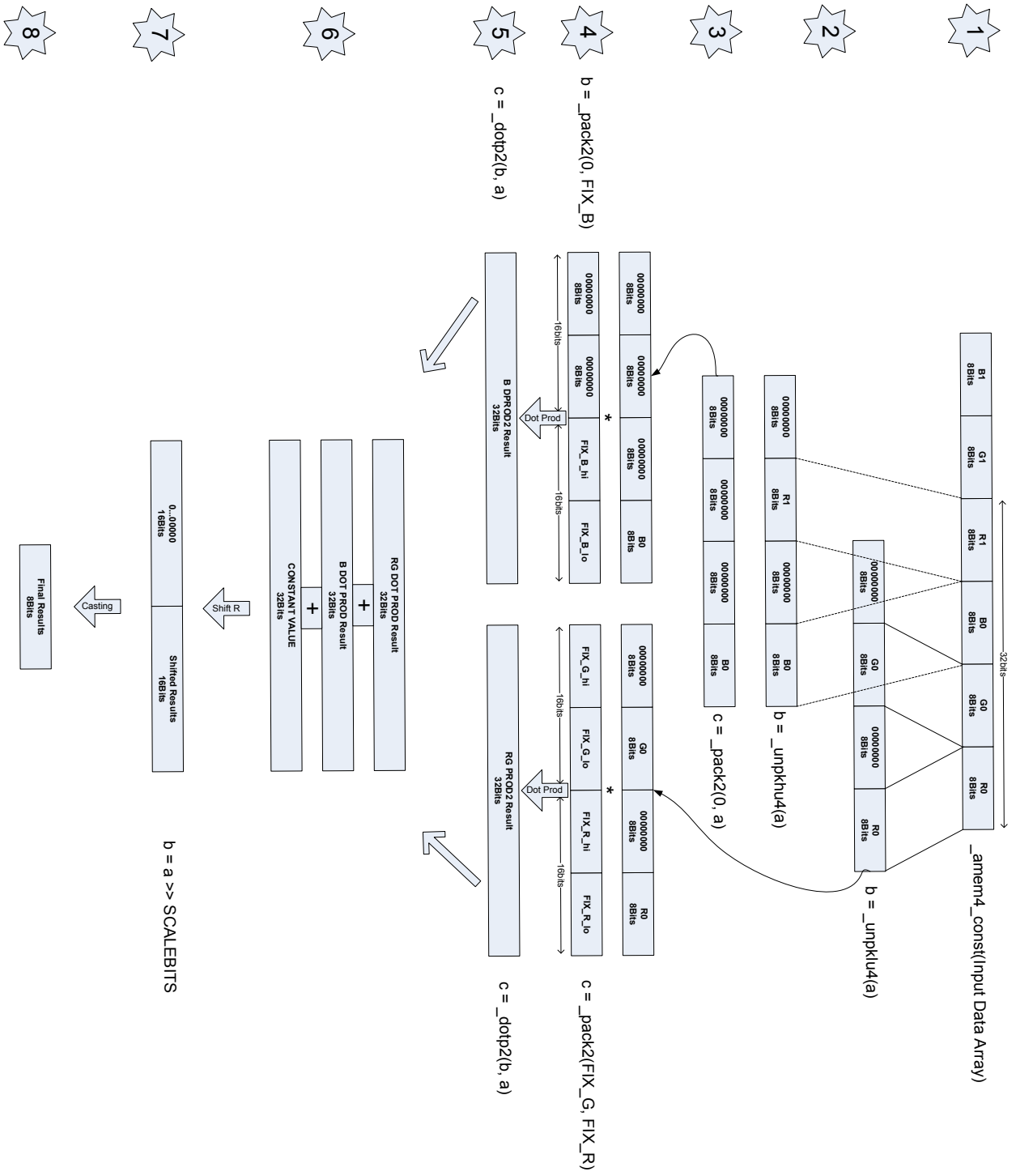


Figure 23: The sequence of DSP intrinsics needed to vectorize each line of code in the color kernel

Conclusion

In this project we had the opportunity to work on three of the most the widely used processors in the Smartphone industry and explore the types of advantages and disadvantages of each. We learned the processors' programming environment using the SAXPY code as the base program, and then modified the CPEG code to explore its execution time efficiency on the three processors: ARM, NEON, and DSP.

In order to obtain the best speedup for CJPEG we decided to makeNEON run the three kernels it runs best, and let the rest of the program run on ARM. The result of this approach was a speedup of about **13%**. Alternatively, we realized that in order to get speedup enhancements on running the CJPEG program on Nokia N900 using the given processor communication setup, we can explore parallel processing of data by running parts of the image data on NEON and parts of it on ARM simultaneously; most likely we do not want to run anything on DSP as it is very unlikely that we would gain any speedup from sending/receiving data from DSP. This way we expect to gain even more than 13% speedup improvement. However, due to the complexity of implanting the code for this approach and the time limitation of this project, we did not manage to explore this interesting alternative.