Optimized Parallel Implementation of Polynomial Approximation Math Functions on a DSP Processor

Mei Yang, Jinchu Wang, Yuke Wang and S.Q. Zheng
Department of Computer Science
Univ. of Texas at Dallas, Richardson, TX 75083
{meiyang, yuke, sizheng}@utdallas.edu

Abstract

This paper presents a general method to implement polynomial approximation math functions on TMS320C67X architecture with multiple parallel execution units. Our method consists of grain packing, mapping and scheduling to reduce data dependency overhead and fully utilize delay slots. Experimental results of our method on TMS320C67x have achieved up to 70.2% performance improvement over 'C67x library functions.

1. Introduction

Many real-time applications in the areas of signal processing, process control, etc., require very fast implementation of a large number of mathematical functions. Four commonly used math functions include: a) the exponential function, b) the logarithmic function, c) trigonometric functions, and d) inverse trigonometric functions [1]. These functions are generally implemented by polynomial approximation. Most C compilers of DSP processors provide these math functions in their libraries.

The TMS320C67x is the general-purpose floating-point DSP family in the TMS320C6000™ DSP platform. It is based on TI advanced VelociTI very-long-instruction-word (VLIW) architecture, which can execute up to eight instructions every clock cycle and achieve up to 1 Giga floating-point operations per second (GFLOPS) [2]. The high performance features make this DSP an excellent choice for multi-channel and multifunction applications, which have large numbers of math functions involved.

The C library functions provided by 'C67x use polynomial approximation to implement these commonly used math functions. The efficiency of these functions is quite low due to the high data dependency and deep delay slots of instructions, which makes it very difficult to do parallel scheduling. The optimization tools in its compiler do not work well for exploring the parallelism of computations with dependent data.

This paper proposes a grain packing approach to reducing the data dependency and improving parallel scheduling for polynomial approximation functions. The proposed method is based on partitioning computations into groups such that the inter-group data dependency is minimized. The implementation of our optimization method in assembly code takes advantage of the VLIW architecture of 'C67x. Experimental results have shown that performance (total cycle counts) improvement of our method over TI 'C67x library functions is up to 70.2%.

2. TMS320C67x DSP Processor

With performance of up to 1 GFLOPS and a complete set of development tools, the TMS320C67x offers cost-effective solutions to high-performance floating-point DSP programming challenges [3].

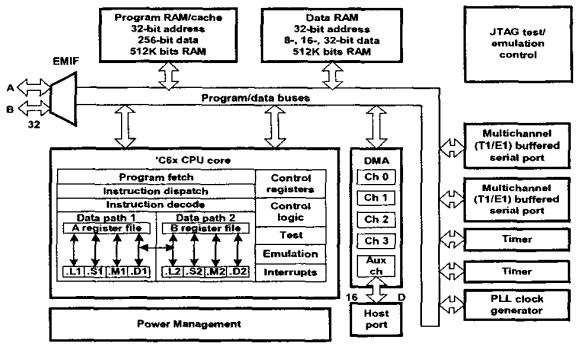

Figure 1. TMS320C67x block diagram

 0-7803-7150-X/01/$10.00@2001 IEEE

The high performance levels of the TMS320C67x DSP chips are made possible by an innovative architecture designed to meet a variety of applications. Figure 1 shows the TMS320C67x architecture [2]. The 'C67x processor consists of three main parts: CPU (or the core), peripherals, and memory. Eight functional units operate in parallel, with two similar sets of the basic four functional units. These units communicate using a cross-path between two register files, each of which contains 16 32-bit registers. 'C67x provides a large bank of on-chip memory and has a powerful and diverse set of peripherals.

Each 'C67x instruction has a functional unit latency of 1 clock cycle followed by variable delay slots, each corresponding to 1 clock cycle [4]. For 'C67x applications, delay slots must be taken care of in parallel scheduling. If instructions are scheduled properly, 'C67x hardware resources can be fully utilized to hide instruction execution latencies and speed up computations.

Due to its high performance, ease of use, and affordable pricing, TMS320C67x generation is an ideal solution for multi-channel, multi-function floating-point applications, such as Digital Receiver, Virtual Reality 3-D graphics, Audio, Radar, etc. These applications involve many math functions including the four commonly used functions mentioned earlier. These functions are generally implemented by polynomial approximations [5].

3. Polynomial Approximation Math Functions

The polynomial approximation method is fundamentally very simple. If a function $f(x)$ has continuous derivatives up to $(n+1)^{th}$ order, then this function can be expanded in the following fashion [5]:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)(x-a)^2}{2!} + \ldots + \frac{f^{(n)}(a)(x-a)^n}{n!} + Rn \quad (1)$$

where Rn, called the remainder after $n+1$ terms. If $a = 0$ the series is called the MacLaurin Series.

To obtain the desired accuracy, TI 'C67x library approximate these functions by a polynomial of certain sufficient order, the same technique introduced for 'C3x [6]. The polynomial [6] can be generally expressed as:

$$P(n,x) = \sum_{i=0}^{n} \{a[i]x^i\}, \quad (2)$$

where x is the independent variable, n is the polynomial order (a fixed integer), and a[i] is a set of $n+1$ fixed coefficients. So the desired function, say $f(x)$ is then approximated by a particular $P(n,x)$ such that:

$$f(x) = P(n,x) + e(x), \quad x1 < x < xu, \quad (3)$$

where $x1$ and xu are the limits of the domain of x, and $e(x)$ or $e(x)/f(x)$ is the error function which has been usually minimized in the min-max (equi-ripple) sense. This is done by selecting appropriate coefficients a[i]. The adjusted a[i] are somewhat different from the coefficients listed in [5]. General description of polynomial approximation techniques used in TI math library functions can be found in [6].

4. How TI 'C67x library Functions Work

TI 'C67x provides C library functions of these commonly used functions for both single-precision floating point and double-precision floating point [7]. Our discussion is focused on single-precision floating-point functions. The implementation of these functions all involve a loop of additions and multiplications of a[i] and x^i. Taking the example of natural logarithm function, *logf* routine, its C main loop [7] is shown in Figure 2, which is based on equation (4).

$$\ln(1+x) = k_1x + k_2x^2 + k_3x^3 + k_4x^4 + k_5x^5 + k_6x^6 + k_7x^7 + k_8x^8$$
$$= x(k_1 + x(k_2 + x(k_3 + x(k_4 + x(k_5 + x(k_6 + x(k_7 + xk_8)))))))) \quad (4)$$

```
{int i;
 float *p=a;
 result=(f)*(*p++);
 for (i=8-1; i>0; i--)
        result=(f)*(((result)+(*p++)));
}
```

Figure 2. The main loop of logf function in 'C67x library

In Figure 2, f represents the independent variable x. It's very simple for C implementation. However, the high data dependency between iterations (the i^{th} result depends on the $(i+1)^{th}$ result) and long delay slots of floating-point instructions of 'C67x make it difficult to do parallel scheduling. Therefore the efficiency of these functions on 'C67x is quite low. Figure 3 shows the compiled assembly code on Code Composer Studio1.2 (CCS1.2). The latency of each iteration is 8 cycles. The total number of cycles of equation calculation is 60. Plus the cost of pre-processing and post-processing, the maximum total running time of *logf* function (with C overhead) is 136 cycles, that is 814ns when system clock is 167M Hz. The running time of other functions (*expf, sinf, cosf and atanf*) is up to 265 cycles. All these cycle numbers are reported by CCS1.2.

```
1. LDW      *p_to_coef++, coef_I    ;delay 4 slots
2. MPYSP    f, sum, result          ;delay 3 slots
3. NOP      3
4. ADDSP    result, coef_i, sum     ;delay 3 slots
5. NOP      2
```

Figure 3. The assembly code for each iteration of logf function in 'C67x library

It is desirable to optimize the efficiency of these library functions, which are heavily used in many real-time applications. Unfortunately, the performance of these functions cannot be improved using all the optimization options of the parallelizing compiler CCS1.2. While these optimization techniques (e.g. software pipelining [8]) are suitable for applications with a large number of independent variables, such as FFT, FIR, IIR, etc.; they do not work well for an implementation with dependent data as shown in equation (4) and Figure 2. In the following, we will introduce our general method to improve the efficiency of these polynomial approximation math functions.

5. Optimized Parallel Implementation of Polynomial Approximation Math Functions

For the same computational problem, different algorithms result in different data dependency structures. When mapped to a particular architecture, different algorithms may lead to different performance. Our goal of optimization is to improve the parallelism of computations by finding an algorithm suitable for TMS320C67x architecture. To achieve this goal, we need to reduce data dependency and utilize delay slots of instructions. Our approach consists of finding a computing process equivalent to the polynomial expansion, partitioning this process into independent groups and mapping these groups to function units of 'C67x so that delay slots can be utilized.

Taking example of *logf* function, we will explain how the optimization method works. First, equation (4) can be derived as the following:

$$\begin{aligned} ln(1+x) &= k_1x + k_2x^2 + k_3x^3 + k_4x^4 + k_5x^5 + k_6x^6 + k_7x^7 + k_8x^8 \\ &= x(k_1 + k_2x) + x^3(k_3 + k_4x) + x^5(k_5 + k_6x) + x^7(k_7 + k_8x) \end{aligned} \quad (5)$$

The fine-grain data dependency graph is thus obtained in Figure 4, which is different from traditional fine-grain program graph [9]. Instead of improving performance by reducing inter-processor communications, we use grain packing technique to assign function units and fully utilize delay slots. Here each node denotes an instruction, L, M and + represents the instruction for load, multiplication and addition respectively. Inside each node is the destination operator. Each edge represents the data dependency of next node (edge head) to previous node (edge tail). The number on each edge gives the instruction delay slots. There are 28 nodes in this graph.

Then we use grain packing [9] to optimize parallel scheduling. Figure 5 presents the grain packing graph which groups 28 small nodes into 16 larger nodes (named as $A_1 \ldots A_9$, $B_1 \ldots B_7$). We observe that operations of the nodes at the same level can be executed in parallel; nodes on different levels should be scheduled according to data dependency; and computation in each node should be done sequentially and at the same side of function units. Computation of x^{2i+1} in node A_9 can be scheduled in the delay slots of other nodes.

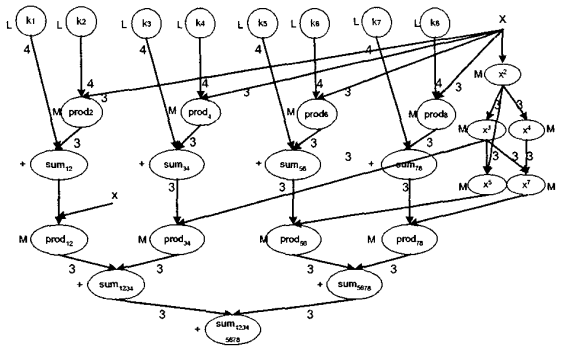

Figure 4. Fine-grain data dependency graph of equation (5)

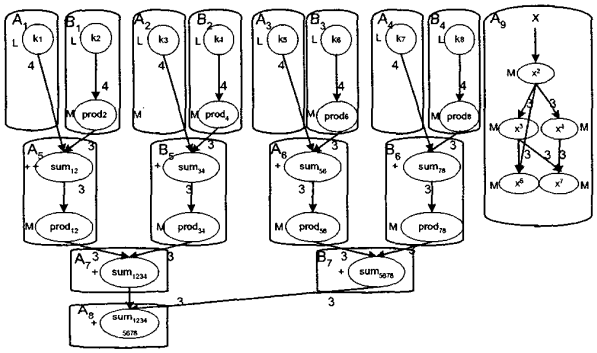

Figure 5. Grain packing graph of equation (5)

'C67x has eight functional units operating in parallel, but each floating-point instruction can only use 2 function units, such as LDW (.D1 and .D2), MPYSP (.M1 and .M2) and ADDSP (.L1 and .L2) [4]. Due to this limitation, the maximum paralleled operations of loads, multiplications or additions in one cycle are 2. Then we get the optimized parallel scheduling of packed graph shown in Figure 6, the shaded area are delay slots. The total number of cycles is only 25, which reduces 58.3% compared to 60.

Generally, these polynomial expansions can be divided into two types, derived as equations (6)-(7).

$$\begin{aligned} f_1(x) &= k_0 + k_1x + k_2x^2 + k_3x^3 + k_4x^4 + k_5x^5 + k_6x^6 + k_7x^7 + k_8x^8 \\ &= k_0 + x(k_1 + k_2x) + x^3(k_3 + k_4x) + x^5(k_5 + k_6x) + x^7(k_7 + k_8x) \end{aligned} \quad (6)$$

$$\begin{aligned} f_2(x) &= k_0 + k_1x + k_2x^3 + k_3x^5 + k_4x^7 + k_5x^9 + k_6x^{11} + k_7x^{13} + k_8x^{15} + k_9x^{17} \\ &= k_0 + k_1x + x^3(k_2 + k_3x^2) + x^7(k_4 + k_5x^2) + x^{11}(k_6 + k_7x^2) + x^{15}(k_8 + k_9x^2) \end{aligned}$$

(7)

e^x, $ln(1+x)$ belong to type 1, denoted as $f_1(x)$. Type 2, denoted as $f_2(x)$, includes $sin(x)$, $cos(x)$ and $tan^{-1}(x)$. Similar to $f_1(x)$, we can get the optimized scheduling for $f_2(x)$. The running time of equation calculation of $f_2(x)$ is 29 cycles. In next section, we will give the performance of our optimization method implemented on TMS320C67x and comparison of it with TI 'C67x library function.

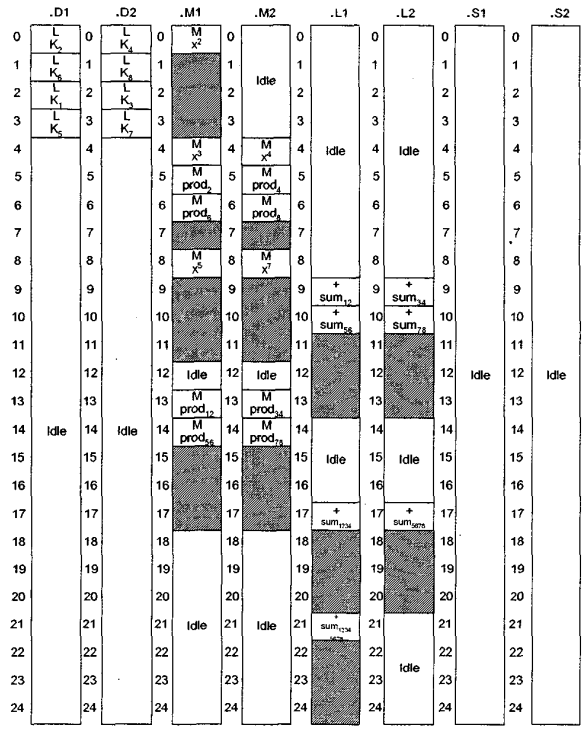

Figure 6. Parallel scheduling for computation of equation (5)
Notes: $prod_{2i+2}$ represents $k_{2i+2}x^2$, $prod_{(2i+1)(2i+2)}$ represents $x^{2i+1}(k_{2i+1}+k_{2i+2}x)$, $0 \le i \le 3$; $sum_{(2i+1)(2i+2)}$ represents $(k_{2i+1}+k_{2i+2}x)$, and so on, $0 \le i \le 3$.

6. Experimental Results

To evaluate the effectiveness of our proposed optimization method, experiments of these commonly used math functions are conducted on TMS320C67x. All assembly programs are written and debugged on Code Composer Studio1.2. Test programs of our optimization functions and 'C67x library functions are profiled and number of clocks for these functions with C overhead are measured in CCS1.2. The maximum running time (all pre-processing and post-processing are counted) in clock cycles (CLKs) of these new functions using our optimized method vs. 'C67x library functions are given in Table 1.

Noticeably, the improvement of our optimization method is over 60% and up to 70.2%. It's expected that other math functions using polynomial approximation can also be improved by our optimization.

Function	Optimized Function(CLKs)	TI Library Function	Improvement Percent
expf	74	213	65.3%
logf	44	136	67.6%
sinf	69	173	60.1%
cosf	73	183	60.1%
atanf	79	265	70.2%

Table 1. Comparison of Optimized Implementation vs. TI library Function in terms of total cycles

7. Conclusion Remarks

DSP processors have gained more importance and popularity in recent years for a wide variety of applications. To implement commonly used mathematical functions on DSP processors with parallel execution units is an important and non-trivial task. In this paper, we have presented a general optimization method based on the grain packing to achieve optimal parallel scheduling for those commonly used mathematical functions. Experiment results of our new implementation using this method have achieved significant performance improvement over the library functions of TMS320C67x provided by TI, the manufacturer of the DSP processors. This work is valuable to real-time applications using DSP, and it is also applicable to other DSPs with parallel execution units.

References

[1]. Das, D.; Mukhopadhyaya, K.; Sinha and B.P., "Implementation of four common functions on an LNS co-processor", *IEEE Transactions on Computers*, Volume: 44 Issue: 1 , Jan. 1995, pp. 155-161.

[2]. Texas Instruments, *TMS320C62x/C67x Technical Brief*, 1998.

[3]. Texas Instruments website, http://focus.ti.com/docs/prod/productfolder.jhtml?genericPartNumber=TMS320C6701&pfsection=desc.

[4]. Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000.

[5]. Milton Abramovitz and Irene A. Stegun, *Handbook of Mathematical Functions*, New York: Dover Publications, 1965.

[6]. Texas Instruments, *A Collection of Functions for the TMS320C30*, 1990.

[7]. Texas Instruments, *rts.src*, TMS320C62x/C67x library functions package with Code Composer Studio1.2.

[8]. Yin-Tsung Hwang, Ying-Chou Chuang, "High Performance Code Generation For VLIW Digital Signal Processors", *Signal Processing Systems, 2000. SiPS 2000. 2000 IEEE Workshop on*, 2000, Page(s): 683 -692

[9]. Kai Hwang, *Advanced Computer Architectures*, McGraw-Hill, 1993.