Application Notes
ZHCA765 – April 2018

AM57x 系列处理器 DSP 性能优化方法

Allen Yin
Processor Application

摘　要

AM57x 系列处理器是 TI 推出的高性能异构处理器，在该处理器中集成了 ARM Cortex A15, C66x DSP， ARM Cortex M4， PRUSS 等多个处理单元，用户的难题在于如何在特定的应用场景中充分发挥各个处理单元的性能。其中，C66x DSP 单元主要用于定点和浮点的算法实现，本文将介绍如何对 C66x DSP 部分的代码进行优化以达到理想的运算效率。

目　录

Figures

Tables

1 AM57x 系列处理器简介

Sitara AM57x 处理器是 TI 推出的一系列高性能异构处理器解决方案，目前已在计算机视觉，通信，工业等多个领域得到广泛的应用，AM57x 系列的处理能力主要由支持 NEON 指令集的 ARM Contex-A15 RISC CPU 核心以及支持定浮点超长指令集（VLIW）的 C66x DSP 核心组成。通常用户会在 ARM 核上运行 Linux，Android 或其它操作系统，而 DSP 则主要负责大量数据运算的工作。

2 C66x DSP 代码优化

2.1 估计算法复杂度

在对循环代码进行优化之前，需要了解该循环代码的算法复杂度，而 DSP 主要擅长的工作是乘加运算，因此首先需要对整个待优化算法的乘加数目进行估计以得到理想的优化目标。

下面以一个短整型向量乘加运算为例来说明复杂度的计算，

```
short  a[BUF_SIZE], b[BUF_SIZE];
int  i,sum;
for (i=0; i < BUF_SIZE; i++)
{
                    sum = sum + a[i] * b[i];
}
```

每一次循环需要做一次短整型乘法操作，两次 16bit（短整型）数据的读入以及一次加法，此处没有写入操作因为结果 sum 在循环里可存放在寄存器里，不需要频繁的写内存。由此可以得到每次循环的乘加运算以及内存操作数目以及未优化时 DSP 计算需要的 Cycle 数，如下表

Table 1. C66x 核运算单元

算法操作	乘法（M1,M2）	加法（L1,L2,S1,S2）	读写内存（D1,D2）
次数	1 (16 bit)	1 (32 bit)	2 (16 bit)
Cycle	1	1	1

通过查阅 C66x DSP 指令手册，其中乘法器 M1，M2 均可支持 DMPY2 指令可同时进行 4 路 16 bit 乘法，加减法均可在 L 单元和 S 单元上执行，且 C66x 支持 DADD 和 DSUB 操作，即每个单元均可支持双路 32 bit 的加减法，内存读写单元 D1 和 D2 可以同时分别读取 64 bit 的数据，由此得到 C66x DSP 的计算能力

Table 2. C66x 核运算能力

算法操作	乘法（M1,M2）	加法（L1,L2,S1,S2）	读写内存（D1,D2）
DSP 计算能力	8 (16 bit)	8 (32 bit)	8 (16 bit)

因此可以得到整个循环代码的理论优化目标为 1/4×BUF_SIZE 的 Cycle 数

Table 3. 向量乘加算法优化目标

算法操作	乘法（M1,M2）	加法（L1,L2,S1,S2）	读写内存（D1,D2）
次数	1 (16 bit)	1 (32 bit)	2 (16 bit)
DSP Cycle	1/8	1/8	1/4

在得到理论优化上限之后，就可以使用编译器反馈以及选项最大化代码效率。

2.2 获取编译器反馈

TI 提供的 DSP 编译器可以给用户反馈很多有用的信息，在 CCS 编译器的选项里选择-k 保留汇编文件，-mw 选项打开编译器的信息反馈，这些反馈可以从编译器生成的汇编文件中读取。

仍以第一节的代码为例

```
int dotp_c(short *a, short *b, int count)
{
int i;
 int sum = 0;

 for (i=0; i < count; i++)
 {
                        sum = sum + a[i] * b[i];
 }
 return(sum);
}
```

打开-k 及-mw 选项以后，编译器输出为

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 6
;*      Loop opening brace source line   : 7
;*      Loop closing brace source line   : 9
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        0
;*      .D units                     1*       1*
;*      .M units                     1*       0
;*      .X cross paths               1*       0
;*      .T address paths             1*       1*
;*      Long read paths              0        0
;*      Long write paths             0        0
```

```
;*     Logical  ops (.LS)         0       0     (.L or .S unit)
;*     Addition ops (.LSD)        1       0     (.L or .S or .D unit)
;*     Bound(.L .S .LS)           0       0
;*     Bound(.L .S .D .LS .LSD)   1*      1*
;*
;*     Searching for software pipeline schedule at ...
;*       ii = 1  Schedule found with 8 iterations in parallel
;*     Done
;*
;*     Loop will be splooped
;*     Collapsed epilog stages       : 0
;*     Collapsed prolog stages       : 0
;*     Minimum required memory pad   : 0 bytes
;*
;*     Minimum safe trip count       : 1
;*--------------------------------------------------------------------------*
```

其中对于代码性能优化比较关键的几条信息解释如下：

- Loop Carried Dependency Bound

  如果该项数值较大，一般原因是一次循环的结果可能要被下一次循环调用，或者用户没有给予编译器足够的信息，编译器不能判断当前的输入输出指针是否相关或者重叠，因此只能做最保守的估计，通常解决方法是使用指令给编译器足够的信息输入，即在输入输出的指针上增加 restrict 关键字，表示两者不相关，举个例子

```
void loop_carrier_depency_bound(int  *input, int  *output, int count)
{
   int i;
   for (i=0; i < count; i++)
   {
      output[i] = input[i] + 1;
   }
   return;
}
```

  在没有 restrict 关键字修饰输入输出指针时，由于编译器不能判断 input 和 output 在内存中是否重叠，因此需要做最坏的估计，而增加了 restrict 关键字表明 input 和 output 在内存中不重叠以后，性能得到了迅速提升。

- Unpartitioned Resource Bound 和 Partitioned Resource Bound

  这两者的区别是前者等同于不区分 A,B 侧资源时单次循环所需的最小 Cycle 数，后者是相当于区分 A,B 侧资源时所需的 Cycle 数，通常关注后者即可，表示单次循环受到某个单元的资源限制而得出的所需 Cycle 数。

- Resource Partition 表

  在该表中可以清晰的看到 A,B 两侧所有 DSP 内核资源的划分，可以很方便的看到优化的瓶颈具体是在哪个单元上，考虑用其它空闲的单元来替代受限的单元，比如用移位操作（S 单元）代替乘法（M 单元），也可以看到 A,B 两侧单元使用是否均衡，来判断是否需要做循环展开（UNROLL）来获得更平衡的资源划分。

2.3 给予编译器优化信息

根据上述编译器反馈信息，结合用户对自己代码的了解，可以通过指令给予编译器更多的优化信息以帮助编译器更好的优化代码。

- MUST_ITERATE

  使用方法是在循环体上面添加语句

  #pragma MUST_ITERATE(a, b, c);

  其中

  a – 最小循环次数

  b – 最大循环次数

  c – 循环次数的公约数

  这些参数可以根据需要填写，如果参数未知则可以留空

  如 #pragma MUST_ITERATE(16, , 8);

- 使用_nassert(<rules>)断言

  使用断言可以给编译器提供一些先验信息如地址的对齐

  short *x;

  _nassert((int) x % 8 == 0);

- Restrict 关键字

  前面已经提到 restrict 关键字用于指示函数输入输出在内存中的相关性，提前告知编译器不相关性可以有效改善代码效率，使用 restrict 指令既可以针对指针，也可以用于数组，如

  myfunc(type1 input[restrict], type2 *restrict output)

- Unroll 展开循环

  一般在编译器反馈信息 A,B 侧资源使用不平衡，或者希望编译器使用 SIMD（单指令多数据）指令时，可以使用 Unroll 指令强制循环展开，由此可能得到性能的提升，但付出的代价则是代码尺寸的变大。一般在循环体之前使用 Unroll 指令

  #pragma UNROLL(# of times to unroll);

使用上述方法对 2.2 节的函数进行优化

```
int dotp_c(short * restrict a, short * restrict b, int count)
{
   int i;
   int sum = 0;
```

```
   _nassert((int) a % 8 == 0);
   _nassert((int) b % 8 == 0);
   #pragma MUST_ITERATE(8, 400, 8);
   #pragma UNROLL(4)
   for (i=0; i < count; i++)
   {
       sum = sum + a[i] * b[i];
   }
   return(sum);
}
```

得到的结果为

```
;*     Loop source line              : 10
;*     Loop opening brace source line   : 11
;*     Loop closing brace source line   : 13
;*     Loop Unroll Multiple          : 4x
;*     Known Minimum Trip Count        : 2
;*     Known Maximum Trip Count        : 100
;*     Known Max Trip Count Factor     : 2
;*     Loop Carried Dependency Bound(^) : 0
;*     Unpartitioned Resource Bound    : 1
;*     Partitioned Resource Bound(*)   : 1
;*     Resource Partition:
;*                               A-side   B-side
;*     .L units                     0        0
;*     .S units                     0        0
;*     .D units                     1*       1*
;*     .M units                     1*       1*
;*     .X cross paths               1*       1*
;*     .T address paths             1*       1*
;*     Long read paths              0        0
;*     Long write paths             0        0
;*     Logical  ops (.LS)           0        0     (.L or .S unit)
;*     Addition ops (.LSD)          1        1     (.L or .S or .D unit)
;*     Bound(.L .S .LS)             0        0
;*     Bound(.L .S .D .LS .LSD)     1*       1*
;*
;*     Searching for software pipeline schedule at ...
;*       ii = 1  Schedule found with 10 iterations in parallel
;*     Done
;*
;*     Loop will be splooped
;*     Collapsed epilog stages       : 0
;*     Collapsed prolog stages       : 0
;*     Minimum required memory pad   : 0 bytes
;*
;*     Minimum safe trip count       : 1 (after unrolling)
```

通过 UNROLL(4)指令，可以充分使用 DSP 资源在一个 Cycle 里完成 4 个数据的乘加运算，效率比不加任何信息提高了 4 倍。

2.4 使用内联指令

大部分情况下，通过 2.3 节的编译器信息输入，用户就可以得到充分优化的 DSP 代码，但是某些复杂条件下编译器可能不能完美的使用 DSP 的指令对代码进行优化，此时用户可以考虑采用内联指令的方式对 C 代码进一步优化，此时需要用户对 DSP 的内联指令有相当程度的了解并能够熟练的嵌入到自己的算法中去，比如上面的例子用 DSP 自带的点乘运算进行优化后，代码变成

```
for (i = 0; i < len; i += 4)
{
   a3_a2 = _hill(_amemd8_const(&a[i]));
   a1_a0 = _loll(_amemd8_const(&a[i]));

   b3_b2 = _hill(_amemd8_const(&b[i]));
   b1_b0 = _loll(_amemd8_const(&b[i]));

/*Perform dot–products on pairs of elements, totaling the results in the accumulator. */
   sum_high += _dotp2(a3_a2, b3_b2);

   sum_low  += _dotp2(a1_a0, b1_b0);
}
```

使用 DSP 内联指令需要包含 c6x.h 头文件，具体指令说明应参考 C66x CPU 指令手册。

3 DSP 内存系统优化

通过第二章的代码优化，用户可以充分利用 TI 提供的编译器优化自己的代码效率，但是有些情况下，仅仅提高代码效率是不够的，当用户将优化过的代码在硬件上实测时常常会发现运行效率与编译器输出的 Cycle 数相差比较大，此时应先排除中断死锁，任务抢占等干扰因素，如果仍然与预想效率有比较大的出入，则需要考虑内存系统效率的优化。

3.1 使用 EDMA 提高吞吐率

一般在开启 DSP 内部 L2 Cache 以后，系统数据吞吐率能够得到很大的提升，但是在实际应用中，当外部数据量非常大，即使开启了 Cache，但是因为 DDR3 的读取延时较大，很容易成为拖慢算法效率的瓶颈，下表反应了在开启与关闭 Cache 时在 AM5728 EVM 上使用 DSP 进行数据拷贝的速率以及 EDMA 拷贝数据的速率对比，可以看到使用 EDMA 进行外部和内部数据交换时的效率比 DSP 核的效率要高 得多，因此在计算数据量比较大且位于 DDR 中时可以考虑使用 EDMA 来进行数据搬移的工作。

Table 4. AM57x C66x 内存拷贝效率

MB/s	L2->L2	L2->OCM	L2->Other L2	DDR->L2	L2->DDR
Enable Cache	1908	1328	1014	412	1116
Disable Cache	1908	1034	558	59	602
EDMA	1589	2705	2698	2698	2698

回顾 AM57x DSP 子系统的框图，在每个 DSP 子系统中都内置了一个 EDMA 模块，因此使用该 EDMA 模块进行外部数据如 DDR 或 OCM 内存与 DSP 内部 L2 之间的数据搬移十分便利，且不会占用系统 EDMA 的通道资源。而且该 EDMA 模块在每个 DSP 模块内部的寄存器地址都完全一样，这样方便了用户能够将同一份代码使用到不同的 DSP 子系统上。

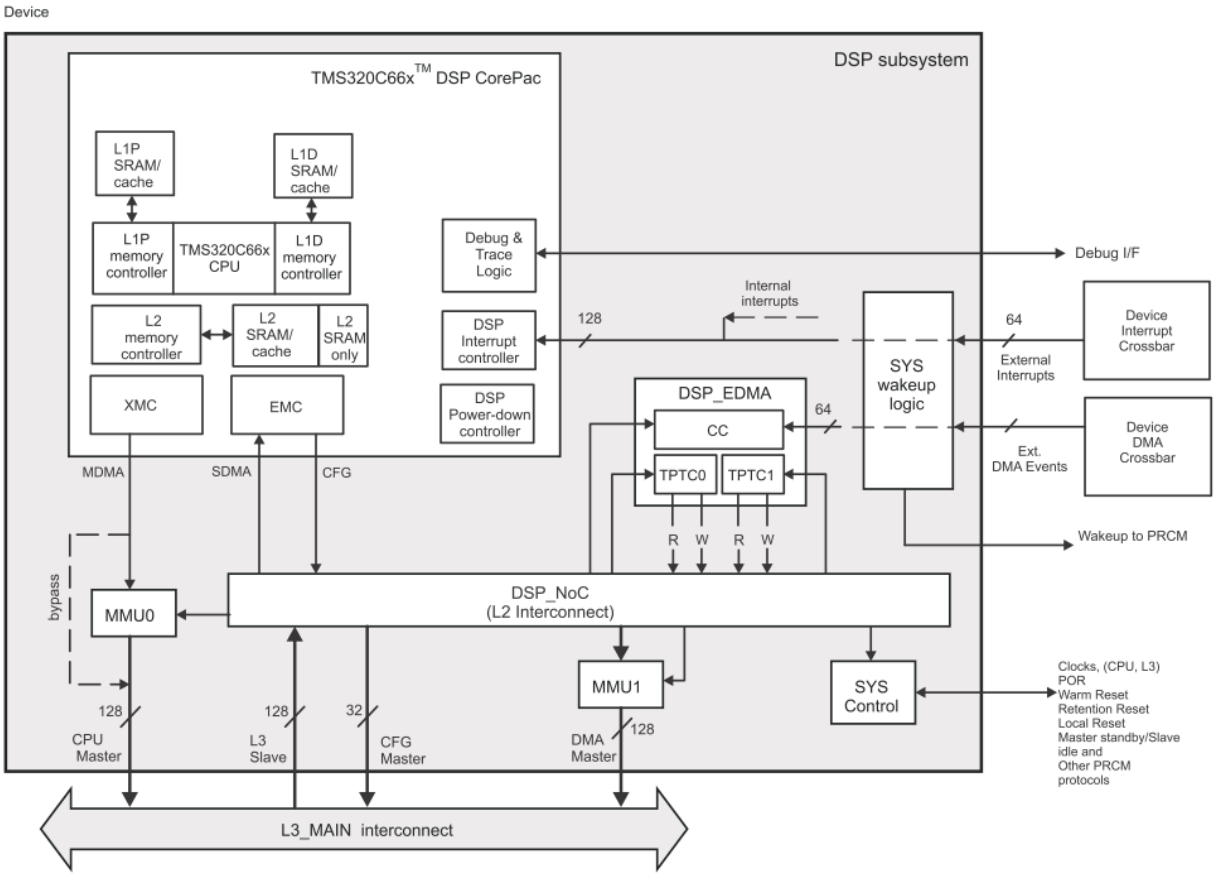

Figure 1. AM57x DSP 子系统

下面使用一个实例来说明 EDMA 对于大数据量计算效率的提升。

需要计算的是一个双精度复数矩阵与其转置的乘积，该矩阵大小为 8192×100，需要的存储空间为 8192×100×8(double) ×2(complex)=12MB，而 AM57x DSP 的内部 L2 内存大小为 288KB，因此只能将原始数据放在 DDR 中。

首先对代码进行优化提高 DSP 的计算效率，对于该矩阵乘法，最终求得的矩阵的每个元素为初始矩阵的不同行的乘积，且为对称阵，优化后的矢量乘法的代码如下

```
void ComplexVectorMul(      double * restrict pdDatSignalI,
                            double * restrict pdDatSignalQ,
                            double * restrict pdDatFilterI,
                            double * restrict pdDatFilterQ,
                            int       vecSize,
                            double * restrict pdResultI0,
```

```
                                    double * restrict pdResultQ0,
                                    double * restrict pdResultI1,
                                    double * restrict pdResultQ1)
{
                int j;
                double dTempI_A = 0.0;
                double dTempI_B = 0.0;
                double dTempQ_A = 0.0;
                double dTempQ_B = 0.0;
                double filterI ,filterQ;
                double signalI ,signalQ;

                _nassert(vecSize % 8 == 0 );
                _nassert(vecSize > 0);
                _nassert((int)pdDatSignalI % 8 == 0);
                _nassert((int)pdDatSignalQ % 8 == 0);
                _nassert((int)pdDatFilterI % 8 == 0);
                _nassert((int)pdDatFilterQ % 8 == 0);

#pragma MUST_ITERATE(128,8192,128);
#pragma UNROLL(2);
                for(j = 0; j < vecSize; j++)
                {
                                    filterI = pdDatFilterI[j];
                                    filterQ = pdDatFilterQ[j];
                                    signalI = pdDatSignalI[j];
                                    signalQ = pdDatSignalQ[j];
                                    dTempI_A += filterI * signalI;
                                    dTempQ_A += filterI * signalQ;
                                    dTempI_B += filterQ * signalQ;
                                    dTempQ_B += filterQ * signalI;
                }
                pdResultI0[0] =  dTempI_A + dTempI_B;
                pdResultQ0[0] =  dTempQ_A - dTempQ_B;
                pdResultI1[0] =  dTempI_A + dTempI_B;
                pdResultQ1[0] =  dTempQ_B - dTempQ_A;

                return;
}
```

对整个矩阵进行计算并统计计算时间，为了保证数据访问效率，此处采用 128KB 的 L2 Cache 配置。

```
printf("**********Tests with DSP ONLY Matrix Multiplication**********\n");
tscl= TSCL;
tsch= TSCH;
startTime = _itoll(tsch, tscl);

for(i=0; i<COLUMN_SIZE; i++)
{
                for(j=i; j<COLUMN_SIZE; j++)
                {
                                    ComplexVectorMul(&matrixI[i*ROW_SIZE] ,&matrixQ[i*ROW_SIZE],
                                                        &matrixI[j*ROW_SIZE] ,&matrixQ[j*ROW_SIZE] ,
                                                        ROW_SIZE,
                                                        &verifyI[i*COLUMN_SIZE+j] ,&verifyQ[i*COLUMN_SIZE+j],
```

```
                                        &verifyI[j*COLUMN_SIZE+i] ,&verifyQ[j*COLUMN_SIZE+i]);
                }
}

endTime = _itoll(TSCH, TSCL);
totalTime = endTime-startTime;
```

通过在 AM5728 EVM 上仿真可得计算 8192×100 大小矩阵与其转置矩阵的时间为 1300M Cycles。

通过分析乘加次数，每次 8192 点的双精度复数向量乘法需要 4 次乘法，而 C66x 核每个 Cycle 可以计算 2 次双精度乘法，因此理论上每次向量乘法需要的 Cycle 数为 8192×4/2=16384，完成 100 行矩阵的 Cycle 数理论下限是 16384×100×100/2=81M Cycles，与实际的结果相差很大，通过分析生成的汇编文件，可以发现编译器已经将代码优化做到极致，因此内存开销是引起这个差异的原因所在。

由前面带宽测试的数据表可知在大数据量情况下 EDMA 可以提供更大的数据传输带宽，将数据拷贝到片内也能够更好的提升数据计算效率，因此，我们在循环中采用 EDMA 拷贝与 DSP 计算并行的流程，在 DSP L2 内存上使用 64KB 作为计算缓冲区，为了保证其它算法模块的效率，L2 Cache 仍然配置成 128KB 不变。

因为 EDMA 拷贝需要与 DSP 计算并行执行以提升整体效率，此处将 64KB 的内存分成 4 部分，分别作为两个复向量矩阵的 I 路和 Q 路的数据缓冲区，同时，在每 16KB 的单个缓冲区内，将其分成两部分，使用 ping-pong 的形式以使 EDMA 和 DSP 可以并行操作。具体代码如下。

```
void EdmaComplexVectorMul(      double * restrict pdDatSignalI,
                                double * restrict pdDatSignalQ,
                                double * restrict pdDatFilterI,
                                double * restrict pdDatFilterQ,
                                int      vecSize,
                                double * restrict pdResultI0,
                                double * restrict pdResultQ0,
                                double * restrict pdResultI1,
                                double * restrict pdResultQ1)
{
//vector is double(8) IQ(2). Buffer splitted to 4 areas(2-way I and Q).
unsigned int splitTimes = vecSize*8*2/(MATRIX_BUFFER_SIZE/4);
unsigned int numSampleOnetime = vecSize/splitTimes;
unsigned int szBuf = numSampleOnetime*8;
double resultI0, resultQ0, resultI1, resultQ1;
int j;
unsigned int ping, pong;
double dTempI_A = 0.0;
double dTempI_B = 0.0;
double dTempQ_A = 0.0;
double dTempQ_B = 0.0;



EdmaCopyStart(pdDatSignalI, bufDataI, szBuf, 1, szBuf,  0);
EdmaCopyStart(pdDatSignalQ, bufDataQ, szBuf, 1, szBuf,  1);
EdmaCopyStart(pdDatFilterI, bufFilterI, szBuf, 1, szBuf,  2);
EdmaCopyStart(pdDatFilterQ, bufFilterQ, szBuf, 1, szBuf,  3);
```

```
for(j = 0; j < splitTimes; j++)
{
    EdmaCopyWait(0);
    EdmaCopyWait(1);
    EdmaCopyWait(2);
   EdmaCopyWait(3);

    ping = (j+1)%2;
    pong = j%2;

    EdmaCopyStart(pdDatSignalI+(j+1)*numSampleOnetime, bufDataI+ping*numSampleOnetime, szBuf, 1, szBuf, 0);
    EdmaCopyStart(pdDatSignalQ+(j+1)*numSampleOnetime, bufDataQ+ping*numSampleOnetime, szBuf, 1, szBuf, 1);
    EdmaCopyStart(pdDatFilterI+(j+1)*numSampleOnetime, bufFilterI+ping*numSampleOnetime, szBuf, 1, szBuf, 2);
    EdmaCopyStart(pdDatFilterQ+(j+1)*numSampleOnetime, bufFilterQ+ping*numSampleOnetime, szBuf, 1, szBuf, 3);

    ComplexVectorMul(&bufDataI[pong*numSampleOnetime] ,
                        &bufDataQ[pong*numSampleOnetime] ,&bufFilterI[pong*numSampleOnetime] ,
                        &bufFilterQ[pong*numSampleOnetime] , numSampleOnetime,
                        &resultI0,&resultQ0,&resultI1,&resultQ1);

    dTempI_A +=  resultI0;
    dTempQ_A +=  resultQ0;
    dTempI_B +=  resultI1;
    dTempQ_B +=  resultQ1;
}

pdResultI0[0] =  dTempI_A;
pdResultQ0[0] =  dTempQ_A;
pdResultI1[0] =  dTempI_B;
pdResultQ1[0] =  dTempQ_B;

return;
}
```

采用 EDMA 优化后，我们得到的矩阵运算的整体时间约为 700M Cycles，比仅使用 Cache 的方式要好很多，但与理论值仍有一定的差距，因此要继续寻找可能的优化方式。

3.2 优化 Cache 效率

对于 C66x DSP 核而言，其内部内存系统分成两级，L1 和 L2。其中 L1 又分成 L1D 和 L1P，分别对应数据和程序存储，其大小都是 32KB，C66x DSP 核直接访问 L1P 获取可执行语句或访问 L1D 获取需要计算的数据。大部分情况下为了提高程序的整体运行效率，L1P 和 L1D 都会配置成 Cache 而非 RAM，在配置成 Cache 时，L1P 和 L1D 没有物理上的可访问地址，因而是程序员不可见的空间。在一些特殊的场景下，为了充分提高某些代码运行或数据访问的效率，用户也可以将一部分 L1 设置成 RAM 而余下的部分配置成 Cache，这样做可能会引起其它代码运行效率的下降，此时应该通过测试来决定 L1P 或 L1D RAM 空间的大小。

L2 内存空间相比 L1 更大，在 AM57X 系列中，DSP 子系统的 L2 大小配置是 288KB。L2 既可以储存数据也可以储存代码，而且也可以配置成 Cache 或者 RAM，AM57X DSP 子系统最大支持 256KB 的 L2 Cache，当然用户也可以将其配置成 128KB，64KB 等等以获得更多的 L2 RAM 空间，大的 Cache 空间可以提高 DDR3 中代码以及数据访问的效率，但是用户能够使用的 L2 RAM 也相对减少，一些关键的代码和数据可能无法配置到内部 RAM 中以提高运行效率，因此 RAM 空间的大小和效率需要用户根据自己的使用场景不断的测试并优化。

因为 DSP 核的 L1D Cache 为 2-Way，每一路大小为 16KB，在上述示例中，使用的数据缓冲区是 4 个 16KB，存放于 L2 内存，代码同时使用这 4 块缓冲区进行计算时会产生频繁的 L1D Cache 替换，这个问题可能是影响整体计算效率的原因，解决方法就是将这 4 块缓冲区错开一个 L1D Cache Line 的大小，使其不会频繁发生 Cache 替换。

```
//Offset is set to L1D cache line size 64 bytes.
#if MATRIX_BUFFER_SIZE==64*1024
static double *bufDataI =       (double *)(0x00800000);
static double *bufDataQ =       (double *)(0x00804040);
static double *bufFilterI =     (double *)(0x00808080);
static double *bufFilterQ =     (double *)(0x0080C0C0);
#endif
```

修改代码后在 EVM 上进行验证，矩阵计算的 Cycle 数降低到 313M，考虑到片内 L2 存储器的存取延时，这个结果是可以被用户所接受的。

比较内存优化的结果，最终的结果如下，相对于没有做内存优化之前的结果，计算效率上有超过 4 倍的提升。

Table 5.　优化内存读写的效率对比

	直接使用 L2 Cache	使用 EDMA 拷贝	使用 EDMA 并优化 Cache
计算 Cycle 数 （M）	1300	700	313

4 小结

本文主要介绍了 AM57x 系列处理器上 DSP 核代码以及内存访问的优化方法，从文中提供的示例可以看到，使用编译器的输出结果可以帮助用户对 C 代码进行优化，充分利用 DSP 的计算单元；同时，利用 AM57x DSP 子系统里提供的 EDMA 模块，优化存储结构等方法能够有效的提升算法在 DSP 上的运行效率，这些方法对于需要在 AM57x 处理器上进行大数据量处理如图像和视频处理，矩阵或向量计算等都有良好效果及实际使用价值。

参考文献

1. *AM572x Sitara™ Processors Silicon Revision 2.0 (SPRS953)*
2. *AM572x Sitara Processor Technical Reference Manual (SPRUHZ6)*
3. *Optimizing Loops on the C66x DSP (SPRABG7)*
4. *TMS320C66x DSP CPU and Instruction Set Reference Guide (SPRUGH7)*
5. *TMS320C6000 Optimizing Compiler v7.4 User's Guide(SPRU187)*
6. *TMS320C66x DSP Cache User Guide(SPRUGY8)*

有关 TI 设计信息和资源的重要通知

德州仪器 (TI) 公司提供的技术、应用或其他设计建议、服务或信息，包括但不限于与评估模块有关的参考设计和材料（总称“TI 资源”），旨在帮助设计人员开发整合了 TI 产品的 应用； 如果您（个人，或如果是代表贵公司，则为贵公司）以任何方式下载、访问或使用了任何特定的 TI 资源，即表示贵方同意仅为该等目标，按照本通知的条款进行使用。

TI 所提供的 TI 资源，并未扩大或以其他方式修改 TI 对 TI 产品的公开适用的质保及质保免责声明；也未导致 TI 承担任何额外的义务或责任。TI 有权对其 TI 资源进行纠正、增强、改进和其他修改。

您理解并同意，在设计应用时应自行实施独立的分析、评价和 判断， 且应全权负责并确保 应用的安全性， 以及您的 应用 （包括应用中使用的所有 TI 产品）） 应符合所有适用的法律法规及其他相关要求。你就您的 应用声明，您具备制订和实施下列保障措施所需的一切必要专业知识，能够 (1) 预见故障的危险后果，(2) 监视故障及其后果，以及 (3) 降低可能导致危险的故障几率并采取适当措施。您同意，在使用或分发包含 TI 产品的任何 应用前， 您将彻底测试该等 应用 和该等应用所用 TI 产品的 功能而设计。除特定 TI 资源的公开文档中明确列出的测试外，TI 未进行任何其他测试。

您只有在为开发包含该等 TI 资源所列 TI 产品的 应用时， 才被授权使用、复制和修改任何相关单项 TI 资源。但并未依据禁止反言原则或其他法理授予您任何TI知识产权的任何其他明示或默示的许可，也未授予您 TI 或第三方的任何技术或知识产权的许可，该等产权包括但不限于任何专利权、版权、屏蔽作品权或与使用TI产品或服务的任何整合、机器制作、流程相关的其他知识产权。涉及或参考了第三方产品或服务的信息不构成使用此类产品或服务的许可或与其相关的保证或认可。使用 TI 资源可能需要您向第三方获得对该等第三方专利或其他知识产权的许可。

TI 资源系“按原样”提供。TI 兹免除对 TI 资源及其使用作出所有其他明确或默认的保证或陈述，包括但不限于对准确性或完整性、产权保证、无屡发故障保证，以及适销性、适合特定用途和不侵犯任何第三方知识产权的任何默认保证。

TI 不负责任何申索，包括但不限于因组合产品所致或与之有关的申索，也不为您辩护或赔偿，即使该等产品组合已列于 TI 资源或其他地方。对因 TI 资源或其使用引起或与之有关的任何实际的、直接的、特殊的、附带的、间接的、惩罚性的、偶发的、从属或惩戒性损害赔偿，不管 TI 是否获悉可能会产生上述损害赔偿，TI 概不负责。

您同意向 TI 及其代表全额赔偿因您不遵守本通知条款和条件而引起的任何损害、费用、损失和/或责任。

本通知适用于 TI 资源。另有其他条款适用于某些类型的材料、TI 产品和服务的使用和采购。这些条款包括但不限于适用于 TI 的半导体产品 (http://www.ti.com/sc/docs/stdterms.htm)、评估模块和样品 (http://www.ti.com/sc/docs/sampterms.htm) 的标准条款。

邮寄地址：上海市浦东新区世纪大道 1568 号中建大厦 32 楼，邮政编码：200122