TMS320 DSP
Number xx.0
DESIGNER'S NOTEBOOK
TEXAS INSTRUMENTS

Nested Loop Optimization on the 'C6x

Contributed by Richard Scales

Design Problem

In many typical DSP applications, loops comprise a majority of the number of cycles, or MIPS. Because of this, performance of loops can greatly affect the performance of the entire application. Many of these loops are nested loops with both an inner and outer loop. Some common examples are FIR and IIR filters, FFT, and DCT. To optimize these nested loops it is necessary to consider not only the inner loop performance, but also the outer loop performance, especially when the inner loop count is small for execution of each outer loop.

One technique used to optimize loops on the highly parallel 'C6x VelociTI architecture is software pipelining. This involves initiating new iterations of the loop before previous iterations have completed to obtain high throughput. This implies there are some cycles (loop prologue) to begin executing, or pipe up, of each inner loop and some more cycles to pipe down the loop (loop epilogue). These cycles will be incurred each outer loop so they can affect performance, especially when the inner loop count is small. The more deeply pipelined the DSP is, the more cycles will be required for the prologue and epilogue.

Figure 1 shows a simple dot product example, (with non-C6x like single cycle loads and multiplies), where inner loop setup is 2 cycles, the prolog is 2 cycles, the epilog is 2 cycles, and the time to execute outer loop instructions is 2 cycles. At the end of cycle 9 there is a branch back to the beginning of the loop setup (Br 1). Thus, 8 cycles will be incurred each time this inner loop is executed in an outer loop. As we move to deeper and deeper pipelines in DSPs for higher clock speeds, the number of cycles of overhead will increase. The higher the number of cycles for setup, prolog, epilog, and outer loop instructions, and the lower the inner loop count, the more overall nested loop performance is reduced.

	cycle	.D1	.D2	.M1	.S1	.S2
Setup for Inner Loop	1	add	add			
	2	add	add			
Prolog	3	ldh	ldh			
Staging for loop	4	ldh	ldh	mpy		
Single-cycle "loop"	5	ldh	ldh	mpy	add	Br
Epilog Completing final operations.	6			mpy	add	
	7				add	
Outer Loop Instructions	8	add	add			
	9	add	add		Br 1	

Figure 1. Nested Loop w/ Software Pipelined Inner Loop

Solution

This designer notebook page will present techniques for reducing and even eliminating the extra cycles due to inner loop setup, prologs and epilogs normally seen in nested loops.

1) Pipeline Outer Loop

	cycle	.D1	.D2	.M1	.S1	.S2	.L1	.L2
Setup for Inner Loop	1	add	add					
	2	add	add					
Prolog	3	ldh	ldh					
Staging for loop	4	ldh	ldh	mpy				
Single-cycle "loop"	5	ldh	ldh	mpy	add	Br		
Epilog - Prolog Setup - Outer Loop instructions.	6			mpy	add	add	add	add
	7	ldh	ldh		add	add	add	add
	8	ldh	ldh	mpy	Br 5	add		add

Figure 2. Nested Loop w/ Software Pipelined Outer Loop

The performance of the loop Figure 2 shows that now there are only 3 cycles in the outer loop because now cycle 8 contains a branch directly to the inner loop (cycle 5).

For more detailed information on this technique, consult the section in the Assembly Optimizations Chapter on Software Pipelining Outer Loops.

2) Conditionally Execute Outer Loop

A second and even more powerful technique for improving outer loop performance is to conditionally execute all inner loop setup and outer loop instructions in parallel with the inner loop. If there are a lot of instructions for the setup and outer loop, this can slow

down inner loop performance (there might not be enough empty slots in the loop to do all the extra instructions). In some cases though, this can still be an overall savings depending on the inner loop count.

Consider the following case:

Inner loop cycles are 4 and outer loop cycles are 10. Thus the total loop cycles are y*(4x+10) where x and y represent the number of inner and outer loop counts respectively. If we slow down the inner loop by 1 cycle to avoid outer loop cycles, the formula becomes y*(5x). So for inner loop counts of x < 10, the slower inner loop yields faster overall results.

In other cases, it is useful to unroll inner loop (to increase the effective number of spare slots) and execute at the same inner loop performance. This effectively eliminates all outer loop overhead. The formula for the above example would then be y*(4x).

	cycle	.D1	.D2	.M1	.S1	.S2	.L1	.L2
Setup for Inner Loop	1	add	add		add			
	2	add	add					
Prolog	3	ldh	ldh					
Staging for loop	4	ldh	ldh	mpy				
Three-cycle "loop" with Setup/Outer Loop Instructions Cond. In Parallel	5	ldh	ldh	mpy	add	add	add	add
	6	ldh	ldh	mpy	add	add	add	add
	7	ldh	ldh	mpy	add	Br	add	add

Figure 3. Conditionally Executed Outer Loop

Figure 3 shows that the inner loop has been unrolled three times to allow enough slots to insert all loop setup and outer loop instructions conditionally and completely avoid any outer loop overhead.

For more detailed information on this technique, consult the section in the Assembly Optimizations Chapter on Conditionally Executing Outer Loops.