RISC-V for AI

Progress

Week 5:

- Agenda:

  1. Compiler Optimization
  2. Execution on Raspberry Pi3
  3. Profiling Results for Convolution, Maxpooling and Matrix-Vector Multiplication operations

Compiler Optimization

- Usage of gcc flag: **-O3**
- Results in (1/4) execution time and (1/8) code size
- The Assembly Code reveals the usage of SIMD Instructions in x86 and NEON-Vector Extension Insturctions in ARM

gcc -O option flag

Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

https://www.rapidtables.com/code/linux/gcc/gcc-o.html

Intel Corei3 8145U CPU Specs.

No. of Cores	64-bit Dual -core/Two computing threads per core
CPU Clock Speed	2.1 Ghz to 3.9 GHz
Memory	12 GB DDR4
Cache Organization	L1: 128 KiB ->L1i = 64 KiB , L1d = 64 KiB, 8 -way Set Associative L2: 512 KiB ->4 Way Set Associative L3: 4 MiB -> 12 Way Set Associative
ISA Extensions	Supports SIMD, Vector Extensions

https://en.wikichip.org/wiki/intel/core_i3/i3-8145u

ARM Cortex A53 (BCM2837 SoC)

CPU	4 x ARM Cortex A-53, 1.2 GHz
RAM	1GB LPDDR2
Cortex A53 Processor	2-wire decode superscalar, 8-stage pipelined , in-order execution (based on ARM v8-A ISA)
ISA Extension Support	Floating Point, DSP, NEON SIMD Instructions
Cache	64-byte lines L1 -8KiB to 64 KiB L2 – 128 KiB to 2MiB

Comparison of Execution Times

Function	x86	Cortex A53	x86 with 03	A53 with O3
CNN_main()	228 205 668	227 123 968	59 625 565	59 916 093
Conv2d_2() Image =[12 , 12] kernel = [5,5,32,64]	201 948 748	196 015 915	57 662 471	57 727 822
Conv2d_1() Image = [28, 28] Kernel = [5,5,1,32]	24 691 568	29 450 033	1 703 225	1 833 513
Maxpool2_1() Inp = [24,24]	638 077	671 422	109 532	128 582
Perceptron(2-layer) 1024 -->10	391 168	411 898	125 477	197 073
relu	10	14	2	5.5
Maxpool2_2 Inp = [8,8]	101 441	173 322	24 474	28 728
max4	21	18	4	10

Profiling Analysis for CNN

- Analysis of Innermost Loop of Convolution

```
{
        filt_offset = x*((FILT1_DIM2)*(FILT1_DIM3)*(FILT1_DIM4))\
                + y*((FILT1_DIM3)*(FILT1_DIM4))\
                + z*(FILT1_DIM4)\
                + output_depth;

        img_offset  = (output_row+x)*((IMG_DIM2)*(IMG_DIM3))\
                + (output_col+y)*(IMG_DIM3)\
                + z;

        sum+= filter[filt_offset] * img[img_offset];
  }
```

No. of Iterations : 3 276 800

Inner Loop Analysis contd...

- **No. of Iterations** : 3 276 800

Parameter	x86	Cortex-A53	X86 with O3	A53 with O3
No. of instructions	54	50	7	7
CPI	1 for all except 11.10 for 1 movss	1 for all except 11.097 for 1 vld	1 for all except 11.10 for 1 movss	1 for all except 11.097 for 1 vld

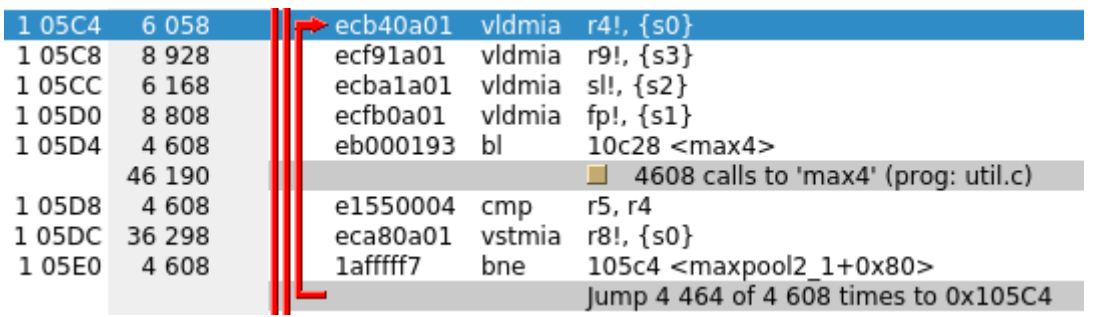
Observations:

1. For computing the Matrix offsets, both x86 and ARM compilers have replaced all multiplications with a combination of SLL and ADD
2. All Loads appear to take 1 clock cycle for L1- Cache hit.
3. Instruction distribution: LD (13), ADD(12), LSL(8), STR(4), MOV
4. Possibility of Improvement: Even with -O3 optimization, the loop executes for 3276800 times. Possibly imptrove by unrolling the inner loop 5 x 5 x 32 (800 - 1) times by accepting image and filters as vectors.
5. -O3 uses instructions such as vmla (mac) , vpust/pop (multiple regs to mem), vldmia(load multiple)

Profiling Analysis for Maxpool Operation

```
for (int row=0; row<MAXPOOL2_DIM1;row++)
 {
     for (int col=0;col<MAXPOOL2_DIM2;col++)
     {
         for (int depth=0; depth<MAXPOOL2_DIM3;depth++)
         {
             e1 =  row*2*(CONV2_DIM2)*(CONV2_DIM3) + col*2*(CONV2_DIM3) + depth;    //00
             e2 =  row*2*(CONV2_DIM2)*(CONV2_DIM3) + (col*2 + 1)*(CONV2_DIM3) + depth;//01
             e3 =  (row*2 + 1)*(CONV2_DIM2)*(CONV2_DIM3) + col*2*(CONV2_DIM3) + depth;   //10
             e4 =  (row*2 + 1)*(CONV2_DIM2)*(CONV2_DIM3) + (col*2 + 1)*(CONV2_DIM3) + depth;//11

             op_offset = row*(MAXPOOL2_DIM2)*(MAXPOOL2_DIM3) + col*(MAXPOOL2_DIM3) + depth;
             output[op_offset] = max4(input[e1],input[e2],input[e3],input[e4]);
         }
     }
 }
```



Observation : A53 and x86 use Vector and SIMD instructions respectively with Compiler Optimization (-O3)

Matrix Vector Multiplication [1 x 300] x [300 x 300]

```
 3 void vmatmul(float vect[], float mat[],int vdim, int mat_dim2, float res[])
 4 {
 5     // Multiply [ 1 x dim1] x [dim1 x dim2]
 6     int mat_offset;
 7     float mac;
 8     float prod, mult_1, mult_2;
 9     for (int i=0;i<mat_dim2;i++)
10     {
11         mac=0;
12         for(int j=0;j<vdim;j++)
13         {
14             mac += vect[j] * (*(mat + mat_dim2*i + j));
15         }
16         *(res+i) = mac;    //store result into product
17     }
18 }
```

```
1 0568          1     e92d4070  push    {r4, r5, r6, lr}                util.c:9
1 056C          1     e1a05103  lsl     r5, r3, #2                      util.c:9
1 0570          1     e59d4010  ldr     r4, [sp, #16]                   util.c:9
1 0574          1     e080e102  add     lr, r0, r2, lsl #2              util.c:9
1 0578          1     e0846005  add     r6, r4, r5                      util.c:9
1 057C        300     e3520000  cmp     r2, #0                          util.c:9
1 0580        420     eddf7a0c  vldr    s15, [pc, #48]      ; 105b8 <...util.c:9
1 0584        300     c1a0c001  movgt   ip, r1                          util.c:9
1 0588        300     c1a03000  movgt   r3, r0                          util.c:9
1 058C        300     da000004  ble     105a4 <vmatmul+0x44>            util.c:9
1 0590     92 090     ecf36a01  vldmia  r3!, {s13}                      util.c:14
1 0594    708 860     ecbc7a01  vldmia  ip!, {s14}                      util.c:14
1 0598     90 000     e15e0003  cmp     lr, r3                          util.c:14
1 059C     90 000     ee467a87  vmla.f32 s15, s13, s14                  util.c:14
1 05A0     90 000     1affffa   bne     10590 <vmatmul+0x30>            util.c:14
                                        Jump 89 700 of 90 000 times to 0x105...
1 05A4      2 390     ece47a01  vstmia  r4!, {s15}                      util.c:16
1 05A8        300     e0811005  add     r1, r1, r5                      util.c:16
1 05AC        300     e1560004  cmp     r6, r4                          util.c:16
1 05B0        300     1afffff1  bne     1057c <vmatmul+0x1c>            util.c:16
                                        Jump 299 of 300 times to 0x1057C
1 05B4          1     e8bd8070  pop     {r4, r5, r6, pc}                util.c:16
1 05B8                00000000  .word   0x00000000
1 05BC                ee200a20  vmul.f32 s0, s0, s1
```

Cache Memory Profiling

Cache Level-wise Profile	x86	Cortex-A53	X86 with -O3	A53 with -03
I refs	193 809 805	192 768 160	25 663 967	25 987 578
I1 misses	882	1139	879	1 121
LLi Misses	877	832	875	812
D refs	82 153 327	86 582 493	7 705 355	7 872 700
D1 misses	3 350 271 (4.1 %)	3 354 623 (3.8 %)	3 355 431 (43.5%)	3 355 668(42.6 %)
LLd misses	7 320	7122	8197	7 086

Inference: Results are similar and cache memory performance is sufficiently good for Convolution

Week 6

- **Week 5 :Action**
- Run Neon Assembly for understaind its performance better
- **Agenda**
- Survey of ARM based Vector Extensions
- Study of ARM’s Edge ML Inference SoC

ARM SIMD ISAs

Scalable Vector Extension

Scalable Vector Extension (SVE) is a vector extension for AArch64 execution mode for the A64 instruction set of the Armv8 architecture. Unlike other SIMD architectures, SVE does not define the size of the vector registers. but constrains it to a range of possible values, from a minimum of 128 bits up to a maximum of 2048 in 128-bit wide units. Therefore, any CPU vendor can implement the extension by choosing the vector register size that better suits the workloads the CPU is targeting.

Read more

Arm Helium technology

Helium is an M-Profile Vector Extension (MVE) that will deliver a significant performance uplift for machine learning and signal processing.

Read more

Arm Neon technology

Arm Neon technology is an advanced Single Instruction Multiple Data (SIMD) architecture extension for the Arm Cortex-A processor series and for Cortex-R52 and Cortex-R82 processors.

Read more

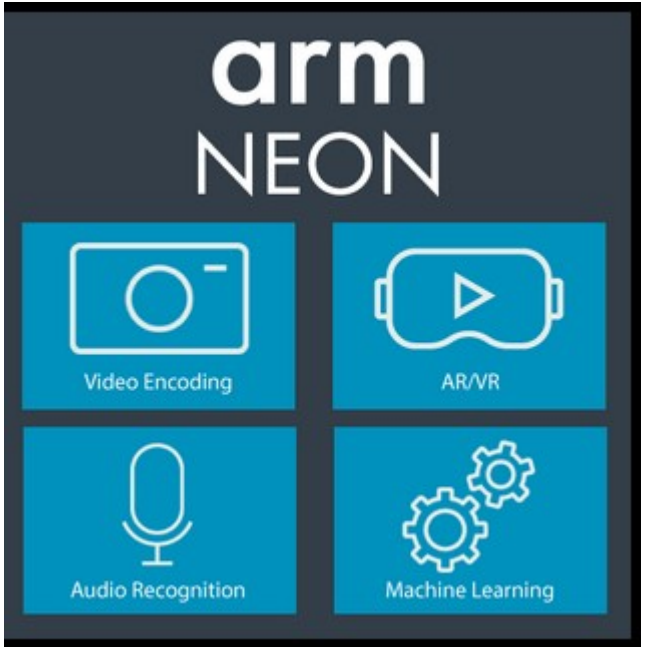
https://developer.arm.com/architectures/instruction-sets/simd-isas

Scalable Vector Extension (SVE)

- SVE enables vectorization of loops which would either be impossible or not beneficial to vectorize with Neon.
- **Scalable Vector Length**
- Vector code allows each implementation to automatically choose its vector length, provided it is a multiple of 128 bits and does not exceed the architectural maximum of 2048 bits. SVE provides 32 scalable vector registers, named Z0 – Z31.
- **Per-lane predication**
- SVE provides 16 predicate registers, named p0-p15, with each predicate register being 1/8th of the size of the vector register (1 bit per byte), and therefore scalable in size. Predicate registers are written to use condition-creating instructions, such as compares. Condition-creating instructions allow later instructions to control which elements (or 'lanes') that a vector should be operated on (the 'active' elements).
- **Gather-load and scatter-store**
- Gather-load and scatter-store allows data to be efficiently transferred to or from a vector of non-contiguous memory addresses. The efficient transfer of data enables a wider range of source code constructs to be vectorized. To permit efficient accesses to contiguous memory, SVE provides an extensive set of load and store instructions which progress sequentially forwards through an array, supporting a full range of packed 8, 16, 32, and 64-bit vector element organizations.

Neon ARM Vector Floating Point Processor

- SIMD architecture extension for the Arm Cortex-A and Cortex-R series processors.
- Neon technology is a packed SIMD architecture.
- Neon registers are considered as vectors of elements of the same data type, with Neon instructions operating on multiple elements simultaneously.
- Adds about 23 new instructions

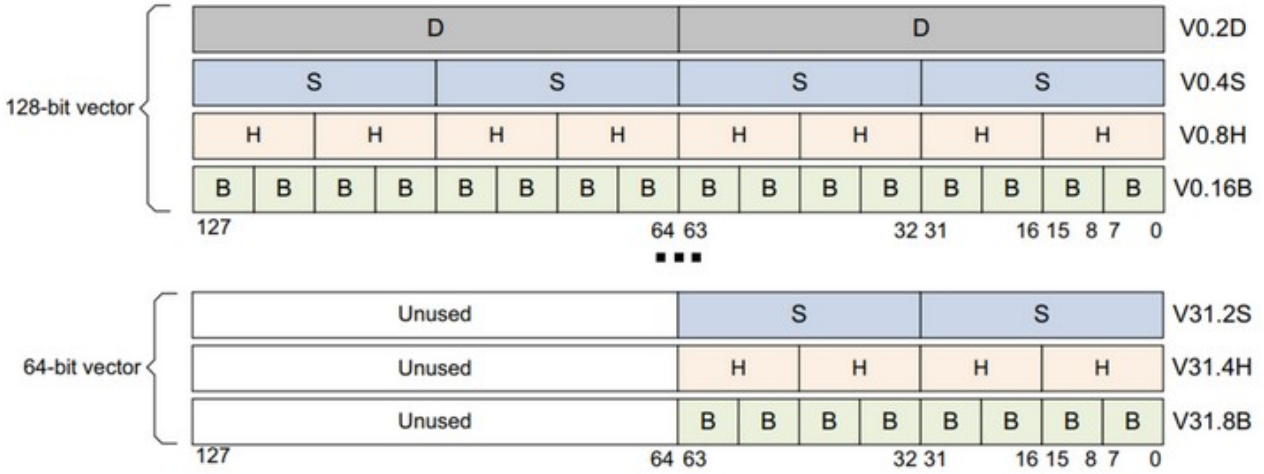


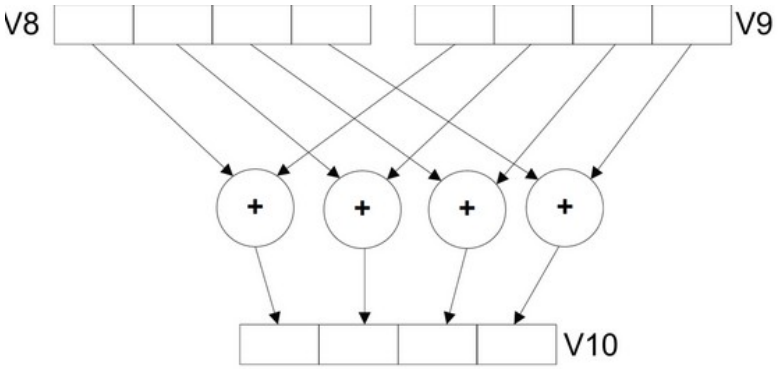
https://developer.arm.com/architectures/instruction-sets/simd-isas/neon

Packed SIMD Data Processing

- Seperate Vector Register File that features 128 bit data.
- Perform the same operation simultaneously for multiple data items. These data items are packed as separate lanes in a larger register.
- The diagram shows 128-bit registers each holding four 32-bit values, but other combinations are possible for Neon registers:

1. Two 64-bit, four 32-bit, eight 16-bit, or sixteen 8-bit integer data elements can be operated on simultaneously using all **128 bits** of a Neon register.
2. Two 32-bit, four 16-bit, or eight 8-bit integer data elements can be operated on simultaneously using the lower 64 bits of a Neon register (in this case, the upper 64 bits of the Neon register are unused).





ners-guide-for-armv8-a/introducing-neon-for-armv8-a/data-processing-methodologies

Classes of Instructions

1. Load /Store : vldr/vstr, vldm, vstm
2: Data Processing: vadd, vsub, vmul, vnmul, vdiv
3. Data Movement vmov,
4. Data type Conversion vcvt

Packed SIMD Drawbacks :
1. The length of all vectors is clamped
2. All operands should be of same type.

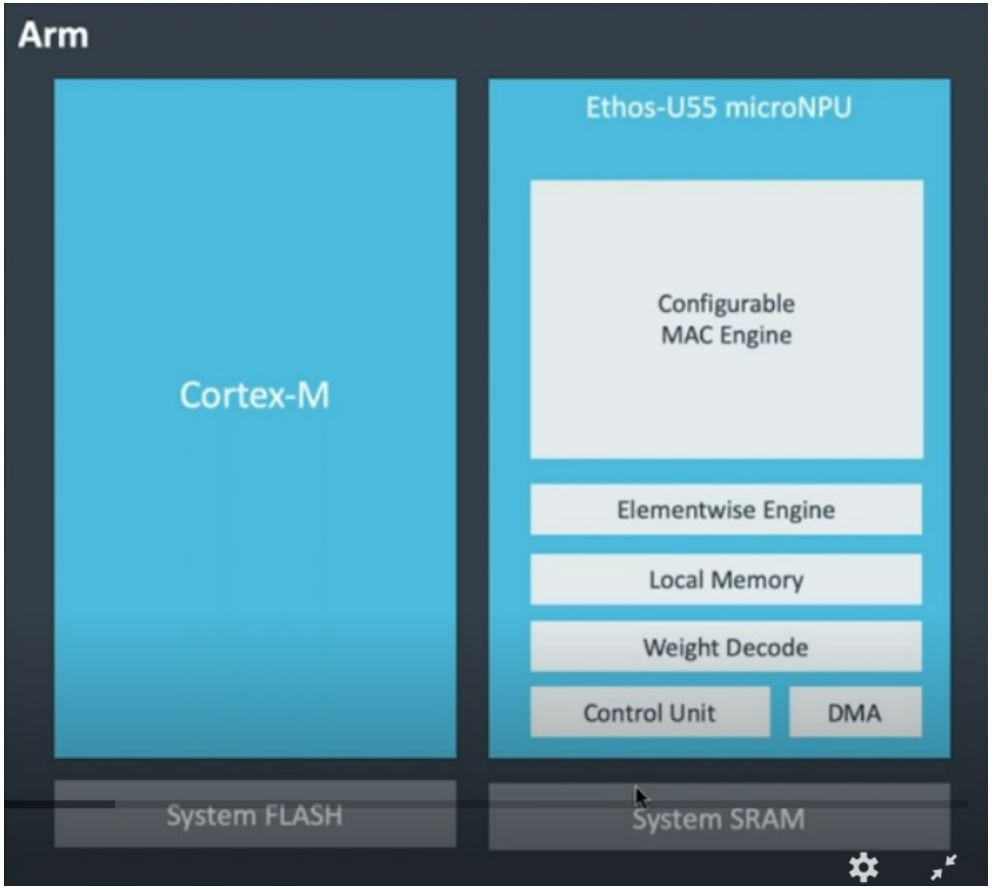
Helium ISA Extension

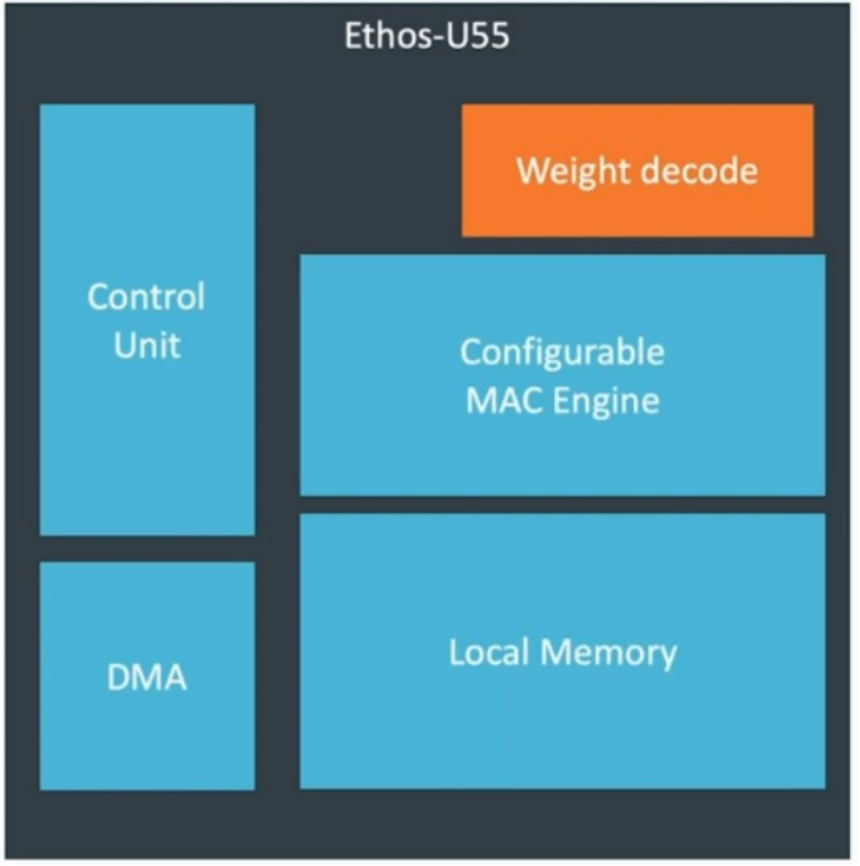
- Helium is a new ground-up design that enables efficient signal processing performance in small processors. It provides many new architectural features that are not available in Neon. The key differences between Helium and Neon are as follows:
- Helium is optimised for low power, high performing CPUs. As such, the architecture is designed to maximise usage of all available hardware.
- Helium has fewer vector registers than Neon, however, some Helium vector instructions can access the scalar register file and the vector register file simultaneously.
- Helium supports features like loop predication, lane predication, complex maths operations, and scatter-gather memory accesses.
- **Vector Processing Support**
- 2 x 32-bit MAC/cycle
- 4 x 16-bit MAC/cycle
- 8 x 8-bit MAC/cycle

ARM Ethos-U55 NPU for ML

- Coprocessor for ARM-Cortex M55
- Embedded ML Inference on Cortex-M uCs
- Designed to accelerate ML inference in area-constrained embedded and IoT devices.
- Uses: ARM Helium Vector ISA extension
- Helium is analogous to NEON for ARM-CortexM Series

ARM Ethos-U55 NPU for ML – introduced in Feb 2020

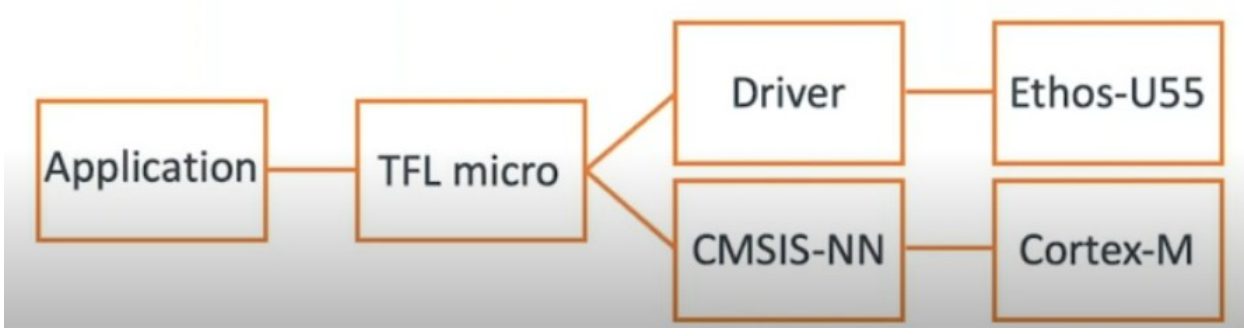




https://www.brighttalk.com/webcast/17792/391867/technical-overview-of-the-arm-cortex-m55-and-ethos-u55-processors

Features

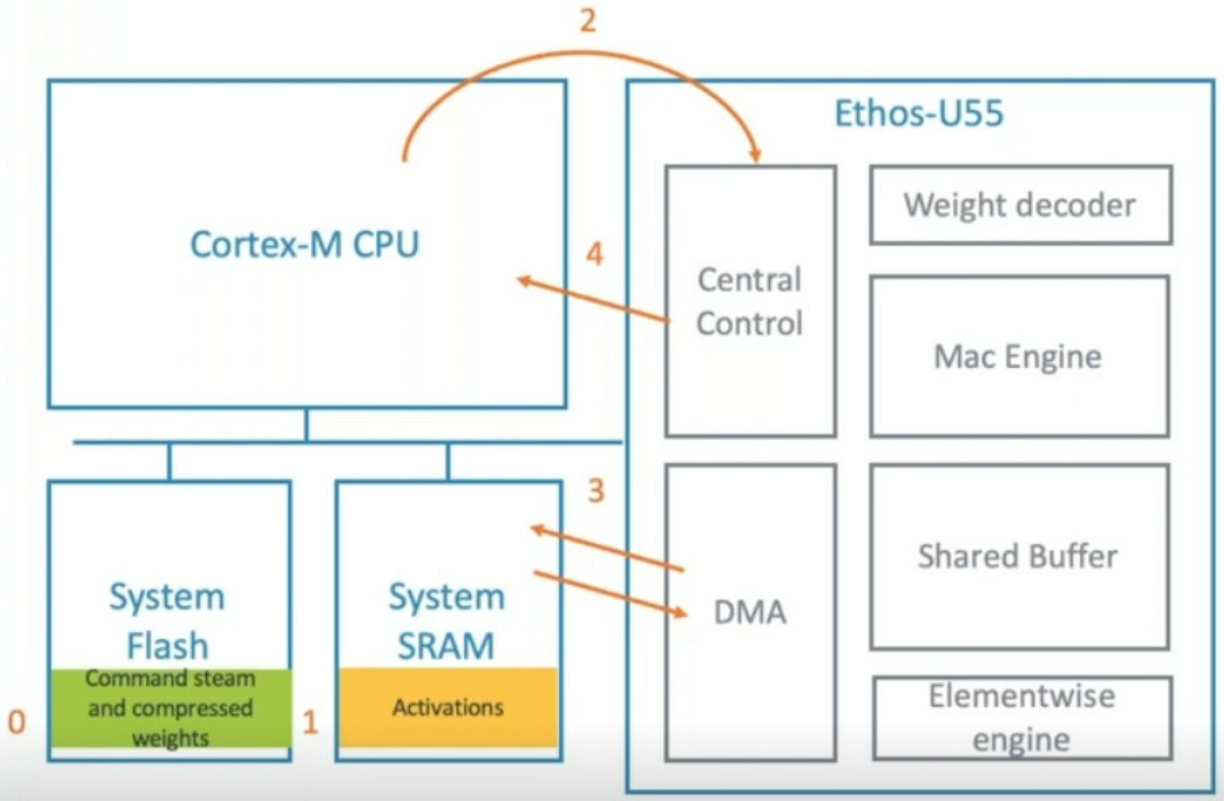
- > 150 new scalar and vector instructions
- Low overhead loops (due to predication)
- Weight Decoder and DMA for on the fly weight decompression (Weight Compression done offline)
- **Configurable MAC(32/64/128/256)**
- Ethos-U55 can completely execute networks that map to the supported operator set. Such as : *Deepspeech-V1 , RNNNoise, Wav2letter ,*
- Any unsupported operation falls back to Cortex-M55 for execution
- Softmax is not supported
-



Execution Flow

Typical Ethos-U55 Data Flow

0. An offline compiled command stream with corresponding compressed weights is put into system Flash.
1. Input activations are put into system SRAM.
2. The host starts Ethos-U55 by defining all memory regions to be used. In particular the location of the command stream and input activations.
3. Ethos-U55 autonomously runs all commands, using SRAM as a scratch buffer. Final results are written to a defined SRAM buffer.
4. Interrupt on completion of writing the final result.



https://www.brighttalk.com/webcast/17792/391867/technical-overview-of-the-arm-cortex-m55-and-ethos-u55-processors

Week-7 and 8

- Getting Started with ARM Assembly
- Disassembly analysis of Arm Neon Code using Raspberry Pi 3

Assemling on ARM

```
as -g main.s -o main.o
as -g mvmul.s -o mvmul.o
ld -o mvmul main.o mvmul.o
```

Fact Check

	Understanding	Source
1.	Raspberry Pi3 contains ARM Cortex A53	Rpi3 User manual
2.	Raspberry Pi3 contains ARM Neon CoProcessor	Cortex A53 Datasheet
3.	Command for Compiling NEON Assembly on Rpi is: gcc **-mfpu=neon-vfpv4** -o <exe> <src1.c> <src2.c> ...	1. ARM v8 Programmers Manual, 2. Raspberry Pi Assembly Language Programming by Stephen Smith (z-lib.org) 3. If Flag not given, “illegal instruction” compiler error pops up.

Programming with NEON

- SIMD Topology
- Can operate on 8,16,32 bit INT and Single Precision Float Numbers.
- It cannot perform 64-bit or 128-bit arithmetic.
- All elements getting operated should be of the same type.
- Greatest Parallelism is obtained using INT8.

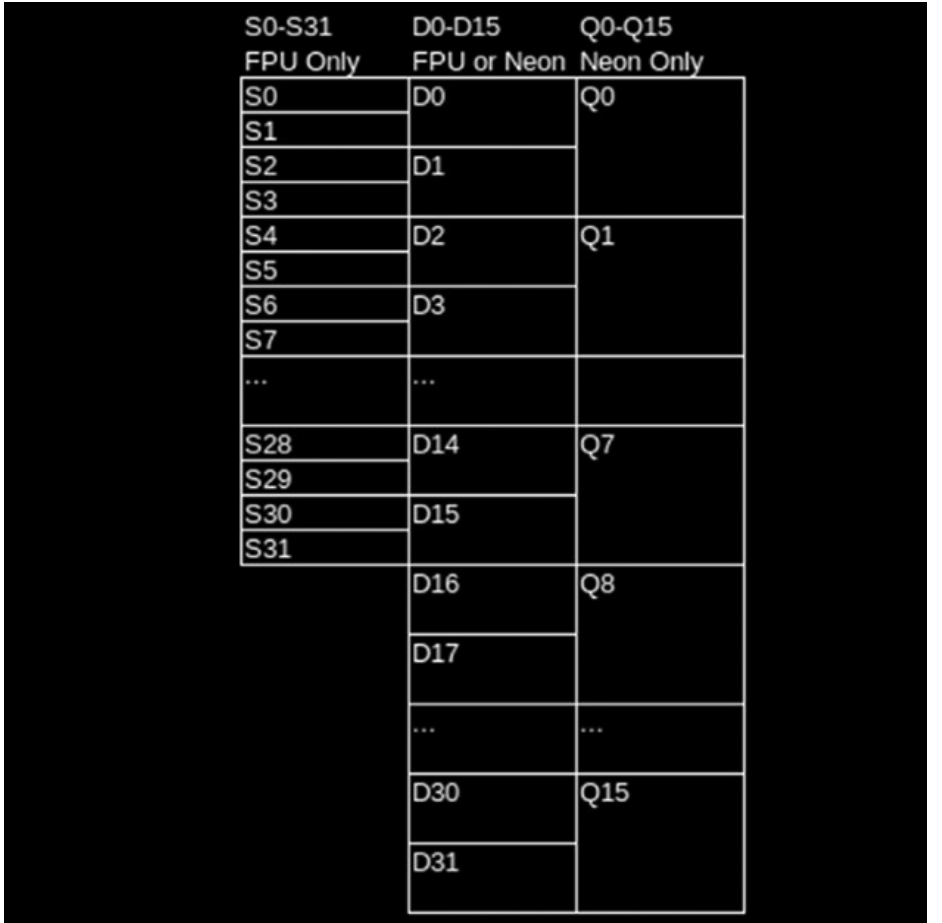


Figure 12-1. *The complete set of coprocessor registers for both the FPU and NEON coprocessors*

Lanes

- Depending on the data-type, the NEON divides the 128-bit register into equal lanes of size (128/ size_of_datatype)

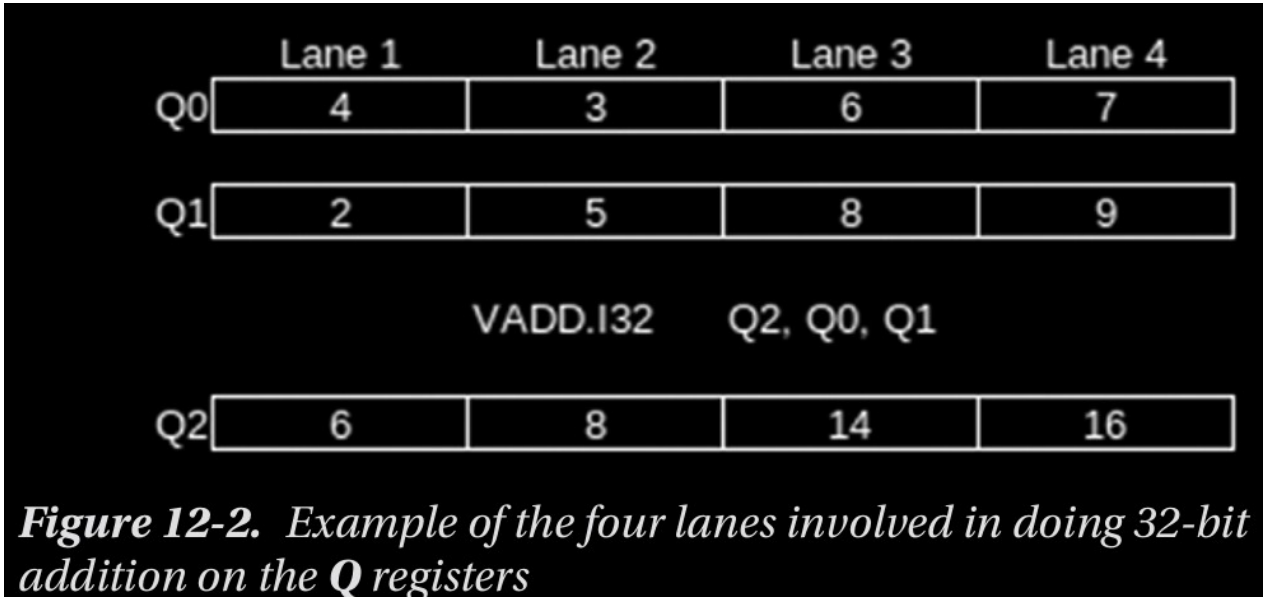


Figure 12-2. *Example of the four lanes involved in doing 32-bit addition on the* ***Q*** *registers*

Example: Distance in 4D

```
@
@ Function to Calculate the distance between 4D points
@ in single precision floating point point using
@ NEON Processor
@
@
.global distance
distance:
        VPUSH {S16-S31}
        PUSH {R4-R12,LR}
        VLDM R0, {Q2,Q3}
        vsub.f32 Q1, Q2, Q3
        vmul.f32 Q1, Q1, Q1
        vpadd.f32 D0, D2, D3
        vadd.f32 S0, S1
        vsqrt.f32 S4,S0
        vmov r0, s4
        pop {r4-r12,PC}
        vpop {s16-s31}
```

gcc **-mfpu=neon-vfpv4** -o distance distance.s main.s

Generates Executable on NEON. If the -mfpu is not specifed, illegal instruction error is obtained during compilation. Source : https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html

SIMD Coding for Matrix Vector Multiplication on NEON : Matrix 3x3 Vec 3x1

- Put a, d, and g in a register in separate lanes
- b, e, and h in another register in matching lanes
- c, f, and i in a third register in the matching lanes
- Multiply each of the lanes with x,y and z respectively and then add all three vector
- Thus, there are no loops
- All three coumns of the output are computed parallely

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \begin{vmatrix} x \\ y \\ z \end{vmatrix}$$

$$\begin{vmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{vmatrix}$$

	Lane 1	Lane 2	Lane 3
D1	a	d	g
D2	b	e	f
D3	c	f	i
D4	ax+by+cz	dx+ey+fz	gx+hy+iz

NEON Assembly vs NEON C Code

- C Code for Matrix Vector Multiplication compiled for NEON **does not use** the optimal SIMD discussed in the previous slide.
- Instead, it continues to loop three times for a 3x3 Matrix .
- Inference : *gcc for NEON does not auto-vectorize*

SIMD Assembly for NEON

- Hand-coded Assembly Program for Matrix Vector Multiplication using SIMD programming
- **Steps are:**
  - 1. Load 3x3 Matrix A into Q0, Q1 and Q2
  - 2. Load 3x1 Vector X into Q3.
  - 3. Multiply each of Q0, Q1, Q2 with Q3
  - 4. Perform Pair-wise addition of elements inside the products
  - 5. Store Results in Y (3 x 1)

```
main:
        push {R4-R12,LR}
@ Load Matrix into NEON Registers Q0, Q1, Q2
        LDR     R0,=A   @POinter to the Matrix
        VLDM    R0, {Q0-Q2}  @ Load A into Q0, Q1, Q2
@ Load Vector into NEON Register Q3
        LDR     R0,=X
        VLDM    R0,{Q3}
@ Product
        VMUL.f32 Q4, Q0, Q3
        VMUL.f32 Q5, Q1, Q3
        VMUL.f32 Q6, Q2, Q3
        VPADD.f32 D0, D8, D9
        VPADD.f32 D1, D10, D11
        VPADD.f32 D2, D12, D13
        VADD.f32 S0, S0, S1
        VADD.f32 S2, S2, S3
        VADD.f32 S4, S4, S5
        LDR     R1,=Y
        VSTM R1, {S0-S2}
        pop {R4-R12,PC}

.data
A:      .single 1.0, 4.0, 7.0, 0.0
        .single 2.0, 5.0, 8.0, 0.0
        .single 3.0, 6.0, 9.0, 0.0

X:      .single 1.0, 1.0, 1.0, 0.0
```

Disassembly of NEON Assembly Code for VMATMUL

```
000103d0 <main>:
   103d0:       e92d5ff0        push    {r4, r5, r6, r7, r8, r9, sl, fp, ip, lr}
   103d4:       e59f0038        ldr     r0, [pc, #56]   ; 10414 <main+0x44>
   103d8:       ec900b0c        vldmia  r0, {d0-d5}
   103dc:       e59f0034        ldr     r0, [pc, #52]   ; 10418 <main+0x48>
   103e0:       ec906b04        vldmia  r0, {d6-d7}
   103e4:       f3008d56        vmul.f32        q4, q0, q3
   103e8:       f302ad56        vmul.f32        q5, q1, q3
   103ec:       f304cd56        vmul.f32        q6, q2, q3
   103f0:       f3080d09        vpadd.f32       d0, d8, d9
   103f4:       f30a1d0b        vpadd.f32       d1, d10, d11
   103f8:       f30c2d0d        vpadd.f32       d2, d12, d13
   103fc:       ee300a20        vadd.f32        s0, s0, s1
   10400:       ee311a21        vadd.f32        s2, s2, s3
   10404:       ee322a22        vadd.f32        s4, s4, s5
   10408:       e59f100c        ldr     r1, [pc, #12]   ; 1041c <main+0x4c>
   1040c:       ec810a03        vstmia  r1, {s0-s2}
   10410:       e8bd9ff0        pop     {r4, r5, r6, r7, r8, r9, sl, fp, ip, pc}
   10414:       00021024        .word   0x00021024
   10418:       00021054        .word   0x00021054
   1041c:       00021064        .word   0x00021064
```

VMATMUL for C Code targetting NEON

```
00010524 <vmatmul>:
   10524:       e3530000        cmp     r3, #0
   10528:       d12fff1e        bxle    lr
   1052c:       e92d4070        push    {r4, r5, r6, lr}
   10530:       e1a05103        lsl     r5, r3, #2
   10534:       e59d4010        ldr     r4, [sp, #16]
   10538:       e080e102        add     lr, r0, r2, lsl #2
   1053c:       e0846005        add     r6, r4, r5
   10540:       e3520000        cmp     r2, #0
   10544:       eddf7a0c        vldr    s15, [pc, #48]  ; 1057c <vmatmul+0x58>
   10548:       c1a0c001        movgt   ip, r1
   1054c:       c1a03000        movgt   r3, r0
   10550:       da000004        ble     10568 <vmatmul+0x44>
   10554:       ecf36a01        vldmia  r3!, {s13}
   10558:       ecbc7a01        vldmia  ip!, {s14}
   1055c:       e15e0003        cmp     lr, r3
   10560:       eee67a87        vfma.f32        s15, s13, s14
   10564:       1affffa         bne     10554 <vmatmul+0x30>
   10568:       ece47a01        vstmia  r4!, {s15}
   1056c:       e0811005        add     r1, r1, r5
   10570:       e1560004        cmp     r6, r4
   10574:       1afffff1        bne     10540 <vmatmul+0x1c>
   10578:       e8bd8070        pop     {r4, r5, r6, pc}
   1057c:       00000000        .word   0x00000000
```

Observations

1. Even with -O3 optiization, the code does not produce SIMD Code for Matrix Vector Multiplication of size 3x3.

2. NEON SIMD is not scalable for Matrices beyond 4 rows as the intermediate results have to be stored onto Stack and then retrived.

Desirable Instructions

1. Dot product of two Vectors : Exists in NEON but is only supported for INT8 numbers

2. Add all elements in a vector register (currently, only Pair Add is available)