Application Report
SPRABG7—November 2010

Optimizing Loops on the C66x DSP

High-Performance and Multicore Processors

Abstract

The TMS320C6000 compiler automatically performs a great deal of performance-related tuning. This compiler-driven optimization usually suffices. For the occasional cases where additional CPU performance is needed, this application report presents strategies and examples for improving performance of C/C++ applications. Memory-related performance improvements (such as background DMA transfers or cache usage) are outside the scope of this report. This report provides the C66x addition to SPRA666, which describes the general tuning techniques for C6000 family.

Contents

List of Tables

List of Figures

1 Introduction

This application report provides optimization techniques specifically for the C66x architecture. It supplements the techniques in [4] for optimizing performance on C6000 architectures including C66x.

The remainder of this document is structured as follows:

- Section 2 introduces the capacities and improvements from C64x+ to C66x architecture. These are necessary for understanding the tuning techniques presented later in this document. This section also introduces the new 128-bit data type for C66x.
- Section 3 introduces the C66x specific features and related optimization techniques, emphasizing on floating-point considerations and intrinsics selections for complex matrix/vector operations.
- Section 4 teaches advanced techniques to deal with register pressure and register living too long problems.

Unless otherwise specified, examples in this application report use C6000 compiler version 7.2 alpha (CCStudio version 4.1) and target the C66x.

2 Overview of the TMS320C6600

TMS320C66x's Instruction Set Architecture is an enhancement of the TMS320C674x DSP. As its predecessor, it is an advanced VLIW architecture with eight functional units (two multipliers unit and six arithmetic units) that operate in parallel. Key additions to the ISA include:

- 4× Multiply Accumulate improvement
- Improvement of the floating point arithmetic
- Enhancement of the vector processing capability for floating point and fixed point
- Addition of domain specific instructions for complex arithmetic and matrix operations

2.1 Key Additions

Multiply/Accumulate Increased 400 Percent—C66x ISA significantly improves the maximum number of multiply operations that can be executed per cycle. The core can now execute up to 32 (16x16-bit) multiplications per cycle or up to 8 single precision floating-point multiplications per cycle.

Floating Point Support—C66x ISA enhances and optimizes the TMS320C674x DSP, which combines the capabilities of the floating point TMS320C67x DSP and the fixed point TMS320C64x+ DSP. C66x ISA implements native support for IEEE 754 single precision and double-precision instructions.

Floating point improvements include:

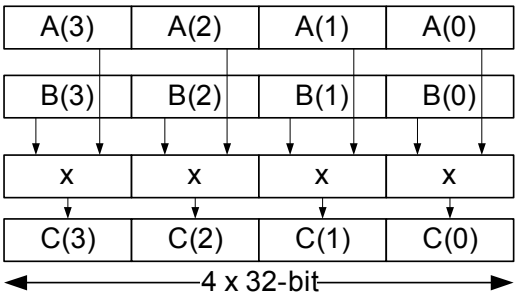
- Fast implementation of all basic floating point operations,
- SIMD support for floating point operations,
- Single precision complex multiply
- Additional resource flexibility (e.g. the INT to/from SP conversion operations can now be executed on .L and .S units)

The improved performance of the floating point processing capability and the ability to execute both fixed-point and floating-point math adds a new dimension to the capabilities available to the DSP algorithm programmers. By selectively using the advanced fixed-point instruction set and the floating-point instruction set, the programmer can achieve significantly higher performance for their algorithm in terms of data precision and optimized cycle count.

Vector Processing—The TMS320C64x+/C674x DSPs support 2-way SIMD operations for 16-bit data and 4-way SIMD operations for 8-bit data. On C66x ISA, the vector processing capability is improved by extending the width of the SIMD instructions. C66x instructions can now execute instructions that operate on 128-bit vectors. For example, the QMPY32 instruction is able to perform the element-by-element multiplication between two vectors of four 32-bit data, each.

Figure 1 shows the quad-vector 32-bit multiply instruction architecture available on the C66x.

Figure 1 QMPY32 - Example of Vector Instruction



SIMD Support—Improved vector processing capability (each instruction can process multiple data in parallel) combined with the natural instruction level parallelism of C6000 architecture (e.g execution of up to 8 instructions per cycle) results in a very high level of parallelism that can be exploited by DSP programmers through the use of TI's optimized C/C++ compiler.

Complex Arithmetic and Matrix Operations—Communication signal processing requires extensive use of complex arithmetic functions and linear algebra (matrix computation). C66x ISA includes a set of specific instructions to handle complex arithmetic and matrix operations.

For example, C66x can now perform up to two multiplications of a [1×2] complex vector by a [2×2] complex matrix per cycle and the core supports a set of instructions that operates either on scalar or vector numbers to perform complex multiplications, conjugate of a complex number, multiplication by the conjugate, and rotations of complex numbers.

Table 1 shows a comparison of the raw performance between the C64x+ and C66x ISA. See the *C66x CPU and ISA Reference Guide* [2] for detailed information on C66x architecture and ISA.

Table 1 Raw Performance Comparison Between C64x+ and C66x

	C64x+	C674x	C66x
Fixed point 16x16 MACs per cycle	8	8	32
Fixed point 32x32 MACs per cycle	2	2	8
Floating point single precision MACs per cycle	n/a	2	8
Arithmetic floating point operations per cycle	n/a	6[1]	16[2]
Arithmetic floating point operations per cycle	n/a	6[3]	16[4]
Load/store width	2 × 64-bit	2 × 64-bit	2 × 64-bit
Vector size (SIMD capability)	32-bit (2 × 16-bit, 4×-8bits)	32-bit (2 × 16-bit, 4×-8bits)	128-bit (4 × 32-bit, 4 × 16-bit, 4×-8bits)

1. One operation per .L, .S, .M units for each side (A and B)
2. 2-way SIMD on .L and .S units (e.g. 8 SP operations for A and B) and 4 SP multiply on one .M unit (e.g 8 SP operations for A and B).
3. One operation per .L, .S, .M units for each side (A and B)
4. 2-way SIMD on .L and .S units (e.g. 8 SP operations for A and B) and 4 SP multiply on one .M unit (e.g 8 SP operations for A and B).

2.2 C66x Floating-Point Overview

In addition to the fixed-point enhancement over previous C64x+ architecture in 32-bit and 16-bit fixed-point operations, C66x also offers both single-precision and double-precision floating-point capability at the same clock speed of fixed-point operation, and sharing the same data path, register file and operation units. The C66x not only inherited the floating-point instruction set from previous C67x+ devices, it also added many SIMD instructions to do addition, subtraction, and complex multiplication more effectively. Furthermore, there is a set of fast data format conversion instructions that can make mixed fixed-floating-point programming very efficient. By selectively using the advanced fixed-point instruction set and the floating-point instruction set, the developer can achieve significantly higher performance for their algorithm in shorter development cycles.

In Section 3, consideration and optimization for floating-point is discussed in depth.

2.3 128-Bit Data Type

__x128_t is a container type for storing 128-bits of data and its use is necessary when performing certain SIMD operations on C6600. When using the __x128_t container type, you must include c6x.h. This type may be used only when compiling for C6600 (use the compiler option -mv6600). Also, note the leading double underscore.

This type can be used to define objects that can be used with certain C6600 intrinsics. The object can be filled and manipulated using various intrinsics. The type is not a full-fledged built-in type (like long long), and so various native C operations are not allowed. Think of this type as a struct with private members and special manipulation functions.

When the compiler puts a __x128_t object in the register file, the __x128_t object takes four registers (a register *quad*).

Objects of this type are aligned to a 128-bit boundary in memory.

The following operations are supported:

- Declare a __x128_t global object (e.g. __x128_t a;). By default, it will be put in the .far section.
- Declare a __x128_t local object (e.g. __x128_t a;). It will be put on the stack.
- Declare a __x128_t global/local pointer (e.g. __x128_t *a;).
- Declare an array of __x128_t objects (e.g. __x128_t a[10];).
- Declare a __x128_t type as a member of a struct, class, or union.
- Assign a __x128_t object to another __x128_t object.
- Pass a __x128_t object to a function (including variadic argument functions). (Pass by value.)
- Return a __x128_t object from a function.
- Use 128-bit manipulation intrinsics to set and extract contents (see list below)
- The following operations are not supported:
- Native-type operations on __x128_t objects, such as +, -, *, etc.
- Cast an object to a __x128_t type.
- Access the *elements* of a __x128_t using array or struct notation.
- Pass a __x128_t object to I/O functions like printf. Instead, extract the values from the __x128_t object by using appropriate intrinsics.

3 C66x Floating-Point and Vector/Matrix Operations and Optimizations

In this section, C66x specific features are considered in detail, including floating-point and advanced complex matrix operations.

For floating-point, the following aspects are considered:

- When to consider using floating-point to better optimize the implementation.
- Elimination of scaling and rounding used with fixed point implementation to reduce cycle cost.
- Use of inverse and inverse square-root instructions.
- Fast data format conversions.

For complex matrix operations the following techniques are explored:

- Effective complex number operations
- Effective advanced matrix and vector instructions for matrix operations
- Matrix inversion

3.1 Floating-Point Arithmetic

C66x floating-point support on C66x adds a new dimension to the capabilities available to the algorithm developer. Almost all advanced algorithms are developed originally in floating-point with extensive performance study. In the area of wireless applications, most of originally verified algorithms are in single-precision floating-point, either in generic C, or Matlab. The focus is on single-precision floating-point arithmetic in the following subsections, although double-precision floating-point is supported on C66x as well for extended precision. Floating-point refers to single-precision in the following text, if not specified otherwise.

See Section 3.3 of the *Hand-Tuning Loops and Control Code on the TMS320C6000* application report [4] for details on IEEE floating-point format definitions.

3.1.1 C66x Floating-Point Advantages

C66x floating-point offers the following advantages:

- Fast algorithm implementations due to no need for additional steps to convert to fixed-point with iterations for Q value tuning, optimization, and performance tests to trade cycle counts against precision requirements. If the algorithm is already tested in generic floating-point C code or Matlab scripts, floating-point implementation on DSP will achieve the same performance.
- Saves cycles from dynamic scaling and Q value adjustments for fixed-point.
- Fast division instructions.
- Combined higher dynamic range and fixed 24-bit precision of floating-point provides the opportunity for less complex and more efficient algorithm to achieve stable results, compared to 32-bit fixed-point representation.
- Fast data format conversion instruction offers the programmer effective ways to program in the mixed fixed-floating-point style to achieve high efficiency and high performance at the same time.

C66x floating-point arithmetic capacities include the following:

- Same number of single-precision operations per cycle as C64x+ 16-bit integer operations per cycle.
- Capability of C66x single-precision floating-point multiplication and subtraction/addition is:
  - Same capability of 16-bit integer operation in C64x+ core
  - Same as 32-bit integer operations in C66x core
  - 4 times capability of 32-bit integer operation in C64x+ core
- Multiplication:
  - Eight single-precision multiplications per cycle: CMPYSP and QMPYSP can calculate four pairs of single–precision multiples per .M unit per cycle.
- Addition/subtraction:
  - Eight single-precision addition/subtraction per cycle: DADDSP and DSUBSP can add/sub 2 floats and they can be executed on both .L and .S units.
- Conversion between floating-point and integer:
  - Eight single-precision to integer conversions, or 8 integers to single-precision conversions per cycle: DSPINT, DSPINTH, DINTHSP, and DSPINTH convert two floats to two integers per cycle and it can be executed on both .L and .S units.
- Division:
  - Two 1/x and 1/sqrt(x) calculations per cycle. Fully pipelinable interpolation steps maybe needed for higher precision.

3.1.2 Example of establishing a floating-point baseline

In the following sections, an example shows floating-point basic arithmetic and optimization techniques. This example takes an input complex array a, and find |a[i]| and $e^{jangle(a[i])}$ for all the elements in the array. The generic C implementation is listed below:

```
void example1_gc(cplxf_t *a, cplxf_t *ejalpha, float *abs_a, int n)
{
       int i;
    float a_sqr, oneOverAbs_a;

       for  ( i = 0; i < n; i++)
       {
        a_sqr          =a[i].real * a[i].real + a[i].imag * a[i].imag;
        oneOverAbs_a       =1.f/(float)sqrt(a_sqr);

        abs_a[i]       =  a_sqr * oneOverAbs_a;
        ejalpha[i].real     =a[i].real * oneOverAbs_a;
        ejalpha[i].imag     =a[i].imag * oneOverAbs_a;
       }
}
```

The test vector is generated using the rand() function for both real and imaginary parts and subtracting 0x4000 to make the random number range between [-16384 16383]. For C66x (mixed-fixed-floating-point) code, the API from the generic C code is used.

One implementation of fixed-point C64x+ code is used as a comparison. The API for fixed-point code will change with input directly converted to fixed-point in 16-bit I/Q format, and *ejalpha* represented in Q15 format.

To check the results, compute the normalized error magnitude:

$$|output_i - ref_i| / |ref_i|$$

for *abs_a[i]* and *ejalpha[i]* of each implementation against generic C outputs. For fixed point implementation, *abs_a[i]* is converted directly to floating-point, while *ejalpha[i]* is converted to float, divided by 32768 because the number is represented in Q15 format, and then calculate the error magnitude compare to generic C output of *abs_a[i]* and *ejalpha[i]*. Maximum of error magnitude is collected for the input test vector.

Cycle measurements and implementation errors for each optimization step is summarized.

Tools used:

- CCSv4.1.0
- Compiler 7.2.0A10232
  - General compiler options: -o3 –k –os. For C66x, using –mv6600.
- TCI6482 Cycle accurate Simulator Little Endian
- Nyquist Device Cycle Approximate Simulator, Little Endian

Here, Little Endian only is used, and code can be easily expanded to Big Endian case.

For cycle measurement, the cycle number is measured by calling the function twice and measure the cycles for the second call. The first call of the function will access all the data and program causing it to be loaded into the L1 cache. And the cycle measurement for the second call is defined as *warm cache cycles* or *warm cycles*. TSCL is used for cycle measurements.

3.1.3 A Quick Start

First, directly compile the generic C code on the C64x+ platform, which is fixed-point only, and on the C66x platform, which is mixed fixed-floating-point. Table 2 shows the benchmark comparison:

Table 2 Benchmark Comparison

Generic C Code Directly Compiled For	Warm Cycles (with division)	Warm Cycles (without division)
C64x+ platform	473483	473483
C66x platform	69644	69644

Both versions have the loop disqualified for pipeline because of function calls inside the loop. That is the reason why these two cycle counts are very high. Particularly for direct compilation on C64x+ platform, not only 1/sqrt(x) functions is a call to runtime support library, but all the floating-point arithmetic are function calls using fixed-point to mimic floating-point.

For direct compilation on C66x platform, only 1/sqrt(x) is a function call and other operations are using floating-point instructions offered by C66x. This is why a 6.8× lift ia seen by simply going to the combined fixed-floating-point platform.

If, for example, this routine is called sporadically and 70 k cycles is acceptable for the first phase optimization, it can stop at this point. This offers a quick start and shorter development cycles for this type of optimization problems.

If the number of cycle counts need to be reduced further, the generic C code must be optimized.

For C64x+ platform, the following are needed:

- Have a function that can quickly calculate 1/sqrt(x). This is the ***key*** to reducing the most cycles.
- Do necessary scaling and Q adjustment accordingly.
- Perform wider data read and write, as suggested in Chapter 4.
- Perform SIMD fixed-point complex multiplication.

For the C66x platform, the single-cycle instruction RSQRSP can be used to do 1/sqrt(x). Complex multiplication must be optimized if further cycle reduction is needed.

In the following sections, one optimized implementation of C64x+ is discussed. Then the optimization of C66x floating-point is covered in detail, including the usage of RSQRSP, floating-point SIMD, and fixed-floating-point conversion.

3.1.4 One C64x+ Optimization

First is the 1/sqrt(x) calculation problem in fixed-point. There are many different ways to implement this function. There are well optimized assembly functions for this purpose, but they cannot be used because calling an assembly function disables the loop pipeline. One better choice is to have a look-up-table-based implementation that can be inlined. The one selected for this example is a 16-bit precision 1/sqrt(x) that uses four tables, each with 256 elements.

Note that the fixed-point implementation is considered from the aspects of complexity of the work. This may not be the state-of-art and best-optimized code in this section. How 1/sqrt(x) is implemented is not explained in detail, either. The refinement is left as an exercise for the reader.

```
_nassert(n % 4 == 0);
_nassert((int) a % 8 == 0);
_nassert((int) ejalpha % 8 == 0);
_nassert((int) abs_a % 8 == 0);
#pragma MUST_ITERATE(4,100, 4);
#pragma UNROLL(2);
for  ( i = 0; i < n; i++)
{
  temp1  = _amem4(&a[i]);
     a_sqr =  _dotp2(temp1, temp1);

  /* 1/sqrt(a_sqr) */
     normal  = _norm(a_sqr);
     normal  = normal & 0xFFFFFFFE;
     x_norm  = _sshvl(a_sqr, normal);
     normal  = normal >> 1;
     Index   = _sshvr(_sadd(x_norm,0x800000),24);
     oneOverAbs_a=_mpylir( xcbia[Index], x_norm );
     oneOverAbs_a=_sadd((int)x3sa[Index]<<16,
_sshvr(oneOverAbs_a,ShiftValDifp1a[Index]));
     normal    =15 - ShiftVala[Index] + normal;

     ejbeta_re =_sadd(_sshvl(_mpyhir(temp1, oneOverAbs_a), normal - 1), 0x8000);
     ejbeta_im =_sadd(_sshvl(_mpylir(temp1, oneOverAbs_a), normal - 1), 0x8000);
     _amem4(&ejalpha[i])= _packh2(ejbeta_re, ejbeta_im);
     abs_a[i] = sshvr(_sadd(_mpyhir(oneOverAbs_a, a_sqr), 1<<(15 - normal)),
16-normal) ;
}
```

The loop information is shown below:

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 198
;*      Loop opening brace source line   : 199
;*      Loop closing brace source line   : 218
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 50
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 9
;*      Partitioned Resource Bound(*)    : 9
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     1        1
;*      .S units                     6        6
;*      .D units                     7        6
;*      .M units                     9*       9*
;*      .X cross paths               5        1
;*      .T address paths             7        6
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           8        7     (.L or .S unit)
;*      Addition ops (.LSD)          5        7     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             8        7
;*      Bound(.L .S .D .LS .LSD)     9*       9*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 9  Register is live too long
;*         ii = 9  Did not find schedule
;*         ii = 10 Register is live too long
;*         ii = 10 Register is live too long
;*         ii = 10 Did not find schedule
;*         ii = 11 Schedule found with 5 iterations in parallel
;*      Done
```

As seen in the code listed above, the 1/sqrt(x) function is manually inlined, as highlighted in the code. _nassert(), restrict keyword, and #pragmas were used to inform the compiler of the details on buffer aliasing, alignment, and loop trip count so that its optimization rules could be invoked.

Use _amem4() for load and store.

Use MPYLIR and MPYHIR to do a 16-bit complex number multiplied by a 32-bit real number. And use DOTP2 to calculate power of a 16-bit complex number.

There are also roundings and scaling for *abs(a[i])* and *ejalpha(a[i])* to align the Q value of eh output data correctly.

After all these steps, the cycles of this function are reduced to **746**. But, this is achieved with perhaps hours of coding and tuning the code, even if there is a 1/sqrt(x) function available from previous work.

Table 3 C64x+ Optimization Results

	Cycles	Max abs_a Error	Max epjalpha Error
Direct C64x+ Compilation	473483	N/A	N/A
Direct C66x Compilation	69644	N/A	N/A
Optimized C64x+	613	2.359918e-04	2.765343e-04

3.1.5 Usage of Division Instructions

The C66x has two sets of single-cycle division instructions: RSQRSP for 1/sqrt(x) calculation and RCPSP for 1/x calculation. As the first step of C66x optimization, start with plugging RSQRSP into the code.

The general guideline for compiler-friendly C code was used by adding _nassert(), restrict keyword, and #pragmas to inform the compiler of the known information on the buffer aliasing, alignment, and loop trip count. Note the loop is forced to unroll by multiples of 2.

An important property of RSQRSP and RCPSP (and also of double-precision-type RSQRDP and RCPDP) is that these instructions provide the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than 2^{-8}). To get higher precision results, Newton-Raphson interpolation is needed.

For example, for RCPxP instructions to calculate 1/v, one Newton-Raphson interpolation x[n+1]=x[n]*(2 - v *x[n]) can improve the mantissa precision to 2^{-16}, and one additional interpolation can improve the mantissa precision to 2^{-24} for single precision and to 2^{-32} for double-precision.

Similarly for RSQRxP instructions to calculate 1/sqrt(v), one Newton-Raphson interpolation x[n+1]= x[n]*(1.5 - (v/2)*x[n]*x[n]) can improve the mantissa precision to 2^{-16}, and one additional interpolation can improve the mantissa precision to 2^{-24} for single precision and to 2^{-32} for double-precision.

Note that all the interpolation steps are fully pipelinable. There is no need for extra memory to be used for look-up tables. And there is no cache impact associated with table look-up.

See the *C66x CPU and ISA Reference Guide* [2] for detailed information for these instructions.

In the following C66x implementation, one iteration of interpolation for all the optimization steps is used. The reader may try with one more iteration to see what the impact would be on the error performance.

```
    _nassert(n % 4 == 0);
    _nassert((int) a % 8 == 0);
    _nassert((int) ejalpha % 8 == 0);
    _nassert((int) abs_a % 8 == 0);
    #pragma MUST_ITERATE(4,100, 4);
 #pragma UNROLL(2);
    for  ( i = 0; i < n; i++)
    {
      a_sqr  = a[i].real * a[i].real + a[i].imag * a[i].imag;
      oneOverAbs_a = _rsqrsp(a_sqr);  /* 1/sqrt() instruction 8-bit mantissa
precision*/
   /* One interpolation*/
      oneOverAbs_a = oneOverAbs_a * (1.5f - (a_sqr/2.f)* oneOverAbs_a
*oneOverAbs_a);

      abs_a[i]= a_sqr * oneOverAbs_a;

      ejalpha[i].real =a[i].real * oneOverAbs_a;
      ejalpha[i].imag =a[i].imag * oneOverAbs_a;
    }
```

The scheduled loop information is listed below:

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 49
;*      Loop opening brace source line   : 50
;*      Loop closing brace source line   : 60
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 50
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 8
;*      Unpartitioned Resource Bound     : 9
;*      Partitioned Resource Bound(*)    : 9
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     1        1
;*      .D units                     3        6
;*      .M units                     9*       9*
;*      .X cross paths               3        5
;*      .T address paths             3        6
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           3        1     (.L or .S unit)
;*      Addition ops (.LSD)          3        9     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        1
;*      Bound(.L .S .D .LS .LSD)     4        6
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 9  no sploop: Dynlen of 54 > 48
;*         ii = 9  no sploop: Dynlen of 53 > 48
;*         ii = 9  Schedule found with 5 iterations in parallel
;*      Done
```

After two lines of code change (no Q value tuning and no extra functions to code and debug), the first step of C66x optimization is achieved.

The resulting cycle count is 496 and has better error performance and no memory usage for look-up tables. In terms of effort, there is probably less than an hour required to become familiarized with the RSQRSP intrinsic. And, the code is very readable – very similar to generic C code with only one intrinsic instruction.

As a general guideline, if there is a 1/x or 1/sqrt(x) calculation in a loop, it is suggested that a floating-point instruction be used at least for the 1/x or 1/sqrt(x) calculation. Table 4 shows the results.

Table 4 Division Instruction Results

	Cycles	Max *abs_a* Error	Max *ejalpha* Error
Optimized C64x+	613	2.359918e-04	2.765343e-04
First Step C66x Optimization	496	0.000000e+00	1.090497e-05

3.1.6 Using Complex Arithmetic

If additional cycle reduction is required, how to further optimize the C66x code must be examined. As can be seen in the compiler output for step 1, the loop is bounded by the M unit. More SIMD instructions to do floating-point multiplication are employed in this step. Although the loop is not bound by the data path, methods to do wider load and store for floating-point are explored. In this step, floating-point intrinsics for load and store, methods to form a register pair and extract a register from a pair, as well as intrinsics to do complex multiplication and dual real multiplication are covered.

3.1.6.1 Load and Store of Floating-Point Numbers

Just as using _amem8(addr) allows simultaneous reading or writing of aligned 8-byte integers, _amemd8(addr) can be used for loading and storing 8-byte floating-point numbers. One _amemd8(addr) can read a single-precision complex number into a pair of registers. In this implementation, the structure cplxf_t is defined as:

```
#ifdef _LITTLE_ENDIAN
typedef struct _CPLXF
{
  float imag;
  float real;
} cplxf_t;
#else
typedef struct _CPLXF
{
  float real;
  float imag;
} cplxf_t;
#endif
```

Using this structure, a floating-point complex number can be loaded using _amemd8() to a pair of registers. The real part resides in the odd register and the imaginary part resides in the even register of the pair. This is the natural order to take advantage of complex multiplication instructions offered by the C66x. Assume that this structure is used throughout this document, unless stated otherwise.

3.1.6.2 Form a Register Pair and Extract a Value From a Register Pair

If the complex number is stored in a register pair in the way defined previously, _hif(src) can be used to get the real part of the complex value, and _lof(src) can be used to get the imaginary part. This is similar to _hill(src) and _loll(src) intrinsics in the C64x+.

To form a register pair from 2 floating point numbers, _fod(src1, scr2) is needed. If this intrinsics is used to form a complex number, _fod(real, imag) must be done. This is similar to _itoll(srcq, src2) intrinsic in C64x+.

See the *C66x CPU and ISA Reference Guide* [2] for other intrinsics in this category.

3.1.6.3 Floating-Point Multiplication

In addition to the MPYSP (SPxSP->SP), MPYDP (DPxDP->DP), MPYSPDP (SPxDP->DP), and MPYSP2DP (SPxSP->DP) instructions already supported in the C674x+, the C66x introduces several varieties of SIMD instructions in the area of complex multiplication.

- DMPYSP:
  - Two-way single-precision floating point multiply producing two single-precision results: C[i] = A[i] * B[i] for i=0 to 1.
  - Both sources and results are in double-precision format.

- CMPYSP:
  - Perform the multiply operations for a complex multiply of two complex numbers a and b. Both sources are in double-precision format. The result is in 128-bit format:

    C3 = A[1] * B[1]

    C2 = A[1] * B[0]

    C1 = -A[0] * B[0]

    C0 = A[0] * B[1]
  - To get the full complex multiplication for A (stored in register pair {A[1] A[0]}) and B (stored in register pair {B[1] B[0]}), the following steps are required:
    - Define a 128-bit data type C: __x128_t C_128;
    - C_128 = _cmpysp(A, B);
    - C = _daddsp(_hid128(C_128), _lod128(C_128)); This step is to do C3+C1 and C2+C0 in a single instruction.
    - Another simpler way is to use combined intrinsics _complex_mpysp(): C=_complex_mpysp(A, B). If this intrinsic is used, both sources and results are in double-precision format.
  - To get the full complex multiplication for A and conjugation of B, the following steps are required:
    - Define a 128-bit data type C: __x128_t C_128;
    - C_128 = _cmpysp(B, A); This gives:

      C3 = B[1] * A[1]

      C2 = B[1] * A[0]

      C1 = -B[0] * A[0]

      C0 = B[0] * A[1]
    - C = _dsubsp(_hid128(C_128), _lod128(C_128)); This step is to do C3-C1 and C2-C0 in a single instruction.
    - Another simpler way is to use combined intrinsics _complex_conjugate_mpysp(): C=_complex_conjugate_mpysp(A, B). If this intrinsic is used, both sources and results are in double format.
  - QMPYSP:
    - Four-way single-precision floating point multiply producing four single-precision results: C[i] = A[i] * B[i] for i=0 to 3.
    - Both sources are in double format. The result is in 128-bit format.

3.1.6.4 Code and Results

```
    _nassert(n % 4 == 0);
    _nassert((int) a % 8 == 0);
    _nassert((int) ejalpha % 8 == 0);
    _nassert((int) abs_a % 8 == 0);
    #pragma MUST_ITERATE(4,100, 4);
    for  ( i = 0; i < n; i++)
    {
      dtemp   =_amemd8(&a[i]);
      /* using SIMD CMPYSP with conjugation for power calculation */
      a_sqr   =_hif(_complex_conjugate_mpysp(dtemp, dtemp));
   /* or use the following                                    */
      /* dtemp2 =   _dmpysp(dtemp, dtemp);     */
      /*   a_sqr    =   _hif(dtemp2) + _lof(dtemp2);  */
      oneOverAbs_a  =  _rsqrsp(a_sqr);  /* 1/sqrt() instruction 8-bit mantissa
precision*/
```

```
      /* 1st interpolation*/
      oneOverAbs_a  =  oneOverAbs_a * (1.5f - (a_sqr/2.f)* oneOverAbs_a
*oneOverAbs_a);

      abs_a[i]   =a_sqr * oneOverAbs_a;
      dtemp1   =_ftod(oneOverAbs_a, oneOverAbs_a);
      /* using SIMD DMPYSP for the following operations */
      /* ejalpha[i].real   = a[i].real * oneOverAbs_a;*/
      /* ejalpha[i].imag   = a[i].imag * oneOverAbs_a;*/
      _amemd8(&ejalpha[i]) = _dmpysp(dtemp, dtemp1);
    }
```

In this version, _complex_conjugate_mpysp(a, a) is used to calculate the power of complex number a. Because the result is a real value, only the 32 MSB of the register pair is needed.

It is also possible to use DMPYSP(a, a) followed by a _hif() + _lof() to calculate the power. This piece of code is in the commented lines.

Both intrinsics have been tried and the cycles are similar.

The scheduled loop information is listed below. Please note the loop is unrolled twice by the compiler.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 76
;*      Loop opening brace source line   : 77
;*      Loop closing brace source line   : 94
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 50
;*      Known Maximum Trip Count         : 50
;*      Known Max Trip Count Factor      : 50
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 7
;*      Partitioned Resource Bound(*)    : 7
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     1        1
;*      .D units                     3        3
;*      .M units                     7*       7*
;*      .X cross paths               1        2
;*      .T address paths             3        3
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           3        1     (.L or .S unit)
;*      Addition ops (.LSD)          2        2     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        1
;*      Bound(.L .S .D .LS .LSD)     3        3
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 7  Schedule found with 7 iterations in parallel
;*      Done
```

By using SIMD floating-point instructions for multiplication, another 1.8× improvement over the first step optimization is obtained. The loop is well balanced for A side and B side after 2× unrolling by the compiler. Yet the code is still heavily M unit-bound. If the loop is examined again, it can be seen that there are still five multiplications per iteration that are not found in the SIMD method, and the compiler was not able to pair them up when unrolling the loop by two. Because manual unrolling has been discussed in the previous sections and generally applicable to both fixed-point and floating-point operations, the reader is left to do the manual unrolling exercise and see how much extra improvement is obtainable.

Also please note that because the method used to accomplish 1/sqrt() in step 2, the error performance stays the same as step 1. Table 5 shows the results.

Table 5 Complex Arithmetic Results

	Cycles	Max *abs_a* Error	Max *ejalpha* Error
Optimized C64x+	613	2.359918e-04	2.765343e-04
First Step C66x Optimization	496	0.000000e+00	1.090497e-05
Second Step C66x Optimization	419	0.000000e+00	1.090497e-05

3.1.7 Starting With Optimized Fixed-Point Code

As detailed in previous sections, if starting with a piece of generic code containing division, it is not difficult to do quick optimization in floating-point to achieve very good cycle performance and error performance. This is especially helpful if the user starts with new development from algorithm simulations in Matlab or generic C code.

Then these questions may come up: what if optimized fixed-point C code already exists? There are divisions and a lot of scaling in the loop, and floating-point may help. Can the old API be kept? Must everything be converted into floating-point?

This section starts with the C64x+ optimized code described in Section 1, keeps the same API, and tries to use floating-point to optimize it further.

Additional methods to use floating-point arithmetic to improve cycle performance of existing optimized fixed-point code is explored.

Techniques to implement mixed-fixed-floating-point programming and how to do conversions between different formats more effectively are covered.

This conversion to floating-point usually increases the precision of the results compared to the original fixed-point code. It is not appropriate to compare the bit-exactness with the fixed-point code and test vectors.

3.1.7.1 Candidates for Using Floating-Point for Further Optimization

Floating-point advantages were discussed in Section 3.1.1. If a developer starts with existing optimized fixed-point code, here are the key areas that floating-point may help:

- 1/x or 1/sqrt(x) in a loop
  - Typically in fixed-point, division is done using table look-up and interpolation. There is an extra memory cost for this approach and maybe cache implications.
  - There are also steps needed to scale the results to the desirable Q values. This could be tedious if there is complicated arithmetic following the division.
- Algorithms that work on data with larger dynamic range
  - In this case, a wider bit-width is needed for input/output/intermediate and 40-bit or even 64-bit arithmetic is needed. This will result in extra cycle cost in fixed-point implementation. For floating-point, using single-precision as an example, there is fixed 24-bit mantissa precision and a much bigger dynamic range of $\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$

- Multiple instances of multiplication with rounding and scaling
  - Many algorithms involve several multiplication iterations. If the multiplicands are not scaled up to the full range, then in fixed-point each multiplication can cause loss of precision if the result is in the same bit-width format of the inputs. The user needs to spend extra cycles either pre-scaling all the inputs to the desirable ranges and keeping track of the Q values, or use multiplication instructions with higher bit-width results, and then scale the results accordingly.

Note that with existing fixed-point code and supporting test bench, converting to floating-point may fail the bit-exactness test using the fixed-point test vectors. The mismatch does not imply floating-point precision is worse. Actually, in most cases, the floating-point has better precision.

3.1.7.2 Conversion Between Floating-Point and Integer

If mixed-fixed-floating-point programming is used in the code, data format conversion must be very effective. In the C66x, the following SIMD instructions are added:

- **DINTHSP, DINTHSPU, and DSPINTH:** convert a pair of 16-bit integers into a pair of single-precision floating-point numbers (with or without sign), and convert a pair of single-precision floating-point numbers to a pair of 16-bit integers.
  - This set is very useful for converting a 16-bit I/Q complex number into a floating-point complex number.
- **DINTSP, DINTSPU and DSPINT:** convert a pair of 32-bit integers into a pair of single-precision floating-point numbers (with or without sign), and convert a pair of single-precision floating-point numbers to a pair of 32-bit integers.
  - This set is very useful for converting a 32-bit I/Q complex number into a floating-point complex number.

Both sets of instructions can be executed on L and S units.

3.1.7.3 Step-by-Step: Optimizing C64x+ Code Using C66x floating-Point

Consider the C64x+ implementation again:

- DOTP2 was used to calculate the power of the input 16-bit complex number, resulting in a 32-bit integer. There is no precision loss in this step. It is efficient and uses fewer registers in the intermediate steps. Optimizing using this instruction will be kept and the 32-bit results converted to floating-point for the input of the 1/sqrt() calculation.
- For the 1/sqrt() calculation, the inlined function call can be directly replaced with RSQRSP and one interpolation.
- For the *abs(a)* result, multiply a^2 to 1/sqrt(a^2). This result should be already in the same Q level as input *a*. Use cast to convert it into a short integer.

- *ejalph(a)* depends on input *a,* which is still in fixed-point, and 1/sqrt(a^2), which is in floating-point. To multiply them together, either *a* must be converted to floating-point, or 1/sqrt(a^2) must be converted to fixed-point. Because 1/sqrt(a^2) is a number smaller than 1, conversion from floating-point to a fixed-point number with Q value must be done. Then the result needs to be scaled using this Q.
  - With the first method, convert the input a into floating-point using DINTHSP instructions and calculate *ejalph(a)* in floating-point. Then convert it into Q15 fixed-point. In this particular example, *ejalph(a)* is already a normalized complex number and can be easily converted into a Q15 fixed-point number by multiplying by 32768 and no overflow should occur.
  - The second method is more general for floating-point to fixed-point conversion with Q, but it will also cost more cycles. This method will not be pursued for this particular example.
  - Converting from floating-point complex number to 16-bit fixed-point I/Q values uses the DSPINTH instruction.

Core loop comparison between C64x+ code and converted code in the C66x is shown in Table 6.

Table 6 Core Loop Comparison

C64x+ loop	C66x loop

`for ( i = 0; i < n; i++)`
`{`
`temp1 = _amem4(&a[i]);`
`a_sqr = _dotp2(temp1, temp1);`
 	
`/* 1/sqrt(a_sqr) */`
`Normal = _norm(a_sqr);`
`normal = normal & 0xFFFFFFFE;`
`x_norm = _sshvl(a_sqr, normal);`
`normal = normal >> 1;`
`Index = _sshvr(_sadd(x_norm,0x800000),24);`
`oneOverAbs_a=_mpylir(xcbia[Index], x_norm );`
`oneOverAbs_a =_sadd((int)x3sa[Index]<<16, _sshvr(oneOverAbs_a,ShiftValDifp1a[Index]));`
`normal =15-ShiftVala[Index] + normal;`
 	
`ejbeta_re =_sadd(_sshvl(_mpyhir(temp1, oneOverAbs_a), normal - 1), 0x8000);`
`ejbeta_im =_sadd(_sshvl(_mpylir(temp1, oneOverAbs_a), normal - 1), 0x8000);`
`_amem4(&ejalpha[i])= _packh2(ejbeta_re, ejbeta_im);`
`abs_a[i] = sshvr(_sadd(_mpyhir(oneOverAbs_a, a_sqr), 1<<(15 - normal)), 16-normal) ;`
`}`
	`for ( i = 0; i < n; i++)`
	`{`
	`temp = _amem4(&a[i]);`
	`a_sqr =(float) ((int) _dotp2(temp, temp));`
	
	`dtemp =_dinthsp(temp);`
	`oneOverAbs_a =_rsqrsp(a_sqr);`
	`/* 1st interpolation*/`
	`oneOverAbs_a = oneOverAbs_a * (1.5f - (a_sqr/2.f)* oneOverAbs_a *oneOverAbs_a);`
	
	`abs_a[i] =(short)(a_sqr * oneOverAbs_a);`
	`dtemp1 = _ftod(oneOverAbs_a, oneOverAbs_a);`
	`dtemp1 = _dmpysp(dtemp, dtemp1);`
	`dtemp1= _dmpysp(_ftod(32768.f, 32768.f), dtemp1);`
	
	`_amem4(&ejalpha[i]) =_dspinth(dtemp1);`
	`}`

The schedule loop information for this C66x is shown below. Note the loop is unrolled by 2.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 162
;*      Loop opening brace source line   : 163
;*      Loop closing brace source line   : 179
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 50
;*      Known Maximum Trip Count         : 50
;*      Known Max Trip Count Factor      : 50
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 8
;*      Partitioned Resource Bound(*)    : 8
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        2
;*      .S units                     2        1
;*      .D units                     3        2
;*      .M units                     8*       8*
;*      .X cross paths               1        7
;*      .T address paths             2        3
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           6        2     (.L or .S unit)
;*      Addition ops (.LSD)          2        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             4        3
;*      Bound(.L .S .D .LS .LSD)     5        4
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 8  Schedule found with 7 iterations in parallel
;*      Done
```

Table 7 shows the gains from the optimized C64x+ code achieved using C66x floating point instructions.

Table 7 Optimization Gains from using Floating-Point Instructions

	Cycles	Max *abs_a* Error	Max *ejalpha* Error
Optimized C64x+	613	2.359918e-04	2.765343e-04
C66x Optimization From C64x+code	523	8.159889e-05	2.223366e-05

By changing to floating-point, approximately1.2× improvement in terms of cycle cost is achieved. Error performance is improved, and the code is more readable.

3.1.7.4 Converting a Floating-Point Number to a Fixed-Point Number with Q Value

A well-optimized C code for converting a floating-point number to a 16-bit fixed-point number with Q value is listed below:

```
/ * strip sign */
sp   = 0x7FFFFFFF & _ftoi(input);

/* shift the 23-bit mantissa to lower 16-bit */
    temp = 0x04C00000 + (head<<23) + (sp & 0xFF800000);
    magic = _itof(0x04C00000 + (head<<23) + (sp & 0xFF800000));

    tempf   = input + magic;
    output  = _ext(_ftoi(input + magic), 16, 16);
    q    = 15 + (127 + 8) - (_ftoi(magic) >> 23);
```

The function finds the magnitude of the input floating point number and uses that to determine the Q-point of the result. The head parameter determines the number of bits of headroom to leave on the result. The headroom value (head) is in the range 0 to 14. Table 8 shows the dynamic range of the output.

Table 8 Output Dynamic Range

Head Room	Low	High
0	-0x8001	+0x7FFF
1	-0xC000	+0x3FFF
2	-0xE000	+0x1FFF
3	-0xF000	+0x0FFF
4	-0xF800	+0x07FF

The user can easily extend this code for multiple conversions, or a complex number conversion.

3.1.7.5 Other SIMD for Floating-Point Complex Number Operations

There are some other SIMDs that can help optimizing floating-point complex number operations:

- **DADDSP and DSUBSP:** these have been used in code shown earlier. These two instructions can do one complex addition or subtraction at a time.
  Both instructions can be executed on the L and S units.
- Because the sign bit of a floating-point value is at the MSB of the 32-bit representation, the XOR instruction (_xor_ll(src)) can be used to negate or conjugate a complex floating-point number. Before using XOR, the double type must be reinterpreted to a long long type by using reinterpretation intrinsics _dtoll(src1). The _xor_ll() instruction can take only long long type as source.

3.2 Complex Matrix Operation and Vector Operations using Advanced C66x Fixed-Point Instructions

The C66x adds additional fixed-point SIMD instructions that enable faster fixed-point complex number operations such as vector addition/subtraction, and vector and matrix complex number multiplication.

3.2.1 Complex Type Definition

Similar to the floating-point complex number definition, within the scope of this document, the following definition for fixed-point complex number is used:

```
#ifdef _LITTLE_ENDIAN
typedef struct _CPLX16
{
  int16_t imag;
  int16_t real;
} cplx16_t;

typedef struct _CPLX32
{
  int32_t imag;
  int32_t real;
} cplx32_t;
#else
typedef struct _CPLX16
{
  int16_t real;
  int16_t imag;
} cplx16_t;

typedef struct _CPLX32
{
  int32_t real;
  int32_t imag;
} cplx32_t;
#endif
```

This definition ensures that when loading with _amem4() for 16-bit I/Q numbers to a register, there is always a real part in the 16MSB and an imaginary part in the 16LSB. In a similar fashion, when loading with _amem8() for 32-bit I/Q numbers to a register pair, there is always a real part in the odd register and an imaginary part in the even register. These are the natural order for the complex multiplication instruction to work properly.

3.2.2 SIMD Instructions for Typical Complex Operation

Table 9 lists SIMD instructions for some typical 16-bit I/Q complex operations.

Table 9 SIMDs for 16-Bit I/Q Complex Operations (Part 1 of 2)

Operation	Instructions	Unit	Intrinsics
ai±bi no saturation, i=0	ADD2/SUB2	L, S, D	_add2/_sub2(a0, b0)
ai±bi no saturation, i=0, 1	DADD2/DSUB2	L,S	_dadd2/_dsub2(ai, bi)
ai±bi with saturation, i=0	SADD2/SSUB2	S/L	_sadd2/_ssub2(a0, b0)
ai±bi with saturation, i=0, 1	DSADD2/DSSUB2	L,S/L	_dsadd2/_dssub2(ai,bi)
aixbi no rounding to 32-bit I/Q, i=0	CMPY	M	_cmpy(a0, b0)
aixbi no rounding to 32-bit I/Q, i=0, 1	DCMPY	M	_dcmpy(ai, bi)
aixconj(bi) no rounding to 32-bit I/Q, i=0, 1	DCCMPY	M	_dccmpy(ai, bi)
aixbi with rounding to 16-bit I/Q, i=0	CMPYR1	M	_cmpyr1(a0, b0)
aixbi with rounding to 16-bit I/Q, i=0, 1	DCMPYR1	M	_dcmpyr1(ai, bi)
ai x conj(bi) with rounding to 16-bit I/Q, i=0, 1	DCCMPYR1	M	_dccmpyr1(ai, bi)

Table 9 SIMDs for 16-Bit I/Q Complex Operations (Part 2 of 2)

Operation	Instructions	Unit	Intrinsics
|ai|^2, i=0	DOTP2	M	_dopt2(a0, a0)
|ai|^2, i=0, 1	DOTP4H	M	_dotp4h(ai, ai)
|ai|^2 i=0,1 and |ai|^2 i=2, 3	DDOTP4H	M	_ddotp4h(ai, ai)
Conj(ai), i=0, 1	DAPYS2	L	_dapys2(0x0000f0000000f000, ai)
-ai, i=0, 1	DAPYS2	L	_dapys2(0xf000f000f000f000, ai)
ai >> k, i=0	SHR2	S	_shr2(a0, k)
ai >> k, i=0, 1	DSHR2	S	_dshr2(ai, k)
ai << k no saturation, i=0	SHL2	S	_shl2(a0, k)
ai << k no saturation, i=0, 1	DSHL2	S	_dshl2(ai, k)
|a0 a1|* |b0 b1| |b2 b3| no rounding to 32-bit I/Q	CMATMPY	M	_cmatmpy(ai, bi)
|a0 a1| * |b0 b1| |b2 b3| with rounding to 16-bit I/Q	CMATMPYR1	M	_cmatmpyr1(ai, bi)
conj(|a0 a1|) * |b0 b1| |b2 b3| no rounding to 32-bit I/Q	CCMATMPY	M	_ccmatmpy(ai, bi)
conj(|a0 a1|) * |b0 b1| |b2 b3| with rounding to 16-bit I/Q	CCMATMPYR1	M	_ccmatmpyr1(ai, bi)

Table 10 lists SIMD instructions for some typical 32-bit I/Q complex operations.

Table 10 SIMDs for 32-Bit I/Q Complex Operations

Operation	Instructions	Unit	Intrinsics
ai±bi no saturation, i=0	DADD/DSUB	L, S, D/L,S,D	_dadd/_dsub(a0, b0)
ai±bi with saturation, i=0	DSADD/DSSUB	S/L	_dsadd/_dssub(a0, b0)
aixbi i=0	CMPY32	M	_cmpy32(a0, b0)
ai >> k, i=0	DSHR	S	_dshr((a0, k)
ai << k no saturation, i=0	DSHL	S	_dshl((a0, k)
Conj(ai), i=0, 1	DAPYS2	L	_dapys2(0x00000000f0000000, a0)
-ai, i=0, 1	DAPYS2	L	_dapys2(0xf0000000f0000000, a0)

3.2.3 Complex Matrix and Multiplication

This section discusses complex matrix multiplication in more detail. The example uses 4×4 matrix multiplications.

3.2.3.1 C64x+ Implementation

A C64x+ implementation:

```
for ( mm = 0; mm < M; mm++ )  //M 4x4 matrix multiplications in the loop
{
  Ptr1 = (int *) &input_mat1[mm * MATRIXSIZE * MATRIXSIZE];
  Ptr2 = (int *) &input_mat2[mm * MATRIXSIZE * MATRIXSIZE];
  OutPtr = (int *) &output_mat[mm * MATRIXSIZE * MATRIXSIZE];
  _nassert( (int) Ptr1 % 8 == 0);
  _nassert( (int) Ptr2 % 8 == 0);
  _nassert( (int) OutPtr % 8 == 0);
```

```
        for ( ii = 0; ii < MATRIXSIZE; ii++ )    //line 70, loop3
        {
          #pragma UNROLL(4);
          for ( jj = 0; jj < MATRIXSIZE; jj++ )  // loop2
          {
            temp     =   0;
         for ( kk = 0; kk < MATRIXSIZE; kk++ )   //inner most loop
         {
         temp=_sadd2(temp,_cmpyr1(_amem4(&Ptr1[ii*MATRIXSIZE+kk]),

                                        _amem4(&Ptr2[kk * MATRIXSIZE + jj])));
         }
         _amem4(&OutPtr[ii * MATRIXSIZE + jj])=temp;
          }
        }
      }
} /* end of MATRIX_Mult4by4_c64p() function */
```

In this implementation, the inner-most loop and loop 2 are unrolled. The idea is to let the compiler schedule for loop3, which calculates a row of input matrix 1 by input matrix 2, and output one row of results (4 elements). This requires 20 32-bit reads, 4 32-bit writes, and 16 complex multiplications. Because the C64x+ can do one complex multiplication per M unit, the loop should be bound by M unit with scheduled ii=8.

The compiler feedback is listed below.

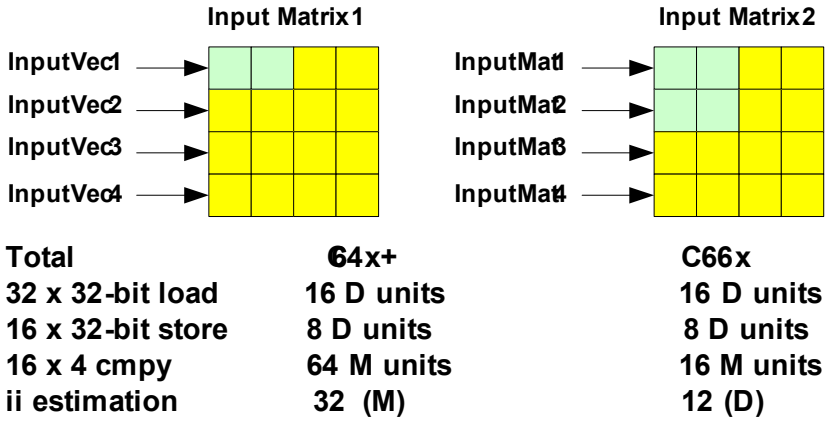
```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 70
;*      Loop opening brace source line   : 71
;*      Loop closing brace source line   : 83
;*      Known Minimum Trip Count         : 4
;*      Known Maximum Trip Count         : 4
;*      Known Max Trip Count Factor      : 4
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 8
;*      Partitioned Resource Bound(*)    : 8
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     6        6
;*      .D units                     3        3
;*      .M units                     8*       8*
;*      .X cross paths               0        0
;*      .T address paths             3        3
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        3
;*      Bound(.L .S .D .LS .LSD)     3        3
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 8  Schedule found with 3 iterations in parallel
;*      Done
```

3.2.3.2 C66x Optimization

The C66x architecture increased the multiplication capacity of the C64x+ by 400%. Therefore, if a loop is M-unit constrained on the C64x+, an improvement should be realized by moving to the C66x. Considering the problem of 4×4 matrix multiplication again, there are a total of 32 32-bit reads, 16 32-bit writes, and 64 complex multiplications. Because the C66x can load 64-bits per D unit and do four complex multiplications per M unit, the C66x loop will be bound by the D unit with scheduled ii = 12, while the M unit is loaded with eight per side.

Figure 2 shows the multiplication of a 1x2 vector by a 2x2 matrix and outlines the the number of load, store and multiplication operations on both the C64x+ and C66x architectures.

Figure 2 illustration of Matrix Multiplication Operations



In this code, the C66x instructions CMATMPYR1 and DSADD2 are used. One (4×4) by (4×4) matrix multiplication is calculated per iteration.

One CMATMPYR1 instruction performs the following operations:

$$[A9 \quad A8] = [A7 \quad A6] * \begin{bmatrix} A3 & A2 \\ A1 & A0 \end{bmatrix}$$

i.e.

$$A9 = A7 * A3 + A6 * A1$$

$$A8 = A7 * A2 + A6 * A0$$

Where: Ai stores 16-bit I/Q complex number. The results are rounded to 16-bit I/Q. Refer to [2] for details of CMATMPYR1, and other flavors of matrix multiplication instructions such as without rounding, or with conjugation.

As shown in Figure 2, the multiplication can be partitioned into a 1×2 vector from source matrix 1 multiplied by a 2×2 matrix from source matrix 2. As shown in blue, read two complex values from inputVec1 pointer to form a 1×2 vector, and use inputMat1 and inputMat2 to read four complex numbers and form a 2×2 matrix. Then use CMATMPYR1 to do the [1×2] by [2×2] multiplication. When inputVec1 pointers advance, use inputMat3 and inputMat4 pointers to form a new 2×2 matrix, perform the multiplication, and accumulate the result to the first step.

In general, multiplication of any even size of matrices can be partitioned to take advantage of the CMATMPYR1 type of instructions.

If a self multiplication of a matrix is needed, meaning a matrix is multiplied by the conjugate transpose of itself, then the D and M resource needed for the operation will be different, and other instructions that can calculate power quickly can be used. This is shown in detail in the following sections.

C66x code and compiler feedback are listed below.

```
for ( mm = 0; mm < M; mm++ )  //M 4x4 matrix multiplications in the loop
{
     input_vec1 =(long long *) &input_mat1[mm * MATRIXSIZE * MATRIXSIZE + 0 * MATRIXSIZE];
     input_vec2 =(long long *) &input_mat1[mm * MATRIXSIZE * MATRIXSIZE + 1 * MATRIXSIZE];
     input_vec3 =(long long *) &input_mat1[mm * MATRIXSIZE * MATRIXSIZE + 2 * MATRIXSIZE];
     input_vec4 =(long long *) &input_mat1[mm * MATRIXSIZE * MATRIXSIZE + 3 * MATRIXSIZE];
     inputMatPtr1=(long long*) &input_mat2[mm * MATRIXSIZE * MATRIXSIZE + 0 * MATRIXSIZE];
     inputMatPtr2=(long long*) &input_mat2[mm * MATRIXSIZE * MATRIXSIZE + 1 * MATRIXSIZE];
     inputMatPtr3=(long long*) &input_mat2[mm * MATRIXSIZE * MATRIXSIZE + 2 * MATRIXSIZE];
     inputMatPtr4=(long long*) &input_mat2[mm * MATRIXSIZE * MATRIXSIZE + 3 * MATRIXSIZE];

     /* (4x4) X (4x4) */
     llinputV1   =_amem8(input_vec1++ );
     llinputV2   =_amem8(input_vec2++ );
     llinputV3   =_amem8(input_vec3++ );
     llinputV4   =_amem8(input_vec4++ );
     llinputM1=_amem8(inputMatPtr1++ );
     llinputM2=_amem8(inputMatPtr2++ );
     #ifdef _LITTLE_ENDIAN
     inputMat    =   _llto128(llinputM2, llinputM1);
     #else
     inputMat    =   _llto128(llinputM1, llinputM2);
     #endif
     acc1    =    _cmatmpyr1(llinputV1, inputMat);
  acc2   =    _cmatmpyr1(llinputV2, inputMat);
     acc3    =    _cmatmpyr1(llinputV3, inputMat);
     acc4    =    _cmatmpyr1(llinputV4, inputMat);
     llinputM1=_amem8(inputMatPtr1++ );
     llinputM2=_amem8(inputMatPtr2++ );
     #ifdef _LITTLE_ENDIAN
     inputMat    =   _llto128(llinputM2, llinputM1);
     #else
     inputMat    =   _llto128(llinputM1, llinputM2);
     #endif
  acc5   =    _cmatmpyr1(llinputV1, inputMat);
     acc6    =    _cmatmpyr1(llinputV2, inputMat);
     acc7    =    _cmatmpyr1(llinputV3, inputMat);
     acc8    =    _cmatmpyr1(llinputV4, inputMat);

     llinputV1   =_amem8(input_vec1++ );
     llinputV2   =_amem8(input_vec2++ );
     llinputV3   =_amem8(input_vec3++ );
     llinputV4   =_amem8(input_vec4++ );

     llinputM1=_amem8(inputMatPtr3++ );
     llinputM2=_amem8(inputMatPtr4++ );
     #ifdef _LITTLE_ENDIAN
     inputMat    =   _llto128(llinputM2, llinputM1);
     #else
     inputMat    =   _llto128(llinputM1, llinputM2);
     #endif
     acc1    =    _dadd2(_cmatmpyr1(llinputV1, inputMat), acc1);
     acc2    =    _dadd2(_cmatmpyr1(llinputV2, inputMat), acc2);
     acc3    =    _dadd2(_cmatmpyr1(llinputV3, inputMat), acc3);
     acc4    =    _dadd2(_cmatmpyr1(llinputV4, inputMat), acc4);
     llinputM1=_amem8(inputMatPtr3++ );
     llinputM2=_amem8(inputMatPtr4++ );
     #ifdef _LITTLE_ENDIAN
  inputMat    =   _llto128(llinputM2, llinputM1);
     #else
     inputMat    =   _llto128(llinputM1, llinputM2);
     #endif
     acc5    =    _dadd2(_cmatmpyr1(llinputV1, inputMat), acc5);
     acc6    =    _dadd2(_cmatmpyr1(llinputV2, inputMat), acc6);
     acc7    =    _dadd2(_cmatmpyr1(llinputV3, inputMat), acc7);
     acc8    =    _dadd2(_cmatmpyr1(llinputV4, inputMat), acc8);

     _amem8(&output_mat[mm * MATRIXSIZE * MATRIXSIZE + 0 * MATRIXSIZE + 0]) = acc1;
     _amem8(&output_mat[mm * MATRIXSIZE * MATRIXSIZE + 0 * MATRIXSIZE + 2]) = acc5;
     _amem8(&output_mat[mm * MATRIXSIZE * MATRIXSIZE + 1 * MATRIXSIZE + 0]) = acc2;
     _amem8(&output_mat[mm * MATRIXSIZE * MATRIXSIZE + 1 * MATRIXSIZE + 2]) = acc6;
     _amem8(&output_mat[mm * MATRIXSIZE * MATRIXSIZE + 2 * MATRIXSIZE + 0]) = acc3;
     _amem8(&output_mat[mm * MATRIXSIZE * MATRIXSIZE + 2 * MATRIXSIZE + 2]) = acc7;
     _amem8(&output_mat[mm * MATRIXSIZE * MATRIXSIZE + 3 * MATRIXSIZE + 0]) = acc4;
     _amem8(&output_mat[mm * MATRIXSIZE * MATRIXSIZE + 3 * MATRIXSIZE + 2]) = acc8;
}
```

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 110
;*      Loop opening brace source line   : 111
;*      Loop closing brace source line   : 190
;*      Known Minimum Trip Count         : 12
;*      Known Maximum Trip Count         : 1200
;*      Known Max Trip Count Factor      : 12
;*      Loop Carried Dependency Bound(^) : 2
;*      Unpartitioned Resource Bound     : 12
;*      Partitioned Resource Bound(*)    : 12
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     4        4
;*      .D units                    12*      12*
;*      .M units                     8        8
;*      .X cross paths               8        8
;*      .T address paths            12*      12*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           4        4     (.L or .S unit)
;*      Addition ops (.LSD)          4        4     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             4        4
;*      Bound(.L .S .D .LS .LSD)     8        8
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 12 Cannot allocate machine registers
;*                   Regs Live Always   :  8/8  (A/B-side)
;*                   Max Regs Live      : 29/28
;*                   Max Cond Regs Live :  0/0
;*         ii = 12 Schedule found with 3 iterations in parallel
;*      Done
```

3.3 Matrix Inversion Considerations

Matrix inversion is one of the key parts for many applications. In the complex domain, the inversion of Hermitian matrices is indicated.

Typically in fixed-point, for a matrix size larger than 2×2, a decomposition-based algorithm must be used to deal with a not-well-conditioned matrix to achieve stable output.

With the larger dynamic range of floating-point, simpler algorithms such the analytical co-factor method and blockwise method can be used. With larger matrix sizes, even for floating-point, a decomposition-based algorithm will be needed. But for a matrix size smaller than 8×8, stable performance with much lower cycle cost can be achieved by using simpler algorithms.

4 Additional Tuning Techniques for C66x Software-Pipelined Loops

It is helpful to have

- A good understanding of TMX320C66x core benefits highlighted in section 2 of this application note as well as techniques to determine when to have complete fixed-point implementation, complete floating-point implementation, or hybrid fixed-point and floating-point hybrid implementation highlighted in section 3.1.
- A familiarity with TMS320C66x C intrinsics and the new 128 data type (__x128_t) defined specially for the C66x compiler.

The TMS320C66x C compiler does not have a complex data type and has limited vectorization support, and, therefore, it is expected that the programmer will use C intrinsics and do manual vectorizing if TMS320C66x SIMD-like instructions are to be used. Because there are multiple methods (using different instructions) to solve a problem in the TMS320C66x, it is very important for the programmer to analyze and select the appropriate instruction (via intrinsics) that will give the desired performance. Note that all the examples used in this chapter are complied with complier version PC v7.2.0A10232.

4.1 Reducing Register Pressure in the TMS320C66x

4.1.1 Avoid 4-Way SIMD Instructions to Reduce Likelihood of Register Pressure

As mentioned in section 2.1, SIMD-like operations have been added to the C66x core and some of the instructions use quad registers for operands and place results in quad registers. Use those instructions with care, otherwise register pressure may be increased in the software pipeline loop and result in a bad schedule for the code. The following example shows the register pressure behavior and explains how to reduce register pressure.

Following example performs a part of a *2 × 4 MIMO MMSE equalizer* and is used to demonstrate the possible register pressure issues associated with using C66x SIMD instructions (especially ones with quad register operands). This code was initially written in for the TMS320C64x+ fixed-point core and then converted into TMS320C66x hybrid fixed-point and floating-point using the techniques highlighted in the previous chapter.

Table 11 MMSE Equalizer Example Demonstrating Register Pressure (Part 1 of 3)

Line	Code
1	for(kk=0; kk<numDataSubCars; kk++)
2	{
3	HH11 = chan0[0][kk];
4	HH12 = chan1[0][kk];
5	HH21 = chan0[1][kk];
6	HH22 = chan1[1][kk];
7	HH31 = chan0[2][kk];
8	HH32 = chan1[2][kk];
9	HH41 = chan0[3][kk];
10	HH42 = chan1[3][kk];
11	H11_H21 = _itoll(HH11, HH21);
12	H12_H22 = _itoll(HH12, HH22);
13	H31_H41 = _itoll(HH31, HH41);

Table 11 MMSE Equalizer Example Demonstrating Register Pressure (Part 2 of 3)

Line	Code
14	H32_H42 = _itoll(HH32, HH42);
15	
16	AA11_40 = (float) noiseVar + (float) ((Int32) _dotp4h(H11_H21, H11_H21));
17	result_0 = _dccmpy(H12_H22, H11_H21);
18	Ain1 = _dintsp(_dadd(_hi128(result_0), _lo128(result_0)));
19	
20	RR1 = _amem4(&recDataBuffPtr[0][kk + allocDesc[alloc].init_k0]);
21	RR2 = _amem4(&recDataBuffPtr[1][kk + allocDesc[alloc].init_k0]);
22	y1_y2 = _itoll(RR1, RR2);
23	
24	result_1 = _dccmpy(H11_H21, y1_y2);
25	Bin1 = _dintsp(_dadd(_hi128(result_1), _lo128(result_1)));
26	
27	result_2 = _dccmpy(H12_H22, y1_y2);
28	Bin2 = _dintsp(_dadd(_hi128(result_2), _lo128(result_2)));
29	AA22_40 = (float) noiseVar + (float) ((Int32) _dotp4h(H12_H22, H12_H22));
30	
31	AA11_40 = AA11_40 + (float) ((Int32) _dotp4h(H31_H41, H31_H41));
32	result_0 = _dccmpy(H32_H42, H31_H41);
33	Ain1 = _daddsp(Ain1, _dintsp(_dadd(_hi128(result_0), _lo128(result_0))));
34	
35	RR3 = _amem4(&recDataBuffPtr[2][kk + allocDesc[alloc].init_k0]);
36	RR4 = _amem4(&recDataBuffPtr[3][kk + allocDesc[alloc].init_k0]);
37	y3_y4 = _itoll(RR3, RR4);
38	
39	result_1 = _dccmpy(H31_H41, y3_y4);
40	Bin1 = _daddsp(Bin1, _dintsp(_dadd(_hi128(result_1), _lo128(result_1))));
41	
42	result_2 = _dccmpy(H32_H42, y3_y4);
43	Bin2 = _daddsp(Bin2, _dintsp(_dadd(_hi128(result_2), _lo128(result_2))));
44	AA22_40 = AA22_40 + (float) ((Int32) _dotp4h(H32_H42, H32_H42));
45	
46	dtemp = _dmpysp(Ain1, Ain1);
47	iDenom = AA22_40 * AA11_40 - _hif(dtemp) - _lof(dtemp);
48	bb = (float)_rcpsp((float)iDenom);
49	bb = bb * (2.f - iDenom * bb); /* 16-bit precision */
50	IlrDenom0 = IlrDenom0 + bb * AA22_40;
51	IlrDenom1 = IlrDenom1 + bb * AA11_40;
52	
53	result0 = _cmpysp(Ain1, Bin2); //conj(H'Y/(H'H+sigmaI))
54	dtemp = _dsubsp(_hid128(result0), _lod128(result0));
55	iNumer0 = _dsubsp(_dmpysp(_ftod(AA22_40, AA22_40), Bin1), dtemp);
56	result1 = _cmpysp(Ain1, Bin1);
57	dtemp = _daddsp(_hid128(result1), _lod128(result1));

Table 11 MMSE Equalizer Example Demonstrating Register Pressure (Part 3 of 3)

Line	Code
58	iNumer1 = _dsubsp(_dmpysp(_ftod(AA11_40, AA11_40), Bin2), dtemp);
59	/* Compute LLR scaling factor denominator */
60	bb = bb * (float) outRms;
61	qtemp = _fto128(bb, bb, bb, bb);
62	iNumer0_iNumer1 = _dto128(iNumer0, iNumer1);
63	DDhat0_DDhat1 = _qmpysp(qtemp, iNumer0_iNumer1);
64	DDhat0 = _hid128(DDhat0_DDhat1);
65	DDhat1 = _lod128(DDhat0_DDhat1);
66	
67	Out1 = _dspinth(DDhat0);
68	Out1 = _loll(_dapys2(0x0001FFFF, Out1));
69	Out2 = _dspinth(DDhat1);
70	Out2 = _loll(_dapys2(0x0001FFFF, Out2));
71	eqOut0[kk] = Out1;
72	eqOut1[kk] = Out2;
73	}
End of Table 11	

Compile this function with –o3 –s –mw –mv6600, which is implicitly little-endian. As can be seen from the optimizer comments, the result is a loop that has been schedule for ii = 25.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 890
;*      Loop opening brace source line   : 891
;*      Loop closing brace source line   : 970
;*      Known Minimum Trip Count         : 12
;*      Known Maximum Trip Count         : 1296
;*      Known Max Trip Count Factor      : 12
;*      Loop Carried Dependency Bound(^) : 12
;*      Unpartitioned Resource Bound     : 13
;*      Partitioned Resource Bound(*)    : 14
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     1        1
;*      .S units                     1        3
;*      .D units                     7        7
;*      .M units                    10       10
;*      .X cross paths              11       10
;*      .T address paths             7        7
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)          22       19     (.L or .S unit)
;*      Addition ops (.LSD)         11        6     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            12       12
;*      Bound(.L .S .D .LS .LSD)    14*      12
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 14 Did not find schedule
;*         ii = 15 Cannot allocate machine registers
;*                   Regs Live Always   : 12/10 (A/B-side)
;*                   Max Regs Live      : 48/34
;*                   Max Cond Regs Live :  1/0
;*         ii = 15 Did not find schedule
;*         ii = 16 Cannot allocate machine registers
;*                   Regs Live Always   : 12/10 (A/B-side)
;*                   Max Regs Live      : 43/31
;*                   Max Cond Regs Live :  1/0
;*         ii = 16 Cannot allocate machine registers
```

```
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 44/31
;*                    Max Cond Regs Live :  1/0
;*          ii = 16 Did not find schedule
;*          ii = 17 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 40/34
;*                    Max Cond Regs Live :  1/0
;*          ii = 17 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 44/36
;*                    Max Cond Regs Live :  1/0
;*          ii = 17 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 42/30
;*                    Max Cond Regs Live :  1/0
;*          ii = 17 Did not find schedule
;*          ii = 18 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 41/33
;*                    Max Cond Regs Live :  1/0
;*          ii = 18 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 39/34
;*                    Max Cond Regs Live :  1/0
;*          ii = 18 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 43/28
;*                    Max Cond Regs Live :  1/0
;*          ii = 18 Did not find schedule
;*          ii = 19 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 33/29
;*                    Max Cond Regs Live :  1/0
;*          ii = 19 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 38/31
;*                    Max Cond Regs Live :  1/0
;*          ii = 19 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 39/32
;*                    Max Cond Regs Live :  1/0
;*          ii = 19 Did not find schedule
;*          ii = 20 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 35/27
;*                    Max Cond Regs Live :  1/0
;*          ii = 20 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 40/32
;*                    Max Cond Regs Live :  1/0
;*          ii = 20 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 37/29
;*                    Max Cond Regs Live :  1/0
;*          ii = 20 Did not find schedule
;*          ii = 21 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 37/28
;*                    Max Cond Regs Live :  1/0
;*          ii = 21 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 35/28
;*                    Max Cond Regs Live :  1/0
;*          ii = 21 Did not find schedule
;*          ii = 22 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 37/28
;*                    Max Cond Regs Live :  1/0
;*          ii = 22 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 39/32
;*                    Max Cond Regs Live :  1/0
;*          ii = 22 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 39/27
;*                    Max Cond Regs Live :  1/0
;*          ii = 22 Did not find schedule
;*          ii = 23 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 33/25
```

```
;*                    Max Cond Regs Live :  1/0
;*         ii = 23 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 33/27
;*                    Max Cond Regs Live :  1/0
;*         ii = 23 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 39/27
;*                    Max Cond Regs Live :  1/0
;*         ii = 23 Did not find schedule
;*         ii = 24 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 32/25
;*                    Max Cond Regs Live :  1/0
;*         ii = 24 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 34/28
;*                    Max Cond Regs Live :  1/0
;*         ii = 24 Cannot allocate machine registers
;*                    Regs Live Always   : 12/10 (A/B-side)
;*                    Max Regs Live      : 37/26
;*                    Max Cond Regs Live :  1/0
;*         ii = 24 Did not find schedule
;*         ii = 25 Schedule found with 3 iterations in parallel
;*
;*
;*        SINGLE SCHEDULED ITERATION
;*
;*        $C$C971:
;*   0              LDW     .D1T2   *A28++,B16        ; |917|
;*   1              LDW     .D2T2   *B23++,B0         ; |910|
;*   2              LDW     .D2T2   *B26++,B28        ; |926|
;*   3              LDW     .D1T2   *A29++,B29        ; |926|
;*   4              LDW     .D1T2   *A26++,B1         ; |910|
;*   5              LDW     .D1T1   *A27++,A22        ; |910|
;*     ||           LDW     .D2T2   *B25++,B17        ; |917|
;*   6              LDW     .D2T1   *B24++,A23        ; |910|
;*   7              LDW     .D1T1   *A30++,A18        ; |927|
;*   8              LDW     .D2T1   *B30++,A19        ; |927|
;*   9              DMV     .S2     B1,B0,B5:B4       ; |926|
;*     ||           DMV     .L2     B29,B28,B7:B6     ; |926|
;*     ||           LDW     .D2T1   *B31++,A24        ; |934|
;*  10             MV      .S2X    A22,B4            ; |910| Define a twin register
;*     ||           DDOTP4H .M2     B7:B6:B5:B4,B7:B6:B5:B4,B5:B4 ; |926|
;*     ||           LDW     .D1T1   *A31++,A25        ; |934|
;*  11             MV      .S2X    A23,B5            ; |910| Define a twin register
;*     ||           DCCMPY  .M2     B1:B0,B17:B16,B19:B18:B17:B16 ; |917|
;*     ||           DCCMPY  .M1X    A23:A22,B17:B16,A7:A6:A5:A4 ; |920|
;*  12              DCCMPY  .M2     B5:B4,B1:B0,B7:B6:B5:B4 ; |910|
;*  13              DCCMPY  .M1X    A19:A18,B29:B28,A19:A18:A17:A16 ; |927|
;*     ||           DMV     .L1     A23,A22,A5:A4     ; |939|
;*     ||           DMV     .S1     A19,A18,A7:A6     ; |939|
;*  14             MV      .L2X    A24,B0            ; |934| Define a twin register
;*     ||           DDOTP4H .M1     A7:A6:A5:A4,A7:A6:A5:A4,A19:A18 ; |939|
;*  15              DADD    .L1     A7:A6,A5:A4,A19:A18 ; |921|
;*     ||           MV      .D2X    A25,B1            ; |934| Define a twin register
;*     ||           DCCMPY  .M1     A19:A18,A25:A24,A7:A6:A5:A4 ; |937|
;*     ||           INTSP   .L2     B4,B29            ; |944|
;*     ||           INTSP   .S2     B5,B28            ; |944|
;*  16              DADD    .L2     B7:B6,B5:B4,B5:B4 ; |911|
;*     ||           DINTSP  .S1     A19:A18,A9:A8     ; |921|
;*     ||           DCCMPY  .M2     B29:B28,B1:B0,B7:B6:B5:B4 ; |934|
;*  17              DINTSP  .S2     B5:B4,B21:B20     ; |911|
;*     ||           DADD    .S1     A19:A18,A17:A16,A17:A16 ; |944|
;*  18              INTSP   .L1     A18,A6            ; |944|
;*  19              DINTSP  .S2X    A17:A16,B5:B4     ; |944|
;*     ||           INTSP   .L1     A19,A18           ; |944|
;*     ||           DADD    .S1     A7:A6,A5:A4,A5:A4 ; |948|
;*  20              DADD    .S2     B19:B18,B17:B16,B17:B16 ; |918|
;*     ||           FADDSP  .L2X    B29,A1,B16        ; |944|
;*     ||           DINTSP  .S1     A5:A4,A7:A6       ; |948|
;*  21              DINTSP  .L2     B17:B16,B7:B6     ; |918|
;*     ||          DADD    .S1X    0,B5:B4,A17:A16   ; |934| Define a twin register
;*  22              DADDSP  .L2     B21:B20,B5:B4,B5:B4 ; |944|
;*     ||           FADDSP  .L1     A6,A1,A16         ; |944|
;*  23             DADD    .S1X    0,B7:B6,A9:A8     ; |934| Define a twin register
;*     ||           DADDSP  .L1     A9:A8,A7:A6,A21:A20 ; |948|
;*  24              DADD    .S1     A9:A8,A17:A16,A9:A8 ; |951|
;*  25              DMPYSP  .M2     B5:B4,B5:B4,B19:B18 ; |944|
;*     ||           FADDSP  .L1     A18,A16,A3        ; |944|  ^
;*  26              FADDSP  .L2     B28,B16,B27       ; |944|
```

```
;*     ||           DINTSP  .S2X    A9:A8,B17:B16     ; |951|
;*  27              NOP             1
;*  28              MV      .L1     A3,A17            ; |959|
;*     ||           MV      .S1     A3,A16            ; |959| Define a twin register
;*  29              MV      .S2     B27,B22           ; |944| Split a long life
;*     ||           MPYSP   .M2X    B27,A3,B6         ; |944|  ^
;*     ||           DADDSP  .L2     B7:B6,B17:B16,B21:B20 ; |951|
;*  30              MV      .L1X    B27,A18           ; |959| Define a twin register
;*  31              NOP             2
;*  33              FSUBSP  .L2     B6,B19,B5         ; |944|  ^
;*     ||           MV      .L1X    B27,A19           ; |959|
;*  34              DADD    .S1X    0,B5:B4,A9:A8     ; |944| Define a twin register
;*     ||           DMPYSP  .M1     A19:A18,A21:A20,A9:A8 ; |959|
;*  35              CMPYSP  .M1X    A9:A8,B21:B20,A23:A22:A21:A20 ; |951|
;*  36              FSUBSP  .L2     B5,B18,B6         ; |944|  ^
;*  37              CMPYSP  .M1     A9:A8,A21:A20,A7:A6:A5:A4 ; |948|
;*  38              NOP             1
;*  39              RCPSP   .S2     B6,B27            ; |944|  ^
;*     ||           DADDSP  .S1     A23:A22,A21:A20,A21:A20 ; |959|
;*  40              MPYSP   .M2     B27,B6,B4         ; |944|  ^
;*  41              DMPYSP  .M1X    A17:A16,B21:B20,A17:A16 ; |959|
;*  42              DSUBSP  .L1     A7:A6,A5:A4,A7:A6 ; |959|
;*  43              DSUBSP  .S1     A9:A8,A21:A20,A5:A4 ; |959|
;*  44              FSUBSP  .L2     B12,B4,B16        ; |944|  ^
;*  45              NOP             1
;*  46              DSUBSP  .L1     A17:A16,A7:A6,A7:A6 ; |959|
;*  47              MPYSP   .M2     B27,B16,B17       ; |944|  ^
;*  48              NOP             3
;*  51              MPYSP   .M2     B17,B22,B6        ; |946|
;*  52              MPYSP   .M2X    B17,A3,B7         ; |945|  ^
;*     ||           MPYSP   .M1X    B17,A2,A8         ; |958|
;*  53              NOP             3
;*  56              FADDSP  .L2     B7,B10,B10        ; |945|
;*     ||           MV      .D1     A8,A11            ; |958|
;*     ||           MV      .S1     A8,A10            ; |958| Define a twin register
;*     ||           MV      .L1     A8,A9             ; |958|
;*  57              FADDSP  .L1X    B6,A15,A15        ; |946|
;*  58              QMPYSP  .M1     A11:A10:A9:A8,A7:A6:A5:A4,A7:A6:A5:A4 ; |960|
;*  59              NOP             3
;*  62              DSPINTH .L1     A7:A6,A20         ; |968|
;*     ||   [ A0]   SUB     .S1     A0,1,A0           ; |890|
;*  63              DSPINTH .L2X    A5:A4,B11         ; |969|
;*     ||   [ A0]   B       .S2     $C$C971           ; |890|
;*  64              NOP             1
;*  65              SHR     .S1     A20,31,A21        ; |968|
;*  66              DAPYS2  .L1     A13:A12,A21:A20,A23:A22 ; |968|
;*     ||           MV      .D2     B11,B6            ; |969|
;*     ||           SHR     .S2     B11,31,B7         ; |969|
;*  67              STW     .D2T1   A22,*B3++         ; |968|
;*     ||           DAPYS2  .L2     B9:B8,B7:B6,B5:B4 ; |969|
;*  68              STW     .D1T2   B4,*A14++         ; |969|
;*  69              ; BRANCHCC OCCURS {$C$C971}       ; |890|
;*----------------------------------------------------------------------------*
```

This loop is resource bound on the .L, .S and .D units for ii=14. The compiler was able to find a schedule at ii=15, but the schedule needs more physical register than what is available (32 of A side and 32 on B side). Therefore the compiler attempts a different schedule with incrementally larger cycle counts until finally it gave a schedule at ii=25.

It is clear from the optimizer feedback that there are more live-in variables than available machine registers for schedules between ii = 14 and ii=24.

Consider the SIMD instructions that are being selected and analyze whether those instruction are increasing the register pressure. The rule to follow is:

If the loop is not bound by any given functional unit, then do not use SIMD instructions for that unit. Otherwise register pressure may be increased.

The following C code uses a four-way SIMD version of MPYSP, QMPYSP, at line 63 of the C source code. QMPYSP is broken into a 2-way SIMD version of MPYSP, DMPYSP, to determine whether QMPYSP is responsible for the existing register pressure as shown in Table 12.

Table 12 Register Pressure Analysis - QMPYSP Versus Dual DMPYSP (Part 1 of 2)

Line	Code
1	for(kk=0; kk<numDataSubCars; kk++)
2	{
3	HH11 = chan0[0][kk];
4	HH12 = chan1[0][kk];
5	HH21 = chan0[1][kk];
6	HH22 = chan1[1][kk];
7	HH31 = chan0[2][kk];
8	HH32 = chan1[2][kk];
9	HH41 = chan0[3][kk];
10	HH42 = chan1[3][kk];
11	H11_H21 = _itoll(HH11, HH21);
12	H12_H22 = _itoll(HH12, HH22);
13	H31_H41 = _itoll(HH31, HH41);
14	H32_H42 = _itoll(HH32, HH42);
15	
16	AA11_40 = (float) noiseVar + (float) ((Int32) _dotp4h(H11_H21, H11_H21));
17	result_0 = _dccmpy(H12_H22, H11_H21);
18	Ain1 = _dintsp(_dadd(_hi128(result_0), _lo128(result_0)));
19	
20	RR1 = _amem4(&recDataBuffPtr[0][kk + allocDesc[alloc].init_k0]);
21	RR2 = _amem4(&recDataBuffPtr[1][kk + allocDesc[alloc].init_k0]);
22	y1_y2 = _itoll(RR1, RR2);
23	
24	result_1 = _dccmpy(H11_H21, y1_y2);
25	Bin1 = _dintsp(_dadd(_hi128(result_1), _lo128(result_1)));
26	
27	result_2 = _dccmpy(H12_H22, y1_y2);
28	Bin2 = _dintsp(_dadd(_hi128(result_2), _lo128(result_2)));
29	AA22_40 = (float) noiseVar + (float) ((Int32) _dotp4h(H12_H22, H12_H22));
30	
31	AA11_40 = AA11_40 + (float) ((Int32) _dotp4h(H31_H41, H31_H41));
32	result_0 = _dccmpy(H32_H42, H31_H41);
33	Ain1 = _daddsp(Ain1, _dintsp(_dadd(_hi128(result_0), _lo128(result_0))));
34	
35	RR3 = _amem4(&recDataBuffPtr[2][kk + allocDesc[alloc].init_k0]);
36	RR4 = _amem4(&recDataBuffPtr[3][kk + allocDesc[alloc].init_k0]);
37	y3_y4 = _itoll(RR3, RR4);
38	
39	result_1 = _dccmpy(H31_H41, y3_y4);
40	Bin1 = _daddsp(Bin1, _dintsp(_dadd(_hi128(result_1), _lo128(result_1))));

Table 12 Register Pressure Analysis - QMPYSP Versus Dual DMPYSP (Part 2 of 2)

Line	Code
41	
42	result_2 = _dccmpy(H32_H42, y3_y4);
43	Bin2 = _daddsp(Bin2, _dintsp(_dadd(_hi128(result_2), _lo128(result_2))));
44	AA22_40 = AA22_40 + (float) ((Int32) _dotp4h(H32_H42, H32_H42));
45	
46	dtemp = _dmpysp(Ain1, Ain1);
47	iDenom = AA22_40 * AA11_40 - _hif(dtemp) - _lof(dtemp);
48	bb = (float)_rcpsp((float)iDenom);
49	bb = bb * (2.f - iDenom * bb); /* 16-bit precision */
50	llrDenom0 = llrDenom0 + bb * AA22_40;
51	llrDenom1 = llrDenom1 + bb * AA11_40;
52	
53	result0 = _cmpysp(Ain1, Bin2); //conj(H'Y/(H'H+sigmal))
54	dtemp = _dsubsp(_hid128(result0), _lod128(result0));
55	iNumer0 = _dsubsp(_dmpysp(_ftod(AA22_40, AA22_40), Bin1), dtemp);
56	result1 = _cmpysp(Ain1, Bin1);
57	dtemp = _daddsp(_hid128(result1), _lod128(result1));
58	iNumer1 = _dsubsp(_dmpysp(_ftod(AA11_40, AA11_40), Bin2), dtemp);
59	/* Compute LLR scaling factor denominator */
60	bb = bb * (float) outRms;
61	dtemp = _ftod(bb, bb);
62	DDhat0 = _dmpysp(dtemp, iNumer0);
63	DDhat1 = _dmpysp(dtemp, iNumer1);
64	Out1 = _dspinth(DDhat0);
65	Out1 = _loll(_dapys2(0x0001FFFF, Out1));
66	Out2 = _dspinth(DDhat1);
67	Out2 = _loll(_dapys2(0x0001FFFF, Out2));
68	eqOut0[kk] = Out1;
69	eqOut1[kk] = Out2;
70	}
End of Table 12	

Compile the updated function above with –o3 –s –mw –mv6600, which is implicitly little-endian. Following is the optimizer feedback and single iteration of the loop.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 890
;*      Loop opening brace source line   : 891
;*      Loop closing brace source line   : 968
;*      Known Minimum Trip Count         : 12
;*      Known Maximum Trip Count         : 1296
;*      Known Max Trip Count Factor      : 12
;*      Loop Carried Dependency Bound(^) : 3
;*      Unpartitioned Resource Bound     : 11
;*      Partitioned Resource Bound(*)    : 12
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     2        0
```

```
;*      .S units                     2        2
;*      .D units                     7        7
;*      .M units                    11       12*
;*      .X cross paths              10        8
;*      .T address paths             8        6
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)          16       18     (.L or .S unit)
;*      Addition ops (.LSD)          3        7     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            10       10
;*      Bound(.L .S .D .LS .LSD)    10       12*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 12 Register is live too long
;*         ii = 12 Did not find schedule
;*         ii = 13 Register is live too long
;*         ii = 13 Register is live too long
;*         ii = 13 Did not find schedule
;*         ii = 14 Cannot allocate machine registers
;*                   Regs Live Always   : 12/8  (A/B-side)
;*                   Max Regs Live      : 30/35
;*                   Max Cond Regs Live :  1/0
;*         ii = 14 Register is live too long
;*         ii = 14 Register is live too long
;*         ii = 14 Did not find schedule
;*         ii = 15 Cannot allocate machine registers
;*                   Regs Live Always   : 12/8  (A/B-side)
;*                   Max Regs Live      : 37/30
;*                   Max Cond Regs Live :  1/0
;*         ii = 15 Cannot allocate machine registers
;*                   Regs Live Always   : 12/8  (A/B-side)
;*                   Max Regs Live      : 36/33
;*                   Max Cond Regs Live :  1/0
;*         ii = 15 Did not find schedule
;*         ii = 16 Schedule found with 4 iterations in parallel
;*
;*       SINGLE SCHEDULED ITERATION
;*
;*       $C$C993:
;*   0              LDW     .D1T1   *A3++,A18         ; |910|
;*   1              LDW     .D1T1   *A22++,A19        ; |910|
;*     ||           LDW     .D2T2   *B23++,B18        ; |926|
;*   2              LDW     .D1T1   *A23++,A8         ; |910|
;*     ||           LDW     .D2T2   *B25++,B19        ; |926|
;*   3              LDW     .D1T1   *A28++,A9         ; |910|
;*   4              LDW     .D1T1   *A29++,A24        ; |917|
;*     ||           LDW     .D2T2   *B26++,B28        ; |927|
;*   5              LDW     .D1T1   *A30++,A25        ; |917|
;*     ||           LDW     .D2T2   *B0++,B29         ; |927|
;*   6              NOP             1
;*   7              DOTP4H  .M1     A19:A18,A19:A18,A4 ; |926|
;*     ||           LDW     .D2T2   *B1++,B30         ; |934|
;*   8              DOTP4H  .M2     B19:B18,B19:B18,B6 ; |926|
;*     ||           LDW     .D2T2   *B2++,B31         ; |934|
;*     ||           DOTP4H  .M1     A9:A8,A9:A8,A4    ; |939|
;*   9              DCCMPY  .M1     A9:A8,A19:A18,A11:A10:A9:A8 ; |910|
;*  10              DCCMPY  .M1     A9:A8,A25:A24,A11:A10:A9:A8 ; |920|
;*     ||           DCCMPY  .M2     B29:B28,B19:B18,B7:B6:B5:B4 ; |927|
;*  11              DCCMPY  .M1     A19:A18,A25:A24,A19:A18:A17:A16 ; |917|
;*     ||           INTSP   .S1     A4,A6             ; |926|
;*  12              INTSP   .L2     B6,B28            ; |926|
;*     ||           INTSP   .L1     A4,A24            ; |939|
;*     ||           DOTP4H  .M2     B29:B28,B29:B28,B4 ; |939|
;*  13              DADD    .S1     A11:A10,A9:A8,A5:A4 ; |911|
;*     ||           DCCMPY  .M2     B29:B28,B31:B30,B7:B6:B5:B4 ; |937|
;*  14              DINTSP  .S1     A5:A4,A9:A8       ; |911|
;*     ||           DCCMPY  .M2     B19:B18,B31:B30,B11:B10:B9:B8 ; |934|
;*     ||           DADD    .S2     B7:B6,B5:B4,B5:B4 ; |944|
;*  15              DADD    .S1     A11:A10,A9:A8,A9:A8 ; |921|
;*     ||           FADDSP  .L1     A6,A31,A24        ; |926|
;*     ||           DINTSP  .L2     B5:B4,B19:B18     ; |944|
;*  16              DINTSP  .S1     A9:A8,A7:A6       ; |921|
;*     ||           FADDSP  .L1     A24,A31,A25       ; |939|
;*     ||           INTSP   .L2     B4,B9             ; |939|
;*  17              NOP             1
;*  18              DADD    .S2     B7:B6,B5:B4,B7:B6 ; |948|
;*     ||           DADD    .L2     B11:B10,B9:B8,B5:B4 ; |951|
;*  19              DADD    .S1     A19:A18,A17:A16,A9:A8 ; |918|
;*     ||           FADDSP  .L2X    B28,A24,B21       ; |926|
;*     ||           DADDSP  .L1X    A9:A8,B19:B18,A17:A16 ; |944|
;*  20              FADDSP  .L2X    B9,A25,B20        ; |939|
```

```
;*  21              DINTSP  .L2     B7:B6,B7:B6       ; |948|
;*  22              DINTSP  .S2X    A9:A8,B9:B8       ; |918|
;*      ||          DMPYSP  .M1     A17:A16,A17:A16,A27:A26 ; |944|
;*      ||          DINTSP  .L2     B5:B4,B5:B4       ; |951|
;*  23              MPYSP   .M2     B21,B20,B6        ; |944|
;*  24              NOP             1
;*  25              DADDSP  .S1X    A7:A6,B7:B6,A25:A24 ; |948|
;*      ||          DADDSP  .L2     B9:B8,B5:B4,B9:B8 ; |951|
;*  26              NOP             1
;*  27              FSUBSP  .L2X    B6,A27,B24        ; |944|
;*  28              CMPYSP  .M1     A17:A16,A25:A24,A7:A6:A5:A4 ; |948|
;*  29              MV      .S2     B21,B29           ; |967|
;*      ||          MV      .D2     B21,B28           ; |967|
;*  30              FSUBSP  .L2X    B24,A26,B27       ; |944|
;*      ||          CMPYSP  .M1X    A17:A16,B9:B8,A7:A6:A5:A4 ; |951|
;*  31              DMPYSP  .M1X    B29:B28,A25:A24,A5:A4 ; |967|
;*  32              MV      .S2     B20,B19           ; |966|
;*      ||          MV      .D2     B20,B18           ; |966|
;*  33              RCPSP   .S2     B27,B24           ; |944|
;*      ||          DSUBSP  .L1     A7:A6,A5:A4,A17:A16 ; |966|
;*      ||          DMPYSP  .M2     B19:B18,B9:B8,B9:B8 ; |966|
;*  34              MPYSP   .M2     B24,B27,B6        ; |944|
;*      ||          DADDSP  .L1     A7:A6,A5:A4,A25:A24 ; |967|
;*  35              MVD     .M2     B20,B22           ; |939| Split a long life
;*  36              NOP             1
;*  37              MVD     .M2     B21,B27           ; |926| Split a long life
;*      ||          DSUBSP  .S2X    B9:B8,A17:A16,B17:B16 ; |966|
;*      ||          DSUBSP  .L1     A5:A4,A25:A24,A21:A20 ; |967|
;*  38              NOP             1
;*  39              FSUBSP  .L2     B3,B6,B5          ; |944|
;*  40              NOP             3
;*  43              MPYSP   .M2     B24,B5,B24        ; |944|
;*  44              NOP             3
;*  47              MPYSP   .M2     B24,B12,B8        ; |966|
;*  48              MPYSP   .M2     B27,B24,B8        ; |946|
;*  49              MV      .S1X    B24,A25           ; |944| Define a twin register
;*  50              MPYSP   .M1X    B22,A25,A4        ; |945|
;*  51              MV      .D2     B8,B19            ; |966|
;*      ||          MV      .S2     B8,B18            ; |966|
;*  52              FADDSP  .S2     B8,B13,B13        ; |946|   ^
;*      ||          DMPYSP  .M2     B19:B18,B17:B16,B5:B4 ; |966|
;*  53              DMPYSP  .M1X    B19:B18,A21:A20,A5:A4 ; |967|
;*  54              FADDSP  .L1     A4,A14,A14        ; |945|  ^
;*  55              NOP             1
;*  56              DSPINTH .L2     B5:B4,B4          ; |966|
;*  57              DSPINTH .L1     A5:A4,A6          ; |967|
;*      ||  [ A0]   SUB     .D1     A0,1,A0           ; |890|
;*  58      [ A0]   B       .S1     $C$C993           ; |890|
;*  59              SHR     .S2     B4,31,B5          ; |966|
;*  60              SHR     .S1     A6,31,A7          ; |967|
;*  61              DAPYS2  .L1X    A13:A12,B5:B4,A19:A18 ; |966|
;*  62              STW     .D1T1   A18,*A15++        ; |966|
;*      ||          DAPYS2  .L1     A13:A12,A7:A6,A5:A4 ; |967|
;*  63              STW     .D1T1   A4,*A2++          ; |967|
;*  64              ; BRANCHCC OCCURS {$C$C993}       ; |890|
;*-----------------------------------------------------------------------------*
```

As can be seen, the M utilization went up to 23, but the iteration interval of the loop went down to ii=16 (a 36% improvement).

The optimum schedule is being approached, but there is still register pressure as well as a register live too long problem.

4.1.2 Resource Balance with SIMD Moves

As mentioned earlier, the C66x core has a SIMD move instruction and an analogous intrinsic available for use inline with C. Note that the existing data movement C intrinsics, _itoll, _ftod _ito128, _fto128, _dto128 and _llto128, get translated into non-SIMD MV instructions. Using these intrinsics heavily may increase the likelihood of the loop being bound on the .L, .S and .D units, with many MVs, and may possibly increase register pressure. It is recommended that the programmer experiment with replacing these non-SIMD data movement intrinsics with SIMD move intrinsics. The SIMD data movement intrinsics are _dmv for interger moves and _fdmv for

floating-point. Note that there is no hard and fast rule as to when to use either non-SIMD data movement intrinsics (_itoll, _ito128 and _llto128) or SIMD data movement intrinsics (_dmv and _fdmv). Some suggestions on when to use SIMD moves:

- Use SIMD move if you are duplicating a register into a register pair
- Use SIMD move when you are sure that those registers will not be used in the next cycle, otherwise it may negatively impact the performance.

Note—Care should be taken in using these SIMD moves as they may increase the dynamic length of the loop.

Using the above suggestions, the code in Table 13 has replaced _itoll at line 13 and 14 with _dmv intrinsic and _ftod intrinsic with _fdmv at line 61.

Table 13 Example of Using SIMD Move Instructions for Register Pairs (Part 1 of 2)

Line	Code
1	for(kk=0; kk<numDataSubCars; kk++)
2	{
3	HH11 = chan0[0][kk];
4	HH12 = chan1[0][kk];
5	HH21 = chan0[1][kk];
6	HH22 = chan1[1][kk];
7	HH31 = chan0[2][kk];
8	HH32 = chan1[2][kk];
9	HH41 = chan0[3][kk];
10	HH42 = chan1[3][kk];
11	H11_H21 = _itoll(HH11, HH21);
12	H12_H22 = _itoll(HH12, HH22);
13	H31_H41 = _dmv(HH31, HH41);
14	H32_H42 = _dmv(HH32, HH42);
15	
16	AA11_40 = (float) noiseVar + (float) ((Int32) _dotp4h(H11_H21, H11_H21));
17	result_0 = _dccmpy(H12_H22, H11_H21);
18	Ain1 = _dintsp(_dadd(_hi128(result_0), _lo128(result_0)));
19	
20	RR1 = _amem4(&recDataBuffPtr[0][kk + allocDesc[alloc].init_k0]);
21	RR2 = _amem4(&recDataBuffPtr[1][kk + allocDesc[alloc].init_k0]);
22	y1_y2 = _itoll(RR1, RR2);
23	
24	result_1 = _dccmpy(H11_H21, y1_y2);
25	Bin1 = _dintsp(_dadd(_hi128(result_1), _lo128(result_1)));
26	
27	result_2 = _dccmpy(H12_H22, y1_y2);
28	Bin2 = _dintsp(_dadd(_hi128(result_2), _lo128(result_2)));
29	AA22_40 = (float) noiseVar + (float) ((Int32) _dotp4h(H12_H22, H12_H22));
30	
31	AA11_40 = AA11_40 + (float) ((Int32) _dotp4h(H31_H41, H31_H41));
32	result_0 = _dccmpy(H32_H42, H31_H41);

Table 13 Example of Using SIMD Move Instructions for Register Pairs (Part 2 of 2)

Line	Code
33	Ain1 = _daddsp(Ain1, _dintsp(_dadd(_hi128(result_0), _lo128(result_0))));
34	
35	RR3 = _amem4(&recDataBuffPtr[2][kk + allocDesc[alloc].init_k0]);
36	RR4 = _amem4(&recDataBuffPtr[3][kk + allocDesc[alloc].init_k0]);
37	y3_y4 = _itoll(RR3, RR4);
38	
39	result_1 = _dccmpy(H31_H41, y3_y4);
40	Bin1 = _daddsp(Bin1, _dintsp(_dadd(_hi128(result_1), _lo128(result_1))));
41	
42	result_2 = _dccmpy(H32_H42, y3_y4);
43	Bin2 = _daddsp(Bin2, _dintsp(_dadd(_hi128(result_2), _lo128(result_2))));
44	AA22_40 = AA22_40 + (float) ((Int32) _dotp4h(H32_H42, H32_H42));
45	
46	dtemp = _dmpysp(Ain1, Ain1);
47	iDenom = AA22_40 * AA11_40 - _hif(dtemp) - _lof(dtemp);
48	bb = (float)_rcpsp((float)iDenom);
49	bb = bb * (2.f - iDenom * bb); /* 16-bit precision */
50	llrDenom0 = llrDenom0 + bb * AA22_40;
51	llrDenom1 = llrDenom1 + bb * AA11_40;
52	
53	result0 = _cmpysp(Ain1, Bin2); //conj(H'Y/(H'H+sigmal))
54	dtemp = _dsubsp(_hid128(result0), _lod128(result0));
55	iNumer0 = _dsubsp(_dmpysp(_ftod(AA22_40, AA22_40), Bin1), dtemp);
56	result1 = _cmpysp(Ain1, Bin1);
57	dtemp = _daddsp(_hid128(result1), _lod128(result1));
58	iNumer1 = _dsubsp(_dmpysp(_ftod(AA11_40, AA11_40), Bin2), dtemp);
59	/* Compute LLR scaling factor denominator */
60	bb = bb * (float) outRms;
61	dtemp = _fdmv(bb, bb);
62	DDhat0 = _dmpysp(dtemp, iNumer0);
63	DDhat1 = _dmpysp(dtemp, iNumer1);
64	Out1 = _dspinth(DDhat0);
65	Out1 = _loll(_dapys2(0x0001FFFF, Out1));
66	Out2 = _dspinth(DDhat1);
67	Out2 = _loll(_dapys2(0x0001FFFF, Out2));
68	eqOut0[kk] = Out1;
69	eqOut1[kk] = Out2;
70	}
End of Table 13	

Compile the updated function above with –o3 –s –mw –mv6600, which is implicitly little-endian. Following is the optimizer feedback and single iteration of the loop.

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 890
;*      Loop opening brace source line   : 891
;*      Loop closing brace source line   : 968
;*      Known Minimum Trip Count         : 12
;*      Known Maximum Trip Count         : 1296
;*      Known Max Trip Count Factor      : 12
;*      Loop Carried Dependency Bound(^) : 12
;*      Unpartitioned Resource Bound     : 11
;*      Partitioned Resource Bound(*)    : 12
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        2
;*      .S units                     2        2
;*      .D units                     7        7
;*      .M units                    12*      11
;*      .X cross paths               7        9
;*      .T address paths             6        8
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)          18       18     (.L or .S unit)
;*      Addition ops (.LSD)          8        2     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            10       11
;*      Bound(.L .S .D .LS .LSD)    12*      11
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 12 Did not find schedule
;*         ii = 13 Cannot allocate machine registers
;*                   Regs Live Always   : 10/10 (A/B-side)
;*                   Max Regs Live      : 32/40
;*                   Max Cond Regs Live :  0/1
;*         ii = 13 Did not find schedule
;*         ii = 14 Cannot allocate machine registers
;*                   Regs Live Always   : 10/10 (A/B-side)
;*                   Max Regs Live      : 39/41
;*                   Max Cond Regs Live :  0/1
;*         ii = 14 Cannot allocate machine registers
;*                   Regs Live Always   : 10/10 (A/B-side)
;*                   Max Regs Live      : 35/39
;*                   Max Cond Regs Live :  0/1
;*         ii = 14 Did not find schedule
;*         ii = 15 Schedule found with 5 iterations in parallel
;*----------------------------------------------------------------------------*
;*         SINGLE SCHEDULED ITERATION
;*
;*         $C$C85:
;*   0              LDW     .D2T2   *B1++,B27         ; |740|
;*   1              NOP             1
;*   2              LDW     .D1T2   *A31++,B28        ; |750|
;*   3              LDW     .D2T1   *B31++,A8         ; |733|
;*   4              LDW     .D1T1   *A27++,A9         ; |733|
;*   5              LDW     .D1T1   *A26++,A19        ; |733|
;*     ||           LDW     .D2T2   *B3++,B6          ; |750|
;*   6              LDW     .D2T1   *B30++,A18        ; |733|
;*     ||           LDW     .D1T2   *A29++,B8         ; |749|
;*   7              LDW     .D2T2   *B2++,B9          ; |749|
;*   8              LDW     .D1T2   *A28++,B26        ; |740|
;*     ||           LDW     .D2T1   *B10++,A22        ; |757|
;*   9              LDW     .D1T1   *A15++,A23        ; |757|
;*     ||           DOTP4H  .M1     A9:A8,A9:A8,A7    ; |762|
;*  10              DMV     .S2     B28,B6,B5:B4      ; |750|
;*  11              DOTP4H  .M1     A19:A18,A19:A18,A17 ; |749|
;*     ||           DOTP4H  .M2     B5:B4,B5:B4,B4    ; |762|
;*  12              DCCMPY  .M1     A9:A8,A19:A18,A19:A18:A17:A16 ; |733|
;*     ||           DOTP4H  .M2     B9:B8,B9:B8,B5    ; |749|
;*  13              DCCMPY  .M1X    A19:A18,B27:B26,A11:A10:A9:A8 ; |740|
;*     ||           DCCMPY  .M2     B5:B4,B9:B8,B7:B6:B5:B4 ; |750|
;*     ||           INTSP   .L1     A7,A22            ; |762|
;*  14              DCCMPY  .M2X    B5:B4,A23:A22,B7:B6:B5:B4 ; |760|
;*  15              INTSP   .S1     A17,A6            ; |749|
;*     ||           INTSP   .S2     B4,B28            ; |762|
;*  16              DCCMPY  .M1X    A9:A8,B27:B26,A7:A6:A5:A4 ; |743|
;*     ||           INTSP   .L2     B5,B26            ; |749|
;*     ||           DCCMPY  .M2X    B9:B8,A23:A22,B7:B6:B5:B4 ; |757|
;*  17              DADD    .S1     A19:A18,A17:A16,A17:A16 ; |734|
```

```
;*      ||              FADDSP  .L1     A22,A30,A16       ; |762|
;*      ||              DADD    .S2     B7:B6,B5:B4,B25:B24 ; |767|
;*  18                  DINTSP  .S1     A17:A16,A9:A8     ; |734|
;*      ||              DINTSP  .S2     B25:B24,B25:B24   ; |767|
;*  19                  DADD    .S1     A11:A10,A9:A8,A9:A8 ; |741|
;*      ||              FADDSP  .L1     A6,A30,A16        ; |749|
;*      ||              DADD    .S2     B7:B6,B5:B4,B21:B20 ; |772|
;*  20                  DINTSP  .L1     A9:A8,A17:A16     ; |741|
;*      ||              DINTSP  .L2     B21:B20,B25:B24   ; |772|
;*  21                  FADDSP  .L1X    B28,A16,A20       ; |762|  ^
;*      ||              DADD    .S2     B7:B6,B5:B4,B21:B20 ; |775|
;*  22                  FADDSP  .L1X    B26,A16,A21       ; |749|
;*      ||              DADDSP  .L2X    A9:A8,B25:B24,B21:B20 ; |767|
;*      ||              DINTSP  .S2     B21:B20,B5:B4     ; |775|
;*  23                  NOP             1
;*  24                  DADD    .S1     A7:A6,A5:A4,A23:A22 ; |744|
;*  25                  DMPYSP  .M2     B21:B20,B21:B20,B23:B22 ; |767|
;*      ||              MPYSP   .M1     A21,A20,A16       ; |767|  ^
;*      ||              DADDSP  .L2X    A17:A16,B5:B4,B29:B28 ; |775|
;*      ||              MV      .D1     A20,A6            ; |790|
;*      ||              MV      .S1     A21,A4            ; |791|
;*  26                  DINTSP  .L2X    A23:A22,B7:B6     ; |744|
;*      ||              MV      .D1     A21,A5            ; |791|
;*  27                  NOP             1
;*  28                  MV      .S1     A20,A7            ; |790|
;*  29                  DADDSP  .L2     B7:B6,B25:B24,B27:B26 ; |772|
;*  30                  FSUBSP  .L1X    A16,B23,A19       ; |767|  ^
;*      ||              DMPYSP  .M2X    A7:A6,B29:B28,B9:B8 ; |790|
;*  31                  NOP             1
;*  32                  CMPYSP  .M2     B21:B20,B27:B26,B23:B22:B21:B20 ; |772|
;*  33                  FSUBSP  .L1X    A19,B22,A3        ; |767|  ^
;*      ||              CMPYSP  .M2     B21:B20,B29:B28,B7:B6:B5:B4 ; |775|
;*  34                  DMPYSP  .M2X    A5:A4,B27:B26,B23:B22 ; |791|
;*  35                  NOP             1
;*  36                  RCPSP   .S1     A3,A25            ; |767|  ^
;*      ||              DSUBSP  .L2     B23:B22,B21:B20,B5:B4 ; |790|
;*  37                  MV      .D1     A21,A24           ; |749|  Split a long life
;*      ||              MPYSP   .M1     A25,A3,A5         ; |767|  ^
;*  38                  MV      .S1     A20,A3            ; |762|  ^ Split a long life
;*      ||              DADDSP  .L2     B7:B6,B5:B4,B7:B6 ; |791|
;*  39                  DSUBSP  .L2     B9:B8,B5:B4,B19:B18 ; |790|
;*  40                  NOP             1
;*  41                  FSUBSP  .L1     A1,A5,A17         ; |767|  ^
;*      ||              DSUBSP  .S2     B23:B22,B7:B6,B17:B16 ; |791|
;*  42                  NOP             2
;*  44                  MPYSP   .M1     A25,A17,A18       ; |767|  ^
;*  45                  NOP             3
;*  48                  MPYSP   .M1     A18,A2,A17        ; |790|
;*  49                  MPYSP   .M1     A24,A18,A3        ; |770|
;*  50                  MPYSP   .M1     A3,A18,A21        ; |769|  ^
;*  51                  NOP             1
;*  52                  DMV     .S1     A17,A17,A19:A18   ; |790|
;*  53                  FADDSP  .L1     A3,A14,A14        ; |770|
;*      ||              DMPYSP  .M1X    A19:A18,B19:B18,A17:A16 ; |790|
;*  54                  FADDSP  .L1     A21,A13,A13       ; |769|
;*      ||              DMPYSP  .M2X    A19:A18,B17:B16,B7:B6 ; |791|
;*  55                  NOP             2
;*  57                  DSPINTH .L1     A17:A16,A25       ; |790|
;*  58                  DSPINTH .L2     B7:B6,B4          ; |791|
;*      ||  [ B0]   SUB     .D2     B0,1,B0           ; |713|
;*  59      [ B0]   B       .S2     $C$C85            ; |713|
;*  60                  MV      .D1     A25,A6            ; |790|
;*  61                  SHR     .S1     A25,31,A7         ; |790|
;*      ||              MV      .D2     B4,B8             ; |791|
;*      ||              SHR     .S2     B4,31,B9          ; |791|
;*  62                  DAPYS2  .L2     B13:B12,B9:B8,B9:B8 ; |791|
;*  63                  DAPYS2  .L2X    B13:B12,A7:A6,B21:B20 ; |790|
;*      ||              STW     .D1T2   B8,*A12++         ; |791|
;*  64                  STW     .D2T2   B20,*B11++        ; |790|
;*  65                  ; BRANCHCC OCCURS {$C$C85}        ; |713|
```

As can be seen, replacing some of the non-SIMD moves with SIMD Move intrinsics improved the iteration interval of the loop to ii = 15 cycles (40% improvement). Note that this is a completely *hit and miss*-based experiment and may or may not help in reducing resource bound computations as well as register pressure, but it is recommended to perform these experiments to gauge their effect.

4.2 C66x Limitation on Common Sub Expressions (CSE)

In the TMS320C66x compiler, the expressions resulting in the data type __x128_t do not factor into common sub-expressions. Therefore, do not use expressions for the intrinsics that result in a __x128_t data type as this may result in a performance reduction as shown in the code below. Note that this will not affect the functionality of the loop, but will cause the performance reduction.

The following code uses an expression that has a CMPYSP instruction that results in a _x128_t data type.

```
void dprod_vcse(double *restrict inputPtr,double *restrict coefsPtr,int
nCoefs,double *restrict sumPtr, int nlength) {
    int i, j;
    double sumTemp = 0, sumTemp1 = 0, sumTemp2 = 0, sumTemp3 = 0;
    for(i = 0; i<nlength/4; i++)
    {
    for (j = 0; j < nCoefs; j++)
    {
        sumTemp  =  _daddsp(sumTemp,
_daddsp(_hid128(_cmpysp(inputPtr[i],coefsPtr[i])),
                                        _lod128(_cmpysp(inputPtr[i],coefsPtr[i]))));
        sumTemp1 =  _daddsp(sumTemp1,
_daddsp(_hid128(_cmpysp(inputPtr[i+1],coefsPtr[i])),

_lod128(_cmpysp(inputPtr[i+1],coefsPtr[i]))));
        sumTemp2  = _daddsp(sumTemp2,
_daddsp(_hid128(_cmpysp(inputPtr[i+2],coefsPtr[i])),

_lod128(_cmpysp(inputPtr[i+2],coefsPtr[i]))));
        sumTemp3 =  _daddsp(sumTemp3,
_daddsp(_hid128(_cmpysp(inputPtr[i+3],coefsPtr[i])),

_lod128(_cmpysp(inputPtr[i+3],coefsPtr[i]))));
        }
        sumPtr[i] = sumTemp;
        sumPtr[i+1] = sumTemp1;
        sumPtr[i+2] = sumTemp2;
        sumPtr[i+3] = sumTemp3;
    }
}
```

Compile the function above with –o3 –s –mw –mv6600, which is implicitly little-endian. As can be seen from the optimizer comments, M unit utilization is 8.

Note that there are eight CMPYSP instructions instead of four CMPYSP instructions in the single iteration of the loop.

```
$C$DW$L$dprod_vcse$4$E:
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 1575
;*      Loop opening brace source line   : 1576
;*      Loop closing brace source line   : 1585
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 3
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 4
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        0
;*      .D units                     0        0
;*      .M units                     4*       4*
;*      .X cross paths               0        4*
;*      .T address paths             0        0
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           4        4     (.L or .S unit)
;*      Addition ops (.LSD)          0        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        2
```

```
;*      Bound(.L .S .D .LS .LSD)     2        2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Schedule found with 3 iterations in parallel
;*
;*        SINGLE SCHEDULED ITERATION
;*
;*        $C$C29:
;*   0              CMPYSP  .M2X    B21:B20,A17:A16,B19:B18:B17:B16 ; |1577|
;*     ||           CMPYSP  .M1     A19:A18,A17:A16,A11:A10:A9:A8 ; |1579|
;*   1              CMPYSP  .M1     A19:A18,A17:A16,A7:A6:A5:A4 ; |1579|
;*     ||           CMPYSP  .M2X    B23:B22,A17:A16,B7:B6:B5:B4 ; |1581|
;*   2              CMPYSP  .M2X    B23:B22,A17:A16,B11:B10:B9:B8 ; |1581|
;*     ||           CMPYSP  .M1     A21:A20,A17:A16,A7:A6:A5:A4 ; |1583|
;*   3              CMPYSP  .M2X    B21:B20,A17:A16,B7:B6:B5:B4 ; |1577|
;*     ||           CMPYSP  .M1     A21:A20,A17:A16,A11:A10:A9:A8 ; |1583|
;*   4              NOP             1
;*   5              DADDSP  .S1     A7:A6,A9:A8,A5:A4 ; |1579|
;*   6              DADDSP  .L2     B11:B10,B5:B4,B9:B8 ; |1581|
;*   7              DADDSP  .L2     B7:B6,B17:B16,B7:B6 ; |1577|
;*     ||           DADDSP  .L1     A11:A10,A5:A4,A9:A8 ; |1583|
;*   8              DADDSP  .L1     A23:A22,A5:A4,A23:A22 ; |1579|  ^
;*   9              DADDSP  .L2     B27:B26,B9:B8,B27:B26 ; |1581|  ^
;*  10              DADDSP  .S2     B25:B24,B7:B6,B25:B24 ; |1577|  ^
;*     ||           DADDSP  .L1     A25:A24,A9:A8,A25:A24 ; |1583|  ^
;*     ||           SPBR            $C$C29
;*  11              NOP             1
;*  12              ; BRANCHCC OCCURS {$C$C29}       ; |1575|
```

The C code below removes the expression from CMPYSP instructions (CMPYSP results in __x128_t data type).

```
void dprod_novcse(double *restrict inputPtr,double *restrict coefsPtr,int
nCoefs,double *restrict sumPtr, int nlength) {
    int i, j;
    double sumTemp = 0, sumTemp1 = 0, sumTemp2 = 0, sumTemp3 = 0;
    __x128_t cmpysp_temp, cmpysp_temp1, cmpysp_temp2, cmpysp_temp3;

    for(i = 0; i<nlength/4; i++)
    {
    for (j = 0; j < nCoefs; j++)
    {
        cmpysp_temp = _cmpysp(inputPtr[i],coefsPtr[i]);
        sumTemp  =  _daddsp(sumTemp, _daddsp(_hid128(cmpysp_temp),
_lod128(cmpysp_temp)));
        cmpysp_temp1 = _cmpysp(inputPtr[i+1],coefsPtr[i]);
        sumTemp1 =  _daddsp(sumTemp1, _daddsp(_hid128(cmpysp_temp1),
_lod128(cmpysp_temp1)));
        cmpysp_temp2 = _cmpysp(inputPtr[i+2],coefsPtr[i]);
        sumTemp2  = _daddsp(sumTemp2, _daddsp(_hid128(cmpysp_temp2),
_lod128(cmpysp_temp2)));
        cmpysp_temp3 = _cmpysp(inputPtr[i+3],coefsPtr[i]);
        sumTemp3 =  _daddsp(sumTemp3, _daddsp(_hid128(cmpysp_temp3),
_lod128(cmpysp_temp3)));
        }
        sumPtr[i]   = sumTemp;
        sumPtr[i+1] = sumTemp1;
        sumPtr[i+2] = sumTemp2;
        sumPtr[i+3] = sumTemp3;
    }

}
```

Compile the updated function above with –o3 –s –mw –mv6600, which is implicitly little-endian. As can be seen from the optimizer comments, the loop is still M-bound, but utilization is reduced by 50%. There are only four CMPYSP instructions used in a single iteration, which matches with the C code.

```
$C$DW$L$dprod_novcse$4$E:
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
```

```
;*      Loop source line                 : 1600
;*      Loop opening brace source line   : 1601
;*      Loop closing brace source line   : 1610
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 3
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        0
;*      .D units                     0        0
;*      .M units                     2*       2*
;*      .X cross paths               2*       0
;*      .T address paths             0        0
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           4        4     (.L or .S unit)
;*      Addition ops (.LSD)          0        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2*       2*
;*      Bound(.L .S .D .LS .LSD)     2*       2*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 3 iterations in parallel
;*

;*        SINGLE SCHEDULED ITERATION
;*
;*        $C$C91:
;*   0              CMPYSP  .M2X    B9:B8,A7:A6,B23:B22:B21:B20 ; |1602|
;*     ||           CMPYSP  .M1     A17:A16,A7:A6,A11:A10:A9:A8 ; |1604|
;*   1              CMPYSP  .M2X    B17:B16,A7:A6,B7:B6:B5:B4 ; |1606|
;*     ||           CMPYSP  .M1     A19:A18,A7:A6,A11:A10:A9:A8 ; |1608|
;*   2              NOP             2
;*   4              DADDSP  .L2     B23:B22,B21:B20,B5:B4 ; |1603|
;*     ||           DADDSP  .L1     A11:A10,A9:A8,A5:A4 ; |1605|
;*   5              DADDSP  .L2     B7:B6,B5:B4,B19:B18 ; |1607|
;*     ||           DADDSP  .L1     A11:A10,A9:A8,A5:A4 ; |1609|
;*   6              NOP             1
;*   7              DADDSP  .S2     B19:B18,B5:B4,B19:B18 ; |1603|  ^
;*     ||           DADDSP  .S1     A21:A20,A5:A4,A21:A20 ; |1605|  ^
;*   8              DADDSP  .S2     B25:B24,B19:B18,B25:B24 ; |1607|  ^
;*     ||           DADDSP  .S1     A23:A22,A5:A4,A23:A22 ; |1609|  ^
;*     ||           SPBR            $C$C91
;*   9              ; BRANCHCC OCCURS {$C$C91}        ; |1600|
```

4.3 Live-Too-Long Problem in C6000 C Code

Live-too-long is the classic problem in DSP code. It causes a dependency in non-adjacent stages of a pipeline. It occurs in algorithms that need to store a result before moving on. It results in a static stall in a code. Note that this is different than the register live-too-long problem seen in the previous section. The programmer must identify any live-too-long problems in the C code and attempt to use the solutions shown in the following examples.

In this section, a high-level overview of the live-too-long problem is explained as a refresher, for details see section 5.10 of the *TMS320C6000 Programmer's Guide* [3].

In Figure 3, the left side flow diagram shows a feedback path between D and A. This feedback path creates a dependency between the successive iterations. As shown at the right side of the figure, one of the possible solutions is to duplicate the feedback path and optimize it.

Figure 3 Duplicate the Feedback Path And Optimize

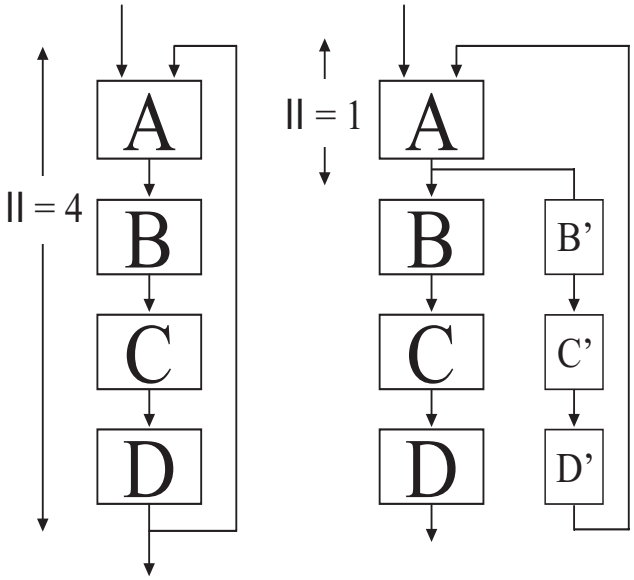
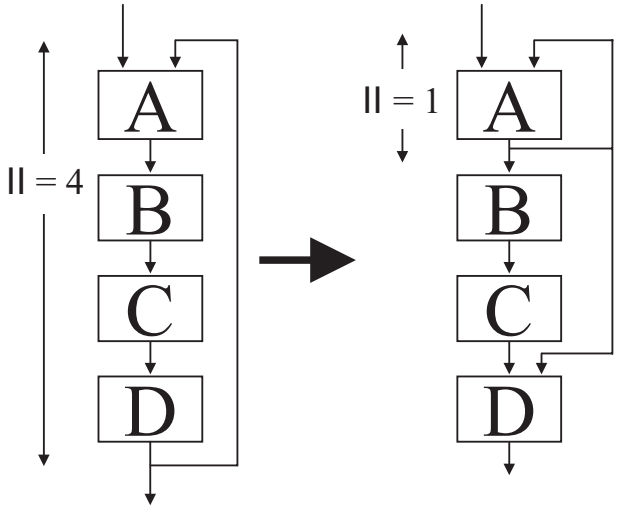


Figure 4 shows another solution to solve this dependence path and that is to copy a value and forward it as shown in the right side of the figure.

Figure 4 Copy and Forward Method



5 Summary

This application report describes instruction-level optimization techniques for the C66x architecture. This report shows how to do the following:

- Use the new 128-bit operand instructions
- Optimizations that take advantage of floating point instructions
- Techniques for resolving register pressure
- Techniques for solving non-adjacent pipeline stage problems

The examples have been intentionally simplified so that the methods are easy to understand.

6 References

1 *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187).
2 *C66x CPU and ISA Reference Guide* (SPRUGH7).
3 *TMS320C6000 Programmer's Guide* (SPRU198).
4 *Hand-Tuning Loops and Control Code on the TMS320C6000* (SPRA666)

Trademarks

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265