ARM NEON programming quick reference

ARM NEON programming quick reference

1 Introduction

This article aims to introduce ARM NEON technology. Hope that beginners can get started with NEON programming quickly after reading the article. The article will also inform users which documents can be consulted if more detailed information is needed.

2 NEON overview

This section describes the NEON technology and supplies some background knowledge

2.1 What is NEON?

NEON technology is an advanced SIMD (Single Instruction, Multiple Data) architecture for the ARM Cortex-A series processors. It can accelerate multimedia and signal processing algorithms such as video encoder/decoder, 2D/3D graphics, gaming, audio and speech processing, image processing, telephony, and sound.

NEON instructions perform "Packed SIMD" processing:

- Registers are considered as vectors of elements of the same data type
- Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single-precision floating-point on ARM 32-bit platform, both single-precision floating-point and double-precision floating-point on ARM 64-bit platform.
- Instructions perform the same operation in all lanes

2.2 History of ARM Adv SIMD

ARMv6[i] SIMD extension	ARMv7-A NEON	ARMv8-A AArch64 NEON
▪ Operates on 32-bit general purpose ARM registers ▪ 8-bit/16-bit integer ▪ 2x16-bit/4x8-bit operations per instruction	▪ Separate register bank, 32x64-bit NEON registers ▪ 8/16/32/64-bit integer ▪ Single precision floating point ▪ Up to 16x8-bit operations per instruction	▪ Separate register bank, 32x128-bit NEON registers ▪ 8/16/32/64-bit integer ▪ Single precision floating point ▪ double precision floating point, both of them are IEEE compliance ▪ Up to 16x8-bit operations per instruction

Feedback

[i] The ARM Architecture Version 6 (ARMv6) David Brash: page 13

2.3 Why use NEON

NEON provides:

- Support for both integer and floating point operations ensures the adaptability of a broad range of applications, from codecs to High Performance Computing to 3D graphics.
- Tight coupling to the ARM processor provides a single instruction stream and a unified view of memory, presenting a single development platform target with a simpler tool flow[ii]

3 ARMv7/v8 comparison

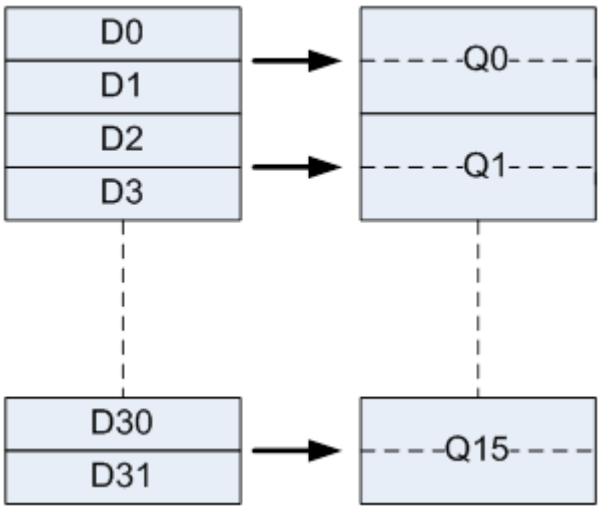
ARMv8-A is a fundamental change to the ARM architecture. It supports the 64-bit Execution state called “AArch64”, and a new 64-bit instruction set “A64”. To provide compatibility with the ARMv7-A (32-bit architecture) instruction set, a 32-bit variant of ARMv8-A “AArch32” is provided. Most of existing ARMv7-A code can be run in the AArch32 execution state of ARMv8-A.

This section compares the NEON-related features of both the ARMv7-A and ARMv8-A architectures. In addition, general purpose ARM registers and ARM instructions, which are used often for NEON programming, will also be mentioned. However, the focus is still on the NEON technology.

3.1 Register

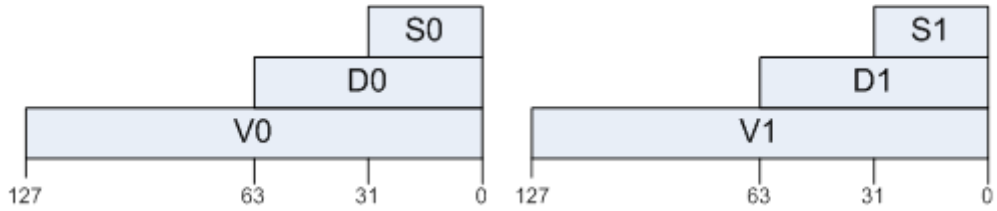
ARMv7-A and AArch32 have the same general purpose ARM registers – 16 x 32-bit general purpose ARM registers (R0-R15).

ARMv7-A and AArch32 have 32 x 64-bit NEON registers (D0-D31). These registers can also be viewed as 16x128-bit registers (Q0-Q15). Each of the Q0-Q15 registers maps to pair of D registers, as shown in the following figure.



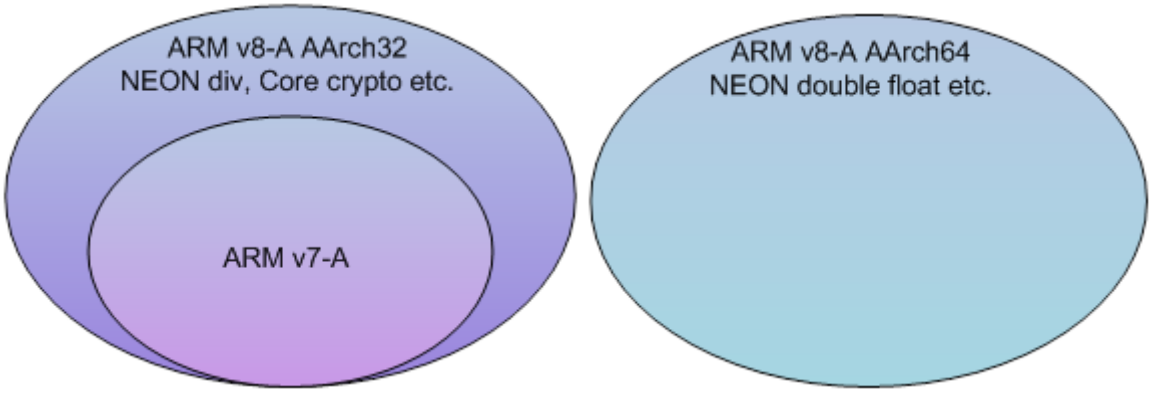
AArch64 by comparison, has 31 x 64-bit general purpose ARM registers and 1 special register having different names, depending on the context in which it is used. These registers can be viewed as either 31 x 64-bit registers (X0-X30) or as 31 x 32-bit registers (W0-W30).

AArch64 has 32 x 128-bit NEON registers (V0-V31). These registers can also be viewed as 32-bit Sn registers or 64-bit Dn registers.



3.2 Instruction set[iii]

The following figure illustrates the relationship between ARMv7-A, ARMv8-A AArch32 and ARMv8-A AArch64 instruction set.



Feedback

The ARMv8-A AArch32 instruction set consists of A32 (ARM instruction set, a 32-bit fixed length instruction set) and T32 (Thumb instruction set, a 16-bit fixed length instruction set; Thumb2 instruction set, 16 or 32-bit length instruction set). It is a superset of the ARMv7-A instruction set, so that it retains the backwards compatibility necessary to run existing software. There are some additions to A32 and T32 to maintain alignment with the A64 instruction set, including NEON division, and the Cryptographic Extension instructions. NEON double precision floating point (IEEE compliance) is also supported.

3.3 NEON instruction format

This section describes the changes to the NEON instruction syntax.

3.3.1 ARMv7-A/AArch32 instruction syntax[iv]

All mnemonics for ARMv7-A/AAArch32 NEON instructions (as with VFP) begin with the letter “V”. Instructions are generally able to operate on different data types, with this being specified in the instruction encoding. The size is indicated with a suffix to the instruction. The number of elements is indicated by the specified register size and data type of operation. Instructions have the following general format:

V{<mod>}<op>{<shape>}{<cond>}{.<dt>}{<dest>}, src1, src2

Where:

<mod> - modifiers

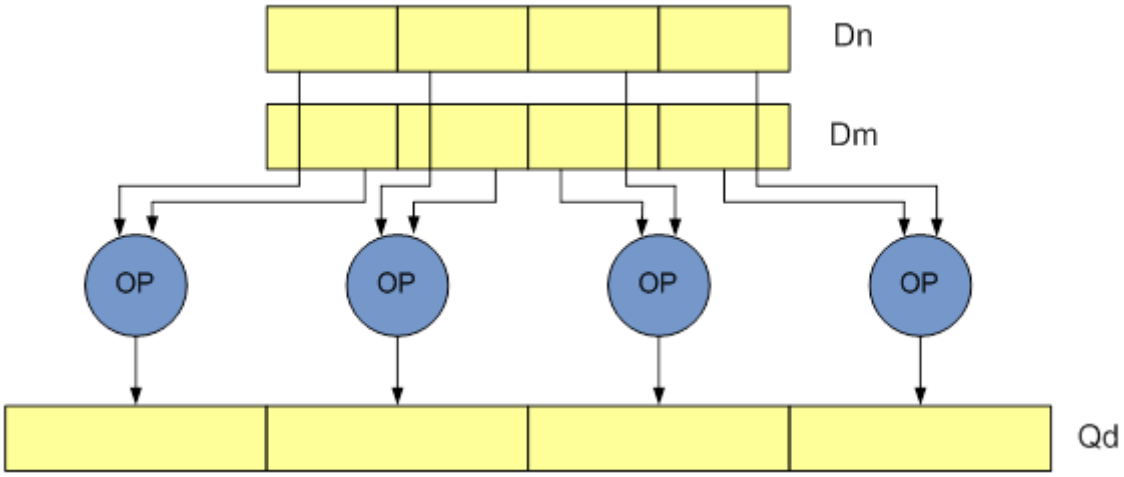
- Q: The instruction uses saturating arithmetic, so that the result is saturated within the range of the specified data type, such as VQABS, VQSHL etc.
- H: The instruction will halve the result. It does this by shifting right by one place (effectively a divide by two with truncation), such as VHADD, VHSUB.
- D: The instruction doubles the result, such as VQDMULL, VQDMLAL, VQDMLSL and VQ{R}DMULH
- R: The instruction will perform rounding on the result, equivalent to adding 0.5 t the result before truncating, such as VRHADD, VRSHR.

<op> - the operation (for example, ADD, SUB, MUL).

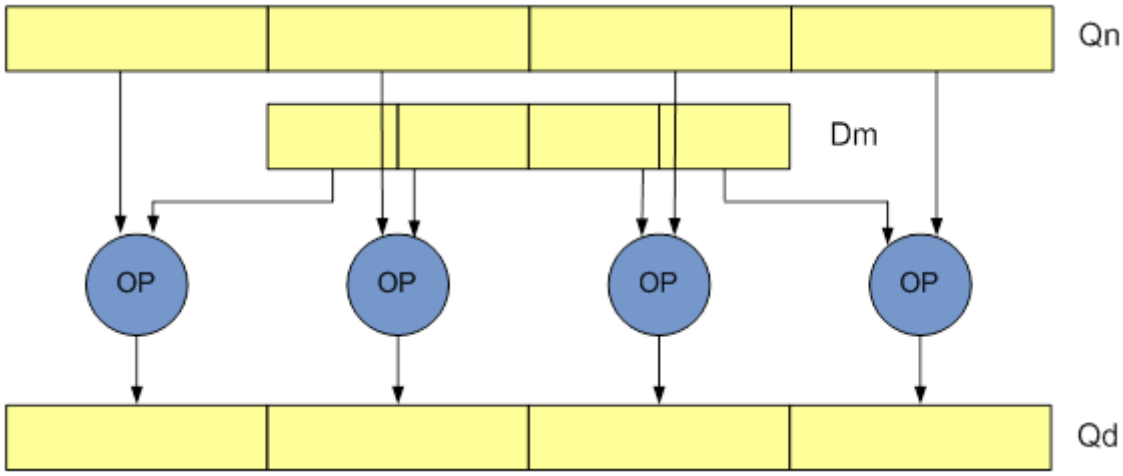
<shape> - Shape.

Neon data processing instructions are typically available in Normal, Long, Wide and Narrow variants.

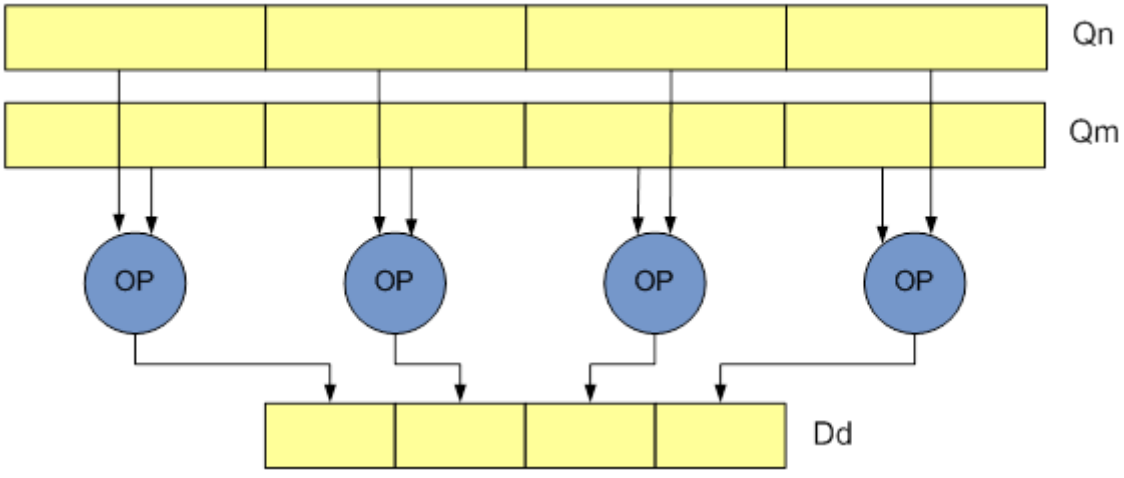
- Long (L): instructions operate on double-word vector operands and produce a quad-word vector result. The result elements are twice the width of the operands, and of the same type. Lengthening instructions are specified using an L appended to the instruction.



- Wide (W): instructions operate on a double-word vector operand and a quad-word vector operand, producing a quad-word vector result. The result elements and the first operand are twice the width of the second operand elements. Widening instructions have a W appended to the instruction.



- Narrow (N): instructions operate on quad-word vector operands, and produce a double-word vector result. The result elements are half the width of the operand elements. Narrowing instructions are specified using an N appended to the instruction.



<cond> - Condition, used with IT instruction

<.dt> - Data type, such as s8, u8, f32 etc.

<dest> - Destination

<src1> - Source operand 1

<src2> - Source operand 2

Note: {} represents and optional parameter.

For example:

VADD.I8 D0, D1, D2

VMULL.S16 Q2, D8, D9

For more information, please refer to the documents listed in the Appendix.

3.3.2 AArch64 NEON instruction syntax[y]

In the AArch64 execution state, the syntax of NEON instruction has changed. It can be described as follows:

{<prefix>}<op>{<suffix>} Vd.<T>, Vn.<T>, Vm.<T>

Where:

<prefix> - prefix, such as using S/U/F/P to represent signed/unsigned/float/bool data type.

<op> – operation, such as ADD, AND etc.

<suffix> - suffix

- P: “pairwise” operations, such as ADDP.
- V: the new reduction (across-all-lanes) operations, such as FMAXV.
- 2： new widening/narrowing “second part” instructions, such as ADDHN2, SADDL2.

ADDHN2: add two 128-bit vectors and produce a 64-bit vector result which is stored as high 64-bit part of NEON register.

SADDL2: add two high 64-bit vectors of NEON register and produce a 128-bit vector result.

<T> - data type, 8B/16B/4H/8H/2S/4S/2D. B represents byte (8-bit). H represents half-word (16-bit). S represents word (32-bit). D represents a double-word (64-bit).

For example:

UADDLP V0.8H, V0.16B

FADD V0.4S, V0.4S, V0.4S

For more information, please refer to the documents listed in the Appendix.

3.4 NEON instructions[vi]

The following table compares the ARMv7-A, AArch32 and AArch64 NEON instruction set.

“√” indicates that the AArch32 NEON instruction has the same format as ARMv7-A NEON instruction.

“Y” indicates that the AArch64 NEON instruction has the same functionality as ARMv7-A NEON instructions, but the format is different. Please check the ARMv8-A ISA document.

If you are familiar with the ARMv7-A NEON instructions, there is a simple way to map the NEON instructions of ARMv7-A and AArch64. It is to check the NEON intrinsics document, so that you can find the AArch64 NEON instruction according to the intrinsics instruction.

New or changed functionality is highlighted.

	ARMv7-A	AArch32	AArch64
logical and compare	VAND, VBIC, VEOR, VORN, and VORR (register)	√	Y
	VBIC and VORR (immediate)	√	Y
	VBIF, VBIT, and VBSL	√	Y
	VMOV, VMVN (register)	√	Y
	VACGE and VACGT	√	Y
	VCEQ, VCGE, VCGT, VCLE, and VCLT	√	Y
	VTST	√	Y
general data processing	VCVT (between fixed-point or integer, and floating-point)	√	Y
	VCVT (between half-precision and single-precision floating-point)	√	Y

	n/a	n/a	FCVTXN(double to single-precision)
	VDUP	√	Y
	VEXT	√	Y
	VMOV, VMVN (immediate)	√	Y
	VMOVL, V{Q}MOVN, VQMOVUN	√	Y
	VREV	√	Y
	VSWP	√	n/a
	VTBL, VTBX	√	Y
	VTRN	√	TRN1, TRN2
	VUZP, VZIP	√	UZP1,UZP2, ZIP, ZIP2
	n/a	n/a	INS
	n/a	VRINTA, VRINM, VRINTN, VRINTP, VRINTR, VRINTX,	FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ

		VRINTZ	
shift	VSHL, VQSHL, VQSHLU, and VSHLL (by immediate)	√	Y
	V{Q}{R}SHL (by signed variable)	√	Y
	V{R}SHR	√	Y
	V{R}SHRN	√	Y
	V{R}SRA	√	Y
	VQ{R}SHR{U}N	√	Y
	VSLI and VSRI	√	Y
general arithmetic	VABA{L} and VABD{L}	√	Y
	V{Q}ABS and V{Q}NEG	√	Y
	V{Q}ADD, VADDL, VADDW, V{Q}SUB, VSUBL, and VSUBW	√	Y
	n/a	n/a	SUQADD, USQADD
	V{R}ADDHN and V{R}SUBHN	√	Y
	V{R}HADD and VHSUB	√	Y
	VPADD{L}, VPADAL	√	Y
	VMAX, VMIN, VPMAX,	√	Y

Feedback

	and VPMIN		
	n/a	n/a	FMAXNMP, FMINNMP
	VCLS, VCLZ, and VCNT	√	Y
	VRECPE and VRSQRTE	√	Y
	VRECPS and VRSQRTS	√	Y
	n/a	n/a	FRECPX
			RBIT
			FSQRT
			ADDV
			SADDLV, UADDLV
			SMAXV,UMAXV,FMAXV
			FMAXNMV
			SMINV,UMINV,FMINV
			FMINNMV
multiply	VMUL{L}, VMLA{L}, and VMLS{L}	√	There isn't float MLA/MLS
	VMUL{L}, VMLA{L}, and VMLS{L} (by scalar)	√	Y

	VFMA, VFMS	√	Y
	VQDMULL, VQDMLAL, and VQDMLSL (by vector or by scalar)	√	Y
	VQ{R}DMULH (by vector or by scalar)	√	Y
	n/a	n/a	FMULX
	n/a	n/a	FDIV
load and store	VLDn/VSTn(n=1, 2, 3, 4)	√	Y
	VPUSH/VPOP	√	n/a
Crypto Extension	n/a	PMULL, PMULL2	PMULL, PMULL2
		AESD, AESE	AESD, AESE
		AESIMC, AESMC	AESIMC, AESMC
		SHA1C, SHA1H, SHA1M, SHA1P	SHA1C, SHA1H, SHA1M, SHA1P
		SHA1SU0, SHA1SU1	SHA1SU0, SHA1SU1
		SHA256H, SHA256H2	SHA256H, SHA256H2

		SHA256SU0, SHA256SU1	SHA256SU0, SHA256SU1

4 NEON programming basics

There are four ways of using NEON[vii]:

- NEON optimized libraries
- Vectorizing compilers
- NEON intrinsics
- NEON assembly

4.1 Libraries

The users can call the NEON optimized libraries directly in their program. Currently, you can use the following libraries:

- OpenMax DL

This provides the recommended approach for accelerating AV codecs and supports signal processing and color space conversions.

- Ne10

It is ARM's open source project. Currently, the Ne10 library provides some math, image processing and FFT function. The FFT implementation is faster than other open source FFT implementations.

4.2 Vectorizing compilers

Adding vectorizing options in GCC can help C code to generate NEON code. GNU GCC gives you a wide range of options that aim to increase the speed, or reduce the size of

the executable files they generate. For each line in your source code there are generally many possible choices of assembly instructions that could be used. The compiler must trade-off a number of resources, such as registers, stack and heap space, code size (number of instructions), compilation time, ease of debug, and number of cycles per instruction in order to produce an optimized image file.

4.3 NEON intrinsics

NEON intrinsics provides a C function call interface to NEON operations, and the compiler will automatically generate relevant NEON instructions allowing you to program once and run on either an ARMv7-A or ARMv8-A platform. If you intend to use the AArch64 specific NEON instructions, you can use the (__aarch64__) macro definition to separate these codes, as in the following example.

NEON intrinsics example:

```
//add for float array. assumed that count is multiple of 4

#include<arm_neon.h>

void add_float_c(float* dst, float* src1, float* src2, int count)

{

    int i;

    for (i = 0; i < count; i++)

        dst[i] = src1[i] + src2[i];

}

void add_float_neon1(float* dst, float* src1, float* src2, int count)
```

```
{
    int i;
    for (i = 0; i < count; i += 4)
    {
        float32x4_t in1, in2, out;
        in1 = vld1q_f32(src1);
        src1 += 4;
        in2 = vld1q_f32(src2);
        src2 += 4;
        out = vaddq_f32(in1, in2);
        vst1q_f32(dst, out);
        dst += 4;
        // The following is only an example describing how to use AArch64 specific NEON
        // instructions.
#if defined (__aarch64__)
        float32_t tmp = vaddvq_f32(in1);
#endif
    }
```

```
}
```

Checking disassembly, you can find vld1/vadd/vst1 NEON instruction on ARMv7-A platform and ldr/fadd/str NEON instruction on ARMv8-A platform.

4.4 NEON assembly

There are two ways to write NEON assembly:

- Assembly files
- Inline assembly

4.4.1 Assembly files

You can use ".S" or ".s" as the file suffix. The only difference is that C/C ++ preprocessc will process .S files first. C language features such as macro definitions can be used.

When writing NEON assembly in a separate file, you need to pay attention to saving the registers. For both ARMv7 and ARMv8, the following registers must be saved:

	ARMv7-A/AArch32	AArch64[viii]
General purpose registers	R0-R3 parameters	X0-X7 parameters
	R4-R11 need to be saved	X8-X18
	R12 IP	X19-X28 need to be saved
	R13(SP)	X29(FP) need to be saved
	R14(LR) need to be saved	X30(LR)
	R0 for return value	X0, X1 for return value

NEON registers	D8-D15 need to be saved	D part of V8-V15 need to be saved
Stack alignment	64-bit alignment	128-bit alignment[ix]
Stack push/pop	PUSH/POP Rn list VPUSH/VPOP Dn list	LDP/STP register pair

The following is an example of ARM v7-A and ARM v8-A NEON assembly.

```
//header
void add_float_neon2(float* dst, float* src1, float* src2, int count);
```

```
//assembly code in .S file
```

ARMv7-A/AArch32

```
    .text
    .syntax unified
    .align 4
    .global add_float_neon2
    .type add_float_neon2, %function
    .thumb
.thumb_func
add_float_neon2:
```

AArch64

```
    .text
    .align 4
    .global add_float_neon2
    .type add_float_neon2, %function
add_float_neon2:
.L_loop:
    ld1     {v0.4s}, [x1], #16
    ld1     {v1.4s}, [x2], #16
```

```
.L_loop:
    vld1.32  {q0}, [r1]!
    vld1.32  {q1}, [r2]!
    vadd.f32 q0, q0, q1
    subs r3, r3, #4
    vst1.32  {q0}, [r0]!
    bgt .L_loop
    bx lr
```

```
    fadd    v0.4s, v0.4s, v1.4s
    subs x3, x3, #4
    st1  {v0.4s}, [x0], #16
    bgt .L_loop
    ret
```


For more examples, see:
https://github.com/projectNe10/Ne10/tree/master/modules/dsp

4.4.2 Inline assembly

You can use NEON inline assembly directly in C/C++ code.

Pros:

- The procedure call standard is simple. You do not need to save registers manually.
- You can use C / C ++ variables and functions, so it can be easily integrated into C / C ++ code.

Cons:

- Inline assembly has a complex syntax.
- NEON assembly code is embedded in C/C ++ code, and it's not easily ported to other platforms.

Example:

```
// ARMv7-A/AArch32
void add_float_neon3(float* dst, float* src1, float* src2, int cour
{
 asm volatile (
 "1: \n"
 "vld1.32 {q0}, [%[src1]]! \n"
 "vld1.32 {q1}, [%[src2]]! \n"
 "vadd.f32 q0, q0, q1 \n"
 "subs %[count], %[count], #4 \n"
 "vst1.32 {q0}, [%[dst]]! \n"
 "bgt 1b \n"
 : [dst] "+r" (dst)
 : [src1] "r" (src1), [src2] "r" (src2), [count] "r" (count)
 : "memory", "q0", "q1"
 );
}
```



```
// AArch64
void add_float_neon3(float* dst, float* src1, float* src2, ir
{
 asm volatile (
 "1: \n"
 "ld1 {v0.4s}, [%[src1]], #16 \n"
 "ld1 {v1.4s}, [%[src2]], #16 \n"
 "fadd v0.4s, v0.4s, v1.4s \n"
 "subs %[count], %[count], #4 \n"
 "st1 {v0.4s}, [%[dst]], #16 \n"
 "bgt 1b \n"
 : [dst] "+r" (dst)
 : [src1] "r" (src1), [src2] "r" (src2), [count] "r" (count)
 : "memory", "v0", "v1"
 );
}
```

4.5 NEON intrinsics and assembly

NEON intrinsics and assembly are the commonly used NEON. The following table describes the pros and cons of these two approaches:

	NEON assembly	NEON intrinsic
Performance	Always shows the best performance for the specified platform for an experienced developer.	Depends heavily on the toolchain used
Portability	The different ISAs (ARMv7-A/AArch32 and AArch64) have different assembly implementations. Even for the same ISA, the assembly might need to be fine-tuned to achieve ideal performance between different micro architectures.	Program once and run on different ISA's. The compiler may also grant performance fine-tuning for different micro-architectures.
Maintainability	Hard to read/write compared to C.	Similar to C code, it's easy to read/write.

This is a simple summary. When applying NEON to more complex scenarios, there will be many special cases. This will be described in a future article *ARM NEON Optimization*.

With the above information, you can choose a NEON implementation and start your NEON programming journey.

For more reference documentation, please check the appendix.

Appendix: NEON reference document

---

[i] The ARM Architecture Version 6 (ARMv6) David Brash: page 13

[ii] ARM Cortex-A Series Programmer's Guide Version 4.0: page 7-5

[iii] http://www.arm.com/zh/products/processors/instruction-set-architectures/armv8-architecture.php

[iv] ARM® Compiler toolchain Version 5.02 Assembler Reference: Chapter 4

NEON and VFP Programming

ARM Cortex -A Series Version: 4.0 Programmer's Guide: 7.2.4 NEON instruction set

[v] ARMv8 Instruction Set Overview: 5.8 Advanced SIMD

[vi] ARMv8 Instruction Set Overview: 5.8.25 AArch32 Equivalent Advanced SIMD Mnemonics

[vii] http://www.arm.com/zh/products/processors/technologies/neon.php

[viii]Procedure Call Standard for the ARM 64-bit Architecture (AArch64) : 5 THE BASE PROCEDURE CALL STANDARD

[ix] Procedure Call Standard for the ARM 64-bit Architecture (AArch64) : 5.2.2 The Stack

Jerome Decamps - 杜尚杰 3 年前

Really great job ! Thanks for all that it clearly explain

Michael Thomas 3 年前

See also the NEON Programmers guide on the Infocenter (requires click-through license to download).

NEON Programmer's Guide

Jens Bauer 3 年前

This is absolutely an excellent document. Thank you for taking the time to write it!

vfar 3 年前

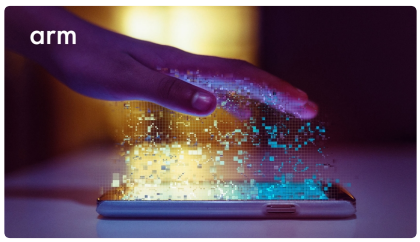
太好了　支持楼上

Zia Chang 3 年前

Very good introduction!

加载后面的

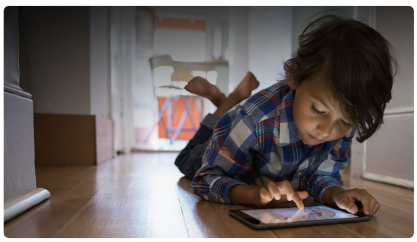
Android blog



Arm support for Android NNAPI gives >4x performance boost

Arm now supports Android Neural Networks API (NNAPI) with open-source, optimized NN operators that deliver serious performance uplift across CPUs and GPUs

Robert Elliott



The 64-bit Future of Android

Arm is enthusiastic about Google's announcement earlier this week encouraging Android applications to take advantage of the 64-bit Arm Architecture

DaveW

Example of calling Java methods through the JNI, in ARM Assembly, on Android

This document demonstrates how to call the JNI, through a procedure :written with the GNU ARM Assembly syntax,assembled and stored in a ELF shared library,executed by an Android system through an Android…

Myy

Feedback