ARM NEON Assembly Optimization

Dae-Hwan Kim
Department of Computer and Information,
Suwon Science College, 288 Seja-ro, Jeongnam-myun,
Hwaseong-si, Gyeonggi-do, Rep. of Korea
kimdh@ssc.ac.kr

***Abstract*—ARM is one of the most widely used 32-bit processors, which is embedded in smartphones, tablets, and various electronic devices. The ARM NEON is the SIMD engine inside ARM core which accelerates multimedia and signal processing algorithms. NEON is widely incorporated in the recent ARM processors for smartphones and tablets. In this paper, various assembly level software optimizations are provided such as instruction scheduling, instruction selection, and loop unrolling for the NEON architecture. The proposed techniques are expected to be applied directly in the software development for the ARM NEON processors.**

Keywords—ARM; NEON; embedded processor; software optimization; assembly

I. INTRODUCTION

ARM is one of the most widely used 32-bit processors, which is embedded in smartphones, tablets, automobiles, and various electronic devices. In 2015, about 15 billion ARM-based chips were shipped, and ARM maintains a more than 95% market share of mobile phones and tablets [2].

The first successful commercial version is ARMv4 [13], and numerous approaches have been proposed to enhance the ARM architecture to meet the market demands. In 2000, ARMv5EJ is introduced where the ARM DSP instruction set extension [10] and Jazelle java execution accelerator [6], which enables the architecture to execute java bytecode by hardware. It improves performance by eight times compared to the software implemented java virtual machine, and reduces power consumption by 80%. It also adds new DSP instructions to speed up signal processing applications. This extension is incorporated in various processors such as ARM926EJ-S and ARM1026EJ-S.

In 2002, ARMv6 is introduced which adopts the SIMD (Single Instruction Multiple Data) instructions which can operate simultaneously on two 16-bit or four 8-bit data packed in a 32-bit register. This extension achieves 75% performance improvement for multimedia applications, and is implemented in the ARM11 processors.

The ARMv7 architecture supports the advanced SIMD extension called NEON [5, 7, 8]. The new instructions can handle data stored in the 64-bit doubleword and 128-bit quadword registers. It accelerates multimedia, signal processing applications, graphics, and image processing algorithms. The architecture is implemented in the recent ARM Cortex-A processors such as Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15, and Cortex-A53.

The ARMv8 architecture [3, 4] introduces a 64-bit architecture, named AArch64, and a new A64 instruction set to the existing instruction set to support the 64-bit operation and the virtual addressing.

In this paper, various assembly software optimization techniques are proposed for the ARM NEON architecture. Practical example code is given to the optimization technique, whose performance is analyzed, compared to the original code.

The rest of this paper is organized as follows. Section II shows the overview of the assembly optimizations, and Section III presents each technique in detail with an example. Conclusions are presented in Section IV.

II. ASSEMBLY OPTIMIZATION OVERVIEW

NEON is the SIMD (Single Instruction Multiple Data) accelerator in the ARM core, which can handle 16 data simultaneously in a single instruction. NEON has separate register set, which can be used various configurations such as 32 64-bit (Dx register) or 16 128-bit register (Qx register). For example, instruction 'VADD.I16 D2, D1, D0' performs four 16-bit additions in parallel and stores the result into D2 where D0, D1, and D2 are 64-bit registers, respectively.

The AP (Application processor) main vendors such as Qualcomm, Samsung Electronics, Apple and NVidia have licensed ARM core from the ARM Limited. Table I shows the recent smartphone processors and their ARM cores. Cortex-A9 is used in NVidia's Tegra3. Cortex-A15 and Cortex-A7 are adopted in Samsung's Exynos 5. Qualcomm's Krait is replacing Qualcomm's Scorpion, which is based on the ARMv7 architecture. Apple's Swift adopts the ARMv7s architecture, which is the enhanced version of ARMv7 by Apple. Most recent smartphones have ARM SIMD (Single Instruction Multiple Data) accelerators named NEON, and thus, it is necessary to optimize the NEON code which is widely used in the multimedia software.

To improve the performance of program code, optimizations are performed for the compiler generated assembly code. The proposed technique reduces pipeline stalls by performing instruction scheduling, increases the number of instructions in the basic block, and selects the optimized instructions.

TABLE I. ARM CORE IN RECENT SMARTPHONES

Mobile CPU	Samsung Exynos 5 Dual	Samsung Exynos 5 Octa	Apple A6, A6X	Nvidia Tegra 3	Qualcomm APQ8064T
Phone Model	Samsung Chromebook, Google Nexus 10	Samsung Galaxy 5	Apple iPhone 5, 4G iPad	Google Nexus 7	LG Optimus G Pro, Samsung Galaxy S4 (GT-I9505)
ARM Core	ARM Cortex-A15 (ARMv7)	ARM Cortex-A15, Cortex-A7 (ARMv7)	Swift (ARMv7s)	ARM Cortex-A9 (ARMv7)	Krait (ARMv7)
NEON inside	O	O	O	O	O

Table II shows the overview of the proposed assembly optimizations. Instruction scheduling reorders the instructions to reduce pipeline stalls and increase instruction-level parallelism [1, 12]. Take the dual-issue Cortex-A8 processor for the target processor. Before instruction scheduling, the code takes 4 cycles. In the code, ‘ADD R10, R9, R5’ and ‘ADD R2, R1, #2’ instructions have no dependence each other, and thus, they can be reordered. This reordering can reduce execution cycles from 4 to 3 as shown in the Table.

Instruction selection improves the compiler generated code. It replaces the compiler selected instruction with the optimized one. The details will be discussed in Table III.

Loop unrolling [1, 12] replaces the body of a loop by multiple copies. This reduces the control hazard from the branch instruction because the number of loop iteration is decreased by the unrolling. On the other hand, it increases the number of instructions in a loop, which accordingly, increases instruction level parallelism and provides the opportunities for other optimizations such as CSE (Common Subexpression Elimination) [1, 12]. The details will be discussed in Table IV.

TABLE II. ASSEMBLY OPTIMIZATION OVERVIEW

Type	Description	Example

Instruction scheduling	Reorder instructions to reduce pipeline stalls	- Before scheduling (Cortex-A8, dual-issue)
		`LDR R5,[R4]        ; cycle 1`
		`ADD R10,R9,R5       ; cycle 3`
		`MOV R1,#1           ; cycle 3`
		`ADD R2,R1,#2        ; cycle 4`
		
		- After scheduling
		`LDR R5,[R4]       ; cycle 1`
		`MOV R1,#1         ; cycle 1`
		`ADD R2,R1,#2      ; cycle 2`
		`ADD R10,R9,R5     ; cycle 3`

Instruction selection	Improve code quality by replacing the existing instructions with the optimized instructions	Refer to Table III
Loop unrolling	Unroll loop to reduce branch hazard and increase instruction level parallelism	Refer to Table IV

III. ASSEMBLY OPTIMIZATION EXAMPLE

This chapter discusses the various assembly optimizations by examples, and analyzes the performance of the optimized code.

Table III shows the instruction selection for the compiler generated code. The C program finds and stores max value element from two matrices. To improve the performance, the compiler unrolls the loop 8 times, resulting in the 32 loop iterations from the original 256 iterations where eight max values are handled in each iteration. Consider the compiler generated assembly code. One vector containing eight 16-bit data is stored in d0 and d1. The other vector is stored in d2 and d3. Now, vector compare operation is performed by 'VCGT.S16 q2, q0, q1', which compares two 16-bit data from q0 and q1, and If the first data is greater than the second data, the corresponding element in the destination register is set to all ones, and zeros otherwise. Then, VBIT (Vector Bitwise Insert if True) instruction inserts each bit from the first operand register into the destination register if the corresponding bit of the second operand register is one, and leaves the destination register unchanged otherwise. These two instructions can extract and save the max value from two array elements, but it is more desirable to use 'VMAX' instruction directly, which get the maximum value between two values. The use of this instruction reduces the number of instructions in each loop from eight to seven.

TABLE III. NEON INSTURCIONT SELECTION OPTIMIZATION

C Code	`__inline max(int a, int b)`
	`{`
	`        if (a > b) return a;`
	`        return b;`
	`}`
	
	`void setMax (unsigned short * __restrict pDst, short * __restrict pSrc1, short * __restrict pSrc2)`
	`{`
	`        int i;`
	`        for (i = 0; i < 256; i++) {`
	`                pDst[i] = max(pSrc1[i], pSrc2[i]);`
	`          }`
	`}`

Compiler generate code	`setMax PROC`
	`    MOV      r3,#0x20`
	`|L1.4|`
	`    VLD1.16  {d2,d3},[r2]!`
	`    SUBS     r3,r3,#1`
	`    VLD1.16  {d0,d1},[r1]!`
	`    VCGT.S16 q2,q0,q1`
	`    VBIT     q1,q0,q2`
	`    VST1.16  {d2,d3},[r0]!`
	`    BNE      |L1.4|`
	`    BX       lr`
	`ENDP`

Instruction selection	`setMax PROC`
	`    MOV      r3,#0x20`

```
|L1.4|
    VLD1.16  {d2,d3},[r2]!
    SUBS     r3,r3,#1
    VLD1.16  {d0,d1},[r1]!
    VMAX.S16 q1,q0,q1
    VST1.16  {d2,d3},[r0]!
    BNE      |L1.4|
    BX       lr
ENDP
```

Table IV shows the loop unrolling optimization. For the C code, the compiler generates the loop which consists of three instructions, which are ‘VLD1.16 {d0, d1}, [r1]!’, ‘VMUL.I16 q0, q0, q1’, and ‘VST1.16 {d0, d1}, [r0]!’. Because these instructions handle 8 data in parallel, the loop iterates 32 times. One loop iteration takes 9 cycles including 6 cycles for ‘VMUL.I16 q0, q0, q1’. For the next iteration, it requires additional 2 stalls from the last instruction of the loop ‘VST1.16 {d0, d1}, [r0]!’ to the first instruction ‘VLD1.16 {d0, d1}, [r1]!’. Therefore, it requires (9+2) x (32-1) cycles for the 31 loop iterations, and 9 cycles for the last loop iteration, which is 351 cycles in total.

In the assembly optimized code, the loop is additionally unrolled 4 times such that the number of loop iteration is reduced from 32 to 8. Now, one iteration requires 39 cycles, and the required total cycles become 327. Thus, it can improve the compiler generated code by 6.8%.

TABLE IV. LOOP UNROLLING OPTIMIZATION

C Compiler generated code		loop unrolling optimized code by hand	
Execution time: 351 cycles		Execution time: 327 cycles (6.8% improvement)	
NEON instruction total cycles (Cortex-A8) (Loop iteration: 32) 1 + (9+2) x (32-1) + 9 = 351		NEON instruction total cycles (Cortex-A8) (Loop iteration: 8) 1+ (39 +2) x (8-1) + 39 = 327	
Instruction	Loop cycle	Instruction	Loop cycle

||sfC|| PROC		||sfC|| PROC	
VDUP.16 q1,r2		VDUP.16 q4,r2	
MOV r2,#0x20		MOV r2,#0x8	
|L1.16|		|L1.16|	
VLD1.16 {d0,d1},[r1]!	1	VLD1.16 {d0,d1},[r1]!	1
VMUL.I16 q0,q0,q1	2	VMUL.I16 q0,q0,q4	2
VST1.16 {d0,d1},[r0]!	9	VST1.16 {d0,d1},[r0]!	9
SUBS r2,r2,#1			
BNE |L1.16|		VLD1.16 {d2,d3},[r1]!	11
BX lr		VMUL.I16 q1,q1,q4	12
ENDP		VST1.16 {d2,d3},[r0]!	19
		VLD1.16 {d4,d5},[r1]!	21
		VMUL.I16 q2,q2,q4	22
		VST1.16 {d4,d5},[r0]!	29
		VLD1.16 {d6,d7},[r1]!	31
		VMUL.I16 q3,q3,q4	32
		VST1.16 {d6,d7},[r0]!	39
		SUBS r2,r2,#1	
		BNE |L1.16|	
		BX lr	
		ENDP	

IV. Conclusions

In this paper, various assembly level software optimization techniques are presented for the ARM NEON accelerator which is widely used in recent smartphones. The proposed techniques are expected to be directly applied in the software development for ARM NEON processors.

References

[1] A. V. Aho, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, USA, 1986.

[2] ARM Ltd., ARM Annual Report & Accounts 2015, ARM Ltd., 2016.

[3] ARM Ltd., ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile, ARM Ltd., 2013.

[4] ARM Ltd., ARMv8 Instruction Set Overview, ARM Ltd., 2012.

[5] ARM Ltd., Introducing NEON™ Development Article, ARM Ltd., 2009.

[6] ARM Ltd., Steele S., Java Program Manager. White paper: Accelerating to meet the challenge of embedded java, 2001.

[7] ARM Ltd., NEON support in the ARM Compiler, 2008.

[8] ARM Ltd., Overview of NEON Technology, 2012.

[9] D. Brash., The ARM Architecture Version 6, ARM White Paper, 2002.

[10] F. Hedley, ARM DSP-Enhanced Extensions, ARM Ltd., 2001.

[11] J. L. Hennessy, and D. A. Patterson, David, Computer Architecture, Fifth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., CA, USA, 2011.

[12] S.S. Muchnick, Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[13] S. Segars, K. Clarke, L. Goudge, "Embedded control problems, Thumb, and the ARM7TDMI". IEEE Micro, Vol. 15, No. 5, 1995, pp.22-30.