International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-2, Issue-5,May 2016
ISSN: 2395-3470
www.ijseas.com

C Program Optimizations for ARM NEON Processors

Dae-Hwan Kim
Department of Computer and Information, Suwon Science College, Hwaseong-si, Gyeonggi-do, Rep. of Korea

Abstract
ARM is the most widely used 32-bit embedded processor which is employed in smartphones, tablets, vehicles, wearable devices, and IoT (Internet of Things) devices. In the recent ARM processors for smartphones and tablets, the ARM NEON is widely deployed, which is the SIMD (Single Instruction Multiple Data) accelerator for multimedia and signal processing algorithms. In this paper, various C program optimization techniques are presented such as loop unrolling and function inlining for the NEON architecture. The proposed techniques can greatly enhance the generated assembly code both in size and execution speed. The proposed techniques do not depend on the specific platform, and thus, they are expected to be applied to the software development for the ARM NEON processors.
Keywords: *ARM, NEON, Software Optimization, C Programming Language, Loop Unrolling, Function Inline.*

1. Introduction

ARM [3-9, 12-13] is the most dominant 32-bit embedded processor which is deployed in various embedded systems such as smartphones, tablets, vehicles, wearable devices, and IoT (Internet of Things) devices. In 2015, 15 billion ARM-based chips are sold, and the ARM's market share is over 95% in the smartphone market [2]. The first commercial version is ARM7TDMI [13], which is introduce in 1995. The architecture version of ARM7TDMI is four, and the successors of ARM7TDMI are continuously introduced to meet the market demands [3-9, 12]. The most recent commercial version is ARMv8 (version 8) [3-4].

Thumb-2 is the new instruction set from the ARMv7 architecture [12]. This instruction set combines 32-bit ARM instructions and 16-bit Thumb instructions into a single instruction set where the 16-bit Thumb instructions are the bit-width reduced versions for 32-bit ARM instructions. Thumb-2 architecture contains both 32-bit ARM instructions and 16-bit Thumb instructions. This eliminates mode conversion overhead between Thumb and ARM modes in the previous architectures. In addition, new instructions such as bit manipulation instructions are added to improve the processor performance. With this integration, Thumb-2 provides 16-bit Thumb code density while preserving 32-bit ARM performance.

SIMD (Single Instruction Multiple Data) [10] engine called NEON [6-7] is introduced from the ARMv7 architecture. The NEON instruction set can execute several arithmetic operations in parallel, and thus, it is useful for audio and video codecs, graphics, image processing, and audio processing algorithms. The NEON is employed in various recent ARM processors such as Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15, and Cortex-A53 processors where the Cortex-A are the latest ARM processors for performance intensive systems. NEON is used in most recent smartphone application processors. It is included in Cortex-A9 for NVidia's Tegra3, Cortex-A15 and Cortex-A7 for Samsung's Exynos 5, and Apple's Swift architecture.

C language is one of the most widely used programming languages, especially for embedded systems. Therefore, it is important to optimize C code for ARM processors. In this paper, various C program software optimization techniques are presented for the NEON C code.

The rest of this paper is organized as follows. Section 2 shows the overview of the C program optimization techniques, and Section 3 explains each technique in detail with an example. Conclusions are presented in Section 4.

2. C Program Optimization Overview

Table 1 shows the overview of the proposed C program optimization techniques. The __promise keyword notifies to the compiler the expression that is true. This enables the compiler to generate efficient code by performing more aggressive optimizations.

One of the most difficult challenges in compiler optimization is the pointer variable analysis. When the compiler encounters a pointer variable, it is hard to know where the pointer is pointing. It becomes more complicated because different pointer variables can point to the same or the overlapping memory locations. Thus, compilers take conservative approaches [1, 11] for optimizations relevant to pointer variables.

Programmers can give information on the pointer to the compiler. The __restrict keyword declares that different pointers do not point to the same and overlapping memory region at runtime. It guarantees that the data pointed by the pointer variable is not modified by the other pointer variables. This enables the compiler to perform optimizations which otherwise are prevented by pointer aliasing. Therefore, the compiler can perform various optimizations which result in the efficient code.

The __inline intrinsic is the directive which suggests the compiler to substitute the function call by the body of the function. Compiler inserts the function code at each function call, which removes the function call overhead. In addition, the size of the basic block is increased, which accordingly increases the optimization opportunity for vectorization. In a structure programming like C programming language, the complex function is normally divided into sub functions. When a function is called in a loop, the NEON vectorization is often restricted at the function call instruction. The __inline keyword gives vectorization opportunity including the code of subfunctions together.

Loop unrolling replicates the body of the loop and decreases the number of iterations. This reduces loop overhead occurring from the branch instruction, and increases the number of instructions in the loop, which provides more vectorization opportunity. This technique attempts to improve the execution speed at the expense of the code size in the space-time trade-off. However, the unrolling by the compiler is often conservative and not sufficient even when the resources such as registers are available for more unrolling. Thus, hand unrolling by the programmer can achieve the better performance. Programmers can be guided by the compiler generate assembly code. The loop unrolling count and resource usages can be easily estimated in the assembly code, and thus, it is not difficult to determine the resource availability. When resources are available, the programmer can increase the unrolling number by using the pragma unroll function.

The vectorization can be more improved if the hint is given for the number of loop iterations. For example, when the number of iteration is a multiple of four, this information can be explicitly given in the loop termination condition. This gives the compiler the vectorization hint.

Table 1: C optimization overview

Technique	**Description** / **Example**
Use __promise	It notifies to the compiler the expression that is true. This enables the compiler to generate more efficient code by performing aggressive optimizations.

	`__promise (0<len&& (len% 8) ==0);`
	
	`for (int i=0; i<len; i++) { ... }`

Use __restrict keyword for pointer variables	It guarantees that there is no other pointers point to the memory block pointed by the __restrict pointer variable.
	`__restrict *ptr;` This declaration notifies the compiler that the area pointed by ptr is not referenced by other pointer variables.
Use __inline	This keyword guides the compiler to replace the function call with the function body. It provides more opportunity for vectorization by increase the number of instructions in the basic block.

	`__inline add(int a, int b)`
	`{`

	`return a+b;`
	`}`
	
	`int caller(int x, int y)`
	`{`
	`add(x, y);`
	`}`

loop unroll	It specifies the optimal loop unrolling number to the compiler.
	`#pragma unroll(n)`
Inform loop iteration number	It gives the loop iteration information to the compiler for the vectorization.
	If the array size is the multiple of four, the following code gives the loop iteration information to the compiler. `for(n=0; n<(limit/4)*4; n++)`

Table 2 shows the result of the C level software optimizations. When no optimization keyword is specified for variables d, n, and m, the ARM Realview compiler generates total 60 instructions which consist of 4 NEON instructions and 56 Thumb-2 instructions. Many instructions are generated because the compiler can not be sure that memory locations pointed by variables d, n and m are not overlapping. In addition, because no information on the loop iteration number is given, compiler should generate code to handle various possible loop iterations. If point variables d, n, and m do not point to the same and overlapping memory regions, we can specify those pointers with __restrict keyword. Furthermore, if variable len is a multiple of eight, we can notify to the compiler that information for optimization. As the result of specifying two keywords, we can obtain far more compact code which consists of total 8 instructions including 4 NEON instructions.

Table 2: C optimization example code

Type	**Description**

Original C Code	// suppose that len is multiple of eight
	
	`void vadd (`
	`short* d, short* n,`
	`short* m, int len)`
	`{`
	`int i;`
	
	`for(i=0; i<len; i++)`
	`{`
	`d[i]= n[i] + m[i];`
	`}`
	`}`

Original Assembly Code	vadd PROC PUSH {r4-r6} ; test if d and n alias CMP r0,r1 BLS |L1.24| SUB r12,r0,r1 CMP r3,r12,ASR #1 BGT |L1.176| |L1.24| ; test if d and m alias CMP r0,r2 BLS |L1.44| SUB r12,r0,r2 CMP r3,r12,ASR #1 BGT |L1.176| |L1.44| ; test if len is multiple of 8 CMP r3,#0 BLE |L1.168| ASR r12,r3,#31 MOV r4,r1 MOV r5,r2 ADD r12,r3,r12,LSR #2 MOV r6,r0 ASRS r12,r12,#3 BEQ |L1.104| |L1.80| ; vector loop VLD1.16 {d0,d1},[r4]! SUBS r12,r12,#1 VLD1.16 {d2,d3},[r5]! VADD.I16 q0,q0,q1 VST1.16 {d0,d1},[r6]! BNE |L1.80| |L1.104| AND r12,r3,#7 ; cleanup loopcount CMP r12,#0 BLE |L1.168|

	SUB r12,r3,r12
	CMP r12,r3
	BGE |L1.168|
	|L1.128|
	; clean-up loop
	ADD r4,r1,r12,LSL #1
	ADD r5,r2,r12,LSL #1
	ADD r6,r0,r12,LSL #1
	LDRH r4,[r4,#0]
	ADD r12,r12,#1
	LDRH r5,[r5,#0]
	CMP r12,r3
	ADD r4,r4,r5
	STRH r4,[r6,#0]
	BLT |L1.128|
	|L1.168|
	; return sequence
	POP {r4-r6 }
	BX lr
	|L1.176|
	; test loop count >0
	CMP r3,#0
	MOV r12,#0
	BLE |L1.168|
	|L1.188|
	; non vector loop
	ADD r4,r1,r12,LSL #1
	ADD r5,r2,r12,LSL #1
	ADD r6,r0,r12,LSL #1
	ADD r12,r12,#1
	LDRH r4,[r4,#0]
	CMP r12,r3
	LDRH r5,[r5,#0]
	ADD r4,r4,r5
	STRH r4,[r6,#0]
	BLT |L1.188|
	POP {r4-r6 }
	BX lr
	ENDP
	Total instruction count: 60 (NEON: 4 + Thumb-2: 56)
Optimized C Code	`void vadd(`
	`short* __restrict d,`
	`short* __restrict n,`
	`short* __restrict m,`
	`int len)`
	`{`
	/* len is positive and a multiple of 8 */
	`__promise(0<len && (len% 8) == 0);`
	`int i;`
	`for (i=0; i<len; i++) {`
	`d[i] = n[i] + m[i];`
	`}`
	`}`
Optimized Assembly Code	vadd PROC
	ASR r3,r3,#3
	|L0.4|
	VLD1.16 {d2,d3},[r1]!
	VLD1.16 {d0,d1},[r2]!
	VADD.I16 q0,q1,q0
	VST1.16 {d0,d1},[r0]!
	SUBS r3,r3,#1
	BNE |L0.4|
	BX lr
	ENDP
	Total instruction count: 8 (NEON: 4 + Thumb-2: 4)

4. Conclusions

In this paper, various C program software optimization techniques are presented for the ARM processors, mainly targeted for the NEON multimedia accelerator which is widely used in recent smartphones and tablets. Most of proposed techniques do not depend on the specific hardware/OS platform, and therefore, they are expected to be applied to 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., NEON support in the ARM Compiler, 2008.
[7] ARM Ltd., Overview of NEON Technology, 2012.
[8] D. Brash., The ARM Architecture Version 6, ARM White Paper, 2002.
[9] F. Hedley, ARM DSP-Enhanced Extensions, ARM Ltd., 2001.
[10] J. L. Hennessy, and D. A. Patterson, David, Computer Architecture, Fifth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., CA, USA, 2011.
[11] S.S. Muchnick, Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco, CA, USA, 1997.
[12] R. Phelan, Improving ARM Code Density and Performance, Technical Report, ARM Ltd., 2003.
[13] S. Segars, K. Clarke, L. Goudge, "Embedded control problems, Thumb, and the ARM7TDMI". IEEE Micro, Vol. 15, No. 5, 1995, pp. 22-30.

Dae-Hwan Kim is an associate professor at the Department of Computer and Information in Suwon Science College. Previously, he worked as a Principal Engineer at Samsung Electronics and LG Electronics where he developed commercial compilers and multimedia embedded processors. He received the Ph.D. degree from the School of Electrical and Computer Engineering of Seoul National University in 2010, and received the B.S. and M.S. degrees in Computer Science from Seoul National University in 1993 and 1995, respectively. His current research interests include computer architecture, compiler, and embedded systems.