Assignment 4

Sheil Sarda, Kate Ballard
sheils@seas.upenn.edu, kballard@seas.upenn.edu

October 2, 2020

Exercise 1

(a) My partner and I split up the work so that we each are responsible for one question on our own for the first part of the lab. Then, for questions 4 and 5 we plan to pair program together. We split it up this way to cover the more familiar parts of the lab (2 & 3) alone and work together for the newer vectorization problems.

(b) We estimated that each of 1 & 2 should take 1.5-2 hours. 4 should take about 30 minutes to go through the example NEON speedup and about another 1.5-2 hours for completing question 5.

(c)

- Question 2- about an hour, a bit less than expected.
- Question 3- about an hour and a half to 2 hours
- Question 4- 30 minutes
- Question 5- about 6 hours.

(d) We will walk each other through the steps we took for questions 2 & 3, but because those were more familiar from past labs and information learned in lecture they will be easier to talk through to ensure that we are on the same page. For the more new and complicated 4 and 5 we will pair program together to ensure both of us gain a thorough understanding of vectorization through NEON.

Exercise 2

(a) Effect of compiler optimizations:

Optimization Level	Latency (ns)	Size (bytes)
`-O0`	$1.920 * 10^7$	15641
`-O1`	$4.960 * 10^6$	12600
`-O2`	$4.818 * 10^6$	12504
`-O3`	$3.415 * 10^6$	13200

(b) `Filter_horizontal` innermost loop at optimization level `-O0`.

```
ldr     w0, [sp, 16]
cmp     w0, 6
bgt     .L4
adrp    x0, Coefficients
add     x0, x0, :lo12:Coefficients
ldrsw   x1, [sp, 16]
ldr     w2, [x0, x1, lsl 2]
ldr     w1, [sp, 28]
mov     w0, w1
lsl     w0, w0, 4
sub     w0, w0, w1
lsl     w0, w0, 5
mov     w1, w0
ldr     w0, [sp, 24]
add     w1, w1, w0
ldr     w0, [sp, 16]
add     w0, w1, w0
sxtw    x0, w0
ldr     x1, [sp, 8]
add     x0, x1, x0
ldrb    w0, [x0]
mul     w0, w2, w0
ldr     w1, [sp, 20]
add     w0, w1, w0
str     w0, [sp, 20]
ldr     w0, [sp, 16]
add     w0, w0, 1
str     w0, [sp, 16]
b       .L5
```

(c) `Filter_horizontal` innermost loop at optimization level `-O2`.

```
ldrb    w4, [x5, x0]
ldr     w3, [x6, x0, lsl 2]
add     x0, x0, 1
cmp     x0, 7
madd    w2, w4, w3, w2
bne     .L2
lsr     w2, w2, 8
add     x5, x5, 1
cmp     x8, x5
strb    w2, [x7], 1
bne     .L3
add     x8, x8, 480
add     x1, x1, 474
cmp     x9, x8
bne     .L4
ret
.cfi_endproc
```

(d) The main difference is that the `-O0` compiles to basic assembly instructions whereas the more optimized version utilizes instructions which combine sequences of simple instructions such as store and branch or multiply then add.

As a consequence of this, the more optimized version requires less instructions and external memory accesses, resulting in a significant speed-up.

(e) `-O0` is preferred when readability matters more than performance. If you are trying to debug a programming issue related to pointers or memory issues by going through the assembly code, it would be significantly harder with `-O3` then with `-O0`.

Also, in the case where you are trying to explain the code to someone unfamiliar with the specific hardware you are working with, they are much more likely to understand the less optimized version since the basic assembly commands are fairly universal, whereas the optimized commands may not be.

(f) `Filter_horizontal` innermost loop at optimization level `-O3`.

```
ldrb    w9, [x2, 3]
add     x2, x2, 1
ldrb    w10, [x2]
ldp     w15, w19, [x4]
ldp     w14, w17, [x4, 8]
ldrb    w8, [x2, 4]
mul     w10, w10, w19
ldp     w13, w16, [x4, 16]
mul     w9, w9, w17
ldrb    w3, [x2, 1]
ldrb    w7, [x2, -1]
mul     w8, w8, w16
ldrb    w6, [x2, 3]
madd    w3, w3, w14, w9
ldrb    w5, [x2, 5]
cmp     x18, x2
madd    w7, w7, w15, w10
ldr     w12, [x4, 24]
madd    w6, w6, w13, w8
add     w3, w3, w7
add     w3, w3, w6
madd    w3, w5, w12, w3
lsr     w3, w3, 8
strb    w3, [x11], 1
bne     .L2
```

(g) The `-O3` assembly uses data parallelism and loop unrolling, whereas the `-O2` version does not.

(h) Two key drawbacks of using higher optimization levels:

- Readability is compromised, making it harder to intuitively grasp what is going on at the hardware level.
- Vectorization and data parallelism changes the order in which $\vec{C}$ code is executed, making it harder for the programmer to visualize the control flow of the program.

Exercise 3

(a) Table of baseline results:

	Latency (ns)	Suitability (Y/N)	Ideal Vectorization Speedup
Scale	159336 (4.7%)	N	N/A
Filter_horizontal	617301 (18.2%)	Y	
Filter_vertical	768860 (22.7%)	Y	1.25×
Differentiate	330438 (9.8%)	N	N/A
Compress	1512370 (44.6%)	N	N/A
Overall	3388305 (100%)		

(b) The following loops in the streaming stages of the application have sufficient data parallelism for vectorization:

- `Scale` does not have sufficient data parallelism since the function only copies pixel values from the input to the output with no intermediate computation. This function can only be sped-up by loop unrolling.
- `Filter_horizontal` does have sufficient data parallelism since we are doing a series of multiplication and addition operations for each pixel in the input array.
- `Filter_vertical` also has sufficient data parallelism for the same reason as above.
- `Differentiate` does not have sufficient data parallelism because for each pixel, the function does a series of branches to select the correct arithmetic formula to put into the output array. Parallelizing this function would require many branch predictors and wasted computations.
- `Compress` also does not have sufficient data parallelism because like `Differentiate`, it also has many branches to determine which instructions get executed next.

(c) Critical path lower bound for work done on a single pixel by `Filter_vertical` in terms of compute operations is:

$$\texttt{FILTER\_LENGTH}(7) * (2 \times T_{\texttt{MULT}} + 1 \times T_{\texttt{ADD}})$$

Thus, work done by the entire function, ignoring control flow and offset (bitshift) ops is:

$$(270 * 480) * 7 * (2 \times T_{\texttt{MULT}} + 1 \times T_{\texttt{ADD}})$$

(d)

- Size of input operands is 8 bits since both `Coefficients` and `Input` are unsigned `char` arrays.
- Size of output operands in bits is also 8 bits since `Output` is a char array.
- Size of intermediate result of the function is 32 bits since `Sum` is an `int`.

(e) `Resource Capacity Lower Bound = Total Ops / Operators`. From the previous parts, we know that the total number of operations is:

- (270 * 480 * 7) `ADD` operations.
- (270 * 480 * 7 * 2) `MULT` operations.

```
Sum += Coefficients[i] * Input[(Y + i) * OUTPUT_WIDTH + X];
```

From the Arm Neon source page:

1. Two 64-bit,
2. Four 32-bit,
3. Eight 16-bit, or
4. Sixteen 8-bit integer data elements can be operated on simultaneously using all 128 bits of a Neon register.

Since the maximum space required by the multiplication of 2 8-bit numbers is 16-bits, we will use the $8 \times$ 16-bit lanes on the Arm Neon. Since there are more `MULT` operations than `ADD`, the `RB` bound is dependent on the cycles taken by the multiplication operations:

$$\frac{270 * 480 * 7 * 2}{8} \times T_{\texttt{MULT}}$$

(f) Speedup expected if we achieve the `RB` lower-bound: $T_{\texttt{AFTER}} = \frac{1}{8} \times T_{\texttt{BEFORE}} \implies 8\times$ speedup in this function.

Applying Amdahl's law and using the latencies from the table above as a proxy for % of code:

$$T_{\texttt{AFTER}} = \frac{1}{8} \times 22.7\% + 1 - 22.7\% = 0.80\times$$

Application speed-up = $1/0.80 = 1.25\times$. [[TODO Complete vectorization speedup column]]

(g) Done in code.

(h) Table of baseline with SIMD results.

	Latency (ns)	% faster
Scale	159279	0.04%
Filter_horizontal	270410	56.19%
Filter_vertical	267731	65.18%
Differentiate	329605	0.25%
Compress	1512860	−0.03%
Overall	2540460	25.02%

(i) Multiplies and adds are currently using 32-bits while we only 16-bits. To resolve this issue, we can change the data type from `int` to `short int` which will allow the compiler to make use of a greater number of lanes to parallelize the `Filter` functions further.

(j) `Filter_horizontal` can be modified as follows:

```
void Filter_horizontal(const unsigned char *Input, unsigned char *Output){
  for (int Y = 0; Y < INPUT_HEIGHT; Y++)
    for (int X = 0; X < OUTPUT_WIDTH; X++){
      unsigned short int Sum = 0;
      for (int i = 0; i < FILTER_LENGTH; i++)
        Sum += Coefficients[i] * Input[Y * INPUT_WIDTH + X + i];
      Output[Y * OUTPUT_WIDTH + X] = Sum >> 8;
    }
}
```

`Filter_vertical` can be modified as follows:

```
void Filter_vertical(const unsigned char *Input, unsigned char *Output){
  for (int Y = 0; Y < OUTPUT_HEIGHT; Y++)
    for (int X = 0; X < OUTPUT_WIDTH; X++){
      unsigned short int Sum = 0;
      for (int i = 0; i < FILTER_LENGTH; i++)
        Sum += Coefficients[i] * Input[(Y + i) * OUTPUT_WIDTH + X];
      Output[Y * OUTPUT_WIDTH + X] = Sum >> 8;
    }
}
```

Assembly of the modified function:

```
Filter_vertical(unsigned char const*, unsigned char*):
.LFB22:
.cfi_startproc
stp x29, x30, [sp, -112]!
.cfi_def_cfa_offset 112
.cfi_offset 29, -112
.cfi_offset 30, -104
adrp x2, .LANCHOR0
add x2, x2, :lo12:.LANCHOR0
add x29, sp, 0
.cfi_def_cfa_register 29
mov x4, x1
stp x23, x24, [sp, 48]
add x11, x0, 474
.cfi_offset 23, -64
.cfi_offset 24, -56
mov x24, x1
stp x25, x26, [sp, 64]
add x10, x0, 948
.cfi_offset 25, -48
.cfi_offset 26, -40
add x26, x1, 474
stp x27, x28, [sp, 80]
add x9, x0, 1422
.cfi_offset 27, -32
.cfi_offset 28, -24
add x28, x1, 16
add x8, x0, 1896
```

```
add x1, x2, 28
stp x19, x20, [sp, 16]
add x7, x0, 2370
add x6, x0, 2844
stp x21, x22, [sp, 32]
.cfi_offset 19, -96
.cfi_offset 20, -88
.cfi_offset 21, -80
.cfi_offset 22, -72
mov x5, x0
mov x16, 0
str x1, [x29, 104]
add x27, x0, 2860
mov w18, 474
.p2align 3,,7
.L26:
add x12, x28, x16
add x3, x27, x16
cmp x12, x5
add x1, x26, x16
ccmp x3, x4, 0, hi
cset w3, ls
cmp x1, x2
ldr x1, [x29, 104]
ccmp x4, x1, 2, hi
cset w1, cs
tst w3, w1
beq .L27
neg w17, w5
ands w17, w17, 15
beq .L28
mov x20, 0
mov w15, 0
.p2align 3,,7
.L21:
ldp w22, w13, [x2]
add w15, w15, 1
cmp w15, w17
ldrb w3, [x5, x20]
ldrb w23, [x10, x20]
ldrb w19, [x11, x20]
mul w3, w22, w3
ldp w14, w1, [x2, 8]
ldrb w25, [x8, x20]
madd w13, w13, w19, w3
sub w19, w18, w15
ldr w12, [x2, 16]
mul w14, w14, w23
ldrb w21, [x9, x20]
ldrb w22, [x7, x20]
mul w12, w12, w25
ldp w23, w3, [x2, 20]
```

```
madd w1, w1, w21, w14
ldrb w14, [x6, x20]
add w1, w1, w13
madd w12, w23, w22, w12
add w1, w1, w12
madd w1, w3, w14, w1
ubfx x1, x1, 8, 8
strb w1, [x4, x20]
add x20, x20, 1
bne .L21
.L20:
ldp w21, w20, [x2, 4]
uxtw x1, w17
add x30, x16, 474
add x22, x30, x1
sub w17, w18, w17
ldp w14, w13, [x2, 12]
lsr w25, w17, 4
add x22, x0, x22
ldp w12, w3, [x2, 20]
dup v5.4s, w21
add x21, x16, 948
add x21, x21, x1
dup v4.4s, w20
add x20, x16, 1422
add x21, x0, x21
add x20, x20, x1
ld1r {v6.4s}, [x2]
add x20, x0, x20
dup v3.4s, w14
add x14, x16, 1896
add x14, x14, x1
dup v2.4s, w13
add x13, x16, 2370
add x14, x0, x14
add x13, x13, x1
dup v1.4s, w12
add x12, x16, 2844
add x16, x1, x16
add x1, x12, x1
add x23, x0, x16
dup v0.4s, w3
add x12, x0, x1
xtn v17.4h, v5.4s
xtn2 v17.8h, v5.4s
xtn v16.4h, v4.4s
xtn2 v16.8h, v4.4s
add x13, x0, x13
xtn v18.4h, v6.4s
xtn2 v18.8h, v6.4s
add x16, x24, x16
xtn v7.4h, v3.4s
```

```
xtn2 v7.8h, v3.4s
mov x1, 0
mov w3, 0
xtn v6.4h, v2.4s
xtn2 v6.8h, v2.4s
xtn v5.4h, v1.4s
xtn2 v5.8h, v1.4s
xtn v4.4h, v0.4s
xtn2 v4.8h, v0.4s
.p2align 3,,7
.L22:
ldr q0, [x22, x1]
add w3, w3, 1
cmp w3, w25
ldr q21, [x23, x1]
ldr q20, [x21, x1]
ushll v1.8h, v0.8b, 0
ldr q19, [x20, x1]
ushll2 v0.8h, v0.16b, 0
ushll v28.8h, v21.8b, 0
ldr q3, [x14, x1]
ushll2 v21.8h, v21.16b, 0
mul v1.8h, v1.8h, v17.8h
ushll v27.8h, v20.8b, 0
ldr q22, [x13, x1]
ushll2 v20.8h, v20.16b, 0
mul v0.8h, v0.8h, v17.8h
ushll v26.8h, v19.8b, 0
ldr q2, [x12, x1]
ushll2 v19.8h, v19.16b, 0
ushll v25.8h, v3.8b, 0
mla v1.8h, v28.8h, v18.8h
ushll2 v3.8h, v3.16b, 0
ushll v24.8h, v22.8b, 0
mla v0.8h, v21.8h, v18.8h
ushll2 v22.8h, v22.16b, 0
ushll v23.8h, v2.8b, 0
ushll2 v2.8h, v2.16b, 0
mla v1.8h, v27.8h, v16.8h
mla v0.8h, v20.8h, v16.8h
mla v1.8h, v26.8h, v7.8h
mla v0.8h, v19.8h, v7.8h
mla v1.8h, v25.8h, v6.8h
mla v0.8h, v3.8h, v6.8h
mla v1.8h, v24.8h, v5.8h
mla v0.8h, v22.8h, v5.8h
mla v1.8h, v23.8h, v4.8h
mla v0.8h, v2.8h, v4.8h
ushll v3.4s, v1.4h, 0
ushll2 v1.4s, v1.8h, 0
ushll v2.4s, v0.4h, 0
ushll2 v0.4s, v0.8h, 0
```

```
sshr v3.4s, v3.4s, 8
sshr v1.4s, v1.4s, 8
sshr v2.4s, v2.4s, 8
sshr v0.4s, v0.4s, 8
xtn v19.4h, v3.4s
xtn2 v19.8h, v1.4s
xtn v1.4h, v2.4s
xtn2 v1.8h, v0.4s
xtn v0.8b, v19.8h
xtn2 v0.16b, v1.8h
str q0, [x16, x1]
add x1, x1, 16
bcc .L22
and w1, w17, -16
cmp w17, w1
add w15, w1, w15
mvn w1, w1
beq .L25
sxtw x22, w15
add w19, w19, w1
add x23, x22, 1
add x25, x19, x23
.p2align 3,,7
.L24:
ldp w13, w1, [x2]
cmp x25, x23
ldrb w21, [x5, x22]
ldrb w17, [x11, x22]
ldrb w20, [x10, x22]
mul w13, w13, w21
ldr w12, [x2, 8]
ldrb w16, [x9, x22]
madd w13, w1, w17, w13
ldr w1, [x2, 12]
mul w12, w12, w20
ldrb w3, [x7, x22]
ldr w19, [x2, 20]
madd w12, w1, w16, w12
ldrb w15, [x8, x22]
ldr w1, [x2, 16]
mul w3, w19, w3
ldrb w14, [x6, x22]
madd w3, w1, w15, w3
ldr w1, [x2, 24]
add w15, w12, w13
add w15, w15, w3
madd w14, w1, w14, w15
ubfx x14, x14, 8, 8
strb w14, [x4, x22]
mov x22, x23
add x23, x23, 1
bne .L24
```

```
.L25:
sub x1, x30, #122880
add x4, x4, 474
subs x1, x1, #2256
add x5, x5, 474
add x11, x11, 474
add x10, x10, 474
add x9, x9, 474
add x8, x8, 474
add x7, x7, 474
add x6, x6, 474
mov x16, x30
bne .L26
ldp x19, x20, [sp, 16]
ldp x21, x22, [sp, 32]
ldp x23, x24, [sp, 48]
ldp x25, x26, [sp, 64]
ldp x27, x28, [sp, 80]
ldp x29, x30, [sp], 112
.cfi_remember_state
.cfi_restore 30
.cfi_restore 29
.cfi_restore 27
.cfi_restore 28
.cfi_restore 25
.cfi_restore 26
.cfi_restore 23
.cfi_restore 24
.cfi_restore 21
.cfi_restore 22
.cfi_restore 19
.cfi_restore 20
.cfi_def_cfa 31, 0
ret
.p2align 3,,7
.L27:
.cfi_restore_state
mov x20, 0
.p2align 3,,7
.L19:
ldp w19, w1, [x2, 8]
ldrb w15, [x10, x20]
ldrb w17, [x5, x20]
ldp w25, w13, [x2]
mul w15, w19, w15
ldrb w3, [x8, x20]
ldp w14, w12, [x2, 16]
mul w17, w25, w17
ldrb w22, [x9, x20]
ldrb w23, [x11, x20]
mul w14, w14, w3
ldrb w21, [x7, x20]
```

```
madd w1, w1, w22, w15
ldrb w19, [x6, x20]
madd w13, w13, w23, w17
ldr w3, [x2, 24]
madd w12, w12, w21, w14
add w1, w1, w13
add w1, w1, w12
madd w1, w3, w19, w1
ubfx x1, x1, 8, 8
strb w1, [x4, x20]
add x20, x20, 1
cmp x20, 474
bne .L19
add x30, x16, 474
b .L25
.L28:
mov w19, 474
mov w15, 0
b .L20
.cfi_endproc
```

(k) Table of baseline with SIMD modified results.

	Latency (ns)	% faster
Scale	156163	1.96%
Filter_horizontal	175546	35.08%
Filter_vertical	179408	32.99%
Differentiate	330352	−0.23%
Compress	1502770	0.67%
Overall	2344950	7.70%

Exercise 4

(a) No submission required.

(b) Done.

(c)

```
NEON add3 takes 72.000000ns
C add3 takes 108.000000ns
```

Time taken by `NEON` is 33.3% less than the time taken by `C`.

(d) Based on the assembly code, the `Neon` version is able to achieve a speedup by using special datatypes such as `v16.16b` which enable vectorization. The `Neon` version also unrolls the loop, unlike the `C` code, leading to an additional speed-up. Assembly of example:

```
0x0000000000400db0 <+496>: bl 0x400a60 <_ZNSt6chrono3_V212system_clock3nowEv@plt>
## Neon assembly
0x0000000000400db4 <+500>: ldr q16, [x29, #64]
```

```
0x0000000000400db8 <+504>: movi v0.16b, #0x3
0x0000000000400dbc <+508>: mov x19, x0
0x0000000000400dc0 <+512>: ldr q7, [x29, #80]
0x0000000000400dc4 <+516>: ldr q6, [x29, #96]
0x0000000000400dc8 <+520>: ldr q5, [x29, #112]
0x0000000000400dcc <+524>: add v16.16b, v16.16b, v0.16b
0x0000000000400dd0 <+528>: ldr q4, [x29, #128]
0x0000000000400dd4 <+532>: add v7.16b, v7.16b, v0.16b
0x0000000000400dd8 <+536>: ldr q3, [x29, #144]
0x0000000000400ddc <+540>: add v6.16b, v6.16b, v0.16b
0x0000000000400de0 <+544>: str q16, [x29, #64]
0x0000000000400de4 <+548>: ldr q2, [x29, #160]
0x0000000000400de8 <+552>: add v5.16b, v5.16b, v0.16b
0x0000000000400dec <+556>: str q7, [x29, #80]
0x0000000000400df0 <+560>: ldr q1, [x29, #176]
0x0000000000400df4 <+564>: add v4.16b, v4.16b, v0.16b
0x0000000000400df8 <+568>: str q6, [x29, #96]
0x0000000000400dfc <+572>: add v3.16b, v3.16b, v0.16b
0x0000000000400e00 <+576>: str q5, [x29, #112]
0x0000000000400e04 <+580>: add v2.16b, v2.16b, v0.16b
0x0000000000400e08 <+584>: str q4, [x29, #128]
0x0000000000400e0c <+588>: add v0.16b, v1.16b, v0.16b
0x0000000000400e10 <+592>: str q3, [x29, #144]
0x0000000000400e14 <+596>: str q2, [x29, #160]
0x0000000000400e18 <+600>: str q0, [x29, #176]
0x0000000000400e1c <+604>: bl 0x400a60 <_ZNSt6chrono3_V212system_clock3nowEv@plt>
0x0000000000400e20 <+608>: sub x19, x0, x19
0x0000000000400e24 <+612>: movi d1, #0x0
0x0000000000400e28 <+616>: adrp x1, 0x401000 <frame_dummy>
0x0000000000400e2c <+620>: str x0, [x29, #56]
0x0000000000400e30 <+624>: add x0, x1, #0x8f0
0x0000000000400e34 <+628>: scvtf d0, x19
0x0000000000400e38 <+632>: fadd d0, d0, d1
0x0000000000400e3c <+636>: bl 0x400ba0 <printf@plt>
## wrapped up the add_neon-- starting addC
0x0000000000400e40 <+640>: bl 0x400a60 <_ZNSt6chrono3_V212system_clock3nowEv@plt>
0x0000000000400e44 <+644>: movi v1.4s, #0x3
0x0000000000400e48 <+648>: add x1, x29, #0xc0
0x0000000000400e4c <+652>: mov x19, x0
0x0000000000400e50 <+656>: add x2, x29, #0x2c0
0x0000000000400e54 <+660>: str x0, [x29, #48]
0x0000000000400e58 <+664>: ldr q0, [x1]
0x0000000000400e5c <+668>: add v0.4s, v0.4s, v1.4s
0x0000000000400e60 <+672>: str q0, [x1], #16
0x0000000000400e64 <+676>: cmp x1, x2
0x0000000000400e68 <+680>: b.ne 0x400e58 <main+664>  // b.any-- has a loop, unlike neon
## wrapped up addC
0x0000000000400e6c <+684>: bl 0x400a60 <_ZNSt6chrono3_V212system_clock3nowEv@plt>
```

Exercise 5

(a) Output and speedup compared to the baseline.

```
Average latency of Scale per loop iteration is: 158534 ns.
Average latency of Filter_horizontal per loop iteration is: 177222 ns.
Average latency of Filter_vertical per loop iteration is: 337984 ns.
Average latency of Differentiate per loop iteration is: 330444 ns.
Average latency of Compress per loop iteration is: 1.50805e+06 ns.
Average latency of each loop iteration is: 2.5127e+06 ns.
Application completed successfully.
```

Compared to the baseline, the neon intrinsic implementation takes 56% less time.

Assembly of neon intrinsic implementation of `Filter_vertical`

```
Dump of assembler code for function _Z15Filter_verticalPKhPh:
0x00000000004019a0 <+0>: adrp x3, 0x420000
0x00000000004019a4 <+4>: add x2, x3, #0xcc0
0x00000000004019a8 <+8>: str d8, [sp, #-16]!
0x00000000004019ac <+12>: sub x1, x1, #0x1f, lsl #12
0x00000000004019b0 <+16>: add x4, x0, #0x1f, lsl #12
=> 0x00000000004019b4 <+20>: ldrh w14, [x3, #3264]
0x00000000004019b8 <+24>: add x12, x0, #0x1f, lsl #12
0x00000000004019bc <+28>: mov x11, #0xfffffffffffff0c14     // #-62444
0x00000000004019c0 <+32>: movk x11, #0xfffe, lsl #16
0x00000000004019c4 <+36>: ldrh w13, [x2, #4]
0x00000000004019c8 <+40>: mov x10, #0xfffffffffffff0dee     // #-61970
0x00000000004019cc <+44>: movk x10, #0xfffe, lsl #16
0x00000000004019d0 <+48>: mov x9, #0xfffffffffffff0fc8     // #-61496
0x00000000004019d4 <+52>: movk x9, #0xfffe, lsl #16
0x00000000004019d8 <+56>: ldrh w3, [x2, #8]
0x00000000004019dc <+60>: mov x8, #0xfffffffffffff11a2     // #-61022
0x00000000004019e0 <+64>: movk x8, #0xfffe, lsl #16
0x00000000004019e4 <+68>: dup v24.8h, w14
0x00000000004019e8 <+72>: mov x7, #0xffffffffffff137c     // #-60548
0x00000000004019ec <+76>: movk x7, #0xfffe, lsl #16
0x00000000004019f0 <+80>: ldrh w2, [x2, #12]
0x00000000004019f4 <+84>: mov x6, #0xffffffffffff1556     // #-60074
0x00000000004019f8 <+88>: movk x6, #0xfffe, lsl #16
0x00000000004019fc <+92>: dup v23.8h, w13
0x0000000000401a00 <+96>: mov x5, #0xffffffffffff1730     // #-59600
0x0000000000401a04 <+100>: movk x5, #0xfffe, lsl #16
0x0000000000401a08 <+104>: sub x1, x1, #0x3ec
0x0000000000401a0c <+108>: add x4, x4, #0x3ec
0x0000000000401a10 <+112>: dup v22.8h, w3
0x0000000000401a14 <+116>: add x12, x12, #0x5cc
0x0000000000401a18 <+120>: dup v21.8h, w2
0x0000000000401a1c <+124>: nop
0x0000000000401a20 <+128>: ldr q20, [x4, x11]
0x0000000000401a24 <+132>: sub x3, x4, x0
0x0000000000401a28 <+136>: add x2, x4, x5
0x0000000000401a2c <+140>: add x3, x1, x3
0x0000000000401a30 <+144>: ldr q6, [x4, x10]
```

```
0x0000000000401a34 <+148>: ldr q5, [x4, x9]
0x0000000000401a38 <+152>: ldr q4, [x4, x8]
0x0000000000401a3c <+156>: ldr q3, [x4, x7]
0x0000000000401a40 <+160>: ldr q2, [x4, x6]
0x0000000000401a44 <+164>: nop
0x0000000000401a48 <+168>: mov d0, v20.d[1]
0x0000000000401a4c <+172>: mov d1, v20.d[0]
0x0000000000401a50 <+176>: ldr q26, [x2]
0x0000000000401a54 <+180>: mov d8, v6.d[1]
0x0000000000401a58 <+184>: mov d25, v2.d[1]
0x0000000000401a5c <+188>: mov d31, v5.d[1]
0x0000000000401a60 <+192>: mov d18, v3.d[1]
0x0000000000401a64 <+196>: add x2, x2, #0x1da
0x0000000000401a68 <+200>: mov d19, v6.d[0]
0x0000000000401a6c <+204>: mov d30, v2.d[0]
0x0000000000401a70 <+208>: cmp x4, x2
0x0000000000401a74 <+212>: mov d17, v5.d[0]
0x0000000000401a78 <+216>: mov d29, v3.d[0]
0x0000000000401a7c <+220>: mov d16, v4.d[1]
0x0000000000401a80 <+224>: mov d7, v4.d[0]
0x0000000000401a84 <+228>: uxtl v16.8h, v16.8b
0x0000000000401a88 <+232>: mov d28, v26.d[1]
0x0000000000401a8c <+236>: mov d27, v26.d[0]
0x0000000000401a90 <+240>: uxtl v7.8h, v7.8b
0x0000000000401a94 <+244>: uaddl v25.8h, v8.8b, v25.8b
0x0000000000401a98 <+248>: uaddl v18.8h, v31.8b, v18.8b
0x0000000000401a9c <+252>: uaddl v19.8h, v19.8b, v30.8b
0x0000000000401aa0 <+256>: uaddl v17.8h, v17.8b, v29.8b
0x0000000000401aa4 <+260>: uaddl v0.8h, v0.8b, v28.8b
0x0000000000401aa8 <+264>: uaddl v1.8h, v1.8b, v27.8b
0x0000000000401aac <+268>: mov v20.16b, v6.16b
0x0000000000401ab0 <+272>: mov v6.16b, v5.16b
0x0000000000401ab4 <+276>: mov v5.16b, v4.16b
0x0000000000401ab8 <+280>: mov v4.16b, v3.16b
0x0000000000401abc <+284>: mov v3.16b, v2.16b
0x0000000000401ac0 <+288>: mov v2.16b, v26.16b
0x0000000000401ac4 <+292>: mul v0.8h, v0.8h, v24.8h
0x0000000000401ac8 <+296>: mla v0.8h, v25.8h, v23.8h
0x0000000000401acc <+300>: mul v1.8h, v1.8h, v24.8h
0x0000000000401ad0 <+304>: mla v0.8h, v18.8h, v22.8h
0x0000000000401ad4 <+308>: mla v1.8h, v19.8h, v23.8h
0x0000000000401ad8 <+312>: mla v0.8h, v16.8h, v21.8h
0x0000000000401adc <+316>: mla v1.8h, v17.8h, v22.8h
0x0000000000401ae0 <+320>: ushr v0.8h, v0.8h, #8
0x0000000000401ae4 <+324>: mla v1.8h, v7.8h, v21.8h
0x0000000000401ae8 <+328>: xtn v0.8b, v0.8h
0x0000000000401aec <+332>: ushr v1.8h, v1.8h, #8
0x0000000000401af0 <+336>: xtn v1.8b, v1.8h
0x0000000000401af4 <+340>: mov d7, v1.d[0]
0x0000000000401af8 <+344>: mov v7.d[1], v0.d[0]
0x0000000000401afc <+348>: str q7, [x3]
0x0000000000401b00 <+352>: add x3, x3, #0x1da
```

```
0x0000000000401b04 <+356>: b.ne 0x401a48 <_Z15Filter_verticalPKhPh+168>  // b.any
0x0000000000401b08 <+360>: add x4, x4, #0x8
0x0000000000401b0c <+364>: cmp x12, x4
0x0000000000401b10 <+368>: b.ne 0x401a20 <_Z15Filter_verticalPKhPh+128>  // b.any
0x0000000000401b14 <+372>: ldr d8, [sp], #16
0x0000000000401b18 <+376>: ret
End of assembler dump.
```

(b) The neon intrinsic implementation deals with a number of data elements not divisible by number of vector lanes by trying to minimize the overlap.

As we can see from the of `Filter_vertical` with vectorization, the overlap between the two sections of computation is in the center of the input.

The overlap causes us to not attain the optimal speedup, but still provides a substantial speedup from the baseline.

(c) `Neon` processes the data in 8 16-bit lanes. Within a lane, data is multiplied by the coefficients matrix and accumulated with all the other values in the same lane. Vectors are horizontal rows of 16-bits each, and there are 8 of them from top to bottom.

This vectorization allows us to process data with a speedup of 16× compared to the baseline.

The flipped order of for loops is because bigger rectangle moves differently from unvectorized computation.

(d) Differences:

- `Neon` assembly has duplicated instructions adjacent to each other since the vectorized instructions are executed in parallel.
- `Neon` makes use of special Arm registers like `V-regs` for the vectorized operations.

(e) Changes made to the innermost loop of `Filter_vertical` using intrinsics:

- Using the multiply accumulate function to multiply data with coefficients
- Using a shared variable to accumulate the intermediate results to cut down on add operations afterwards.

```
uint8x16_t Data6 = vld1q_u8(Input + (Y + 6) * OUTPUT_WIDTH + X);
uint16x8_t SumH;
uint16x8_t SumL;
SumH = vmlaq_u16(SumH, vmovl_u8(vget_high_u8(Data0)), Coef0);
SumH = vmlaq_u16(SumH, vmovl_u8(vget_high_u8(Data1)), Coef1);
SumH = vmlaq_u16(SumH, vmovl_u8(vget_high_u8(Data2)), Coef2);
SumH = vmlaq_u16(SumH, vmovl_u8(vget_high_u8(Data3)), Coef3);
SumH = vmlaq_u16(SumH, vmovl_u8(vget_high_u8(Data4)), Coef2);
SumH = vmlaq_u16(SumH, vmovl_u8(vget_high_u8(Data5)), Coef1);
SumH = vmlaq_u16(SumH, vmovl_u8(vget_high_u8(Data6)), Coef0);
SumH = vshrq_n_u16(SumH, 8);
uint8x8_t ResultH = vmovn_u16(SumH);
SumL = vmlaq_u16(SumL, vmovl_u8(vget_low_u8(Data0)), Coef0);
SumL = vmlaq_u16(SumL, vmovl_u8(vget_low_u8(Data1)), Coef1);
```

```
SumL = vmlaq_u16(SumL, vmovl_u8(vget_low_u8(Data2)), Coef2);
SumL = vmlaq_u16(SumL, vmovl_u8(vget_low_u8(Data3)), Coef3);
SumL = vmlaq_u16(SumL, vmovl_u8(vget_low_u8(Data4)), Coef2);
SumL = vmlaq_u16(SumL, vmovl_u8(vget_low_u8(Data5)), Coef1);
SumL = vmlaq_u16(SumL, vmovl_u8(vget_low_u8(Data6)), Coef0);
SumL = vshrq_n_u16(SumL, 8);
uint8x8_t ResultL = vmovn_u16(SumL);
vst1q_u8(Output + Y * OUTPUT_WIDTH + X, vcombine_u8(ResultL, ResultH));
Data0 = Data1;
Data1 = Data2;
Data2 = Data3;
Data3 = Data4;
Data4 = Data5;
Data5 = Data6;
```

Output after running the application:

```
Output:
Total latency of Scale is: 1.64106e+07 ns.
Total latency of Filter_horizontal is: 6.17498e+07 ns.
Total latency of Filter_vertical is: 3.68338e+07 ns.
Total latency of Differentiate is: 3.3021e+07 ns.
Total latency of Compress is: 1.50484e+08 ns.
Total time taken by the loop is: 2.98538e+08 ns.
------------------------------------------------------------
Average latency of Scale per loop iteration is: 164106 ns.
Average latency of Filter_horizontal per loop iteration is: 617498 ns.
Average latency of Filter_vertical per loop iteration is: 368338 ns.
Average latency of Differentiate per loop iteration is: 330210 ns.
Average latency of Compress per loop iteration is: 1.50484e+06 ns.
Average latency of each loop iteration is: 2.98538e+06 ns.
Success
```

(f) This modification does not result in the same magnitude of speedup calculated in $3(k)$ because of the compiler using a lot of branch logic in `Neon` then in the automatically vectorized program.

(g) Results after our speed-up:

```
Total latency of Scale is: 1.68585e+07 ns.
Total latency of Filter_horizontal is: 6.18106e+07 ns.
Total latency of Filter_vertical is: 2.61425e+07 ns.
Total latency of Differentiate is: 3.30877e+07 ns.
Total latency of Compress is: 3.1347e+08 ns.
Total time taken by the loop is: 4.51402e+08 ns.
------------------------------------------------------------
Average latency of Scale per loop iteration is: 168585 ns.
Average latency of Filter_horizontal per loop iteration is: 618106 ns.
Average latency of Filter_vertical per loop iteration is: 261425 ns.
Average latency of Differentiate per loop iteration is: 330877 ns.
Average latency of Compress per loop iteration is: 3.1347e+06 ns.
Average latency of each loop iteration is: 4.51402e+06 ns.
Success
```

(h) The `RB` was an 8× speedup for `Filter_vertical`, but in both part 3 and 5, our achieved speedup was approximately 3× speedup.

(i) Automatic vectorization yielded a much better speedup without any tweaks, but we were able to manually achieve the same optimization by minimizing the number of instrinics used.

Exercise 6

	Original estimate	Actual time
Question 2	1.5-2 hours	1 hour
Question 3	1.5-2 hours	∼1.5 hours
Question 4	30 minutes	∼30 minutes
Question 5	1.5-2 hours	6 hours

(a) Question 5 took much longer than expected because we had to go line by line in the provided code to find any opportunities for optimization or using fewer intrinsics. This required a lot of referencing the cited documentation and trying out new intrinsics to see if the code still compiled.

(b) We were able to complete the tasks pretty much as they were originally assigned. I completed 3 on my own, and my partner completed 2 on her own. We came back together to discuss and complete the end of 3 to make sure we were on the same page with the optimization needed there. 4 and 5 we did together as intended using pair programming over zoom.

(c) By working together we were both able to help each other find the explanations we needed to better understand vectorization and intrinsics.

(d) Our combined areas of expertise were broad, covering both CS and EE.

(e) AWS Usage: