KeyStone Training

KeyStone C66x CorePac
Instruction Set Architecture
Disclaimer

• This section describes differences between the TMS320C674x instruction set architecture and the TMS320C66x instruction set included in the KeyStone CorePac.

• Users of this training should already be familiar with the TMS320C674x CPU and Instruction Set Architecture.
Agenda

• Introduction
• Increased SIMD Capabilities
• C66x Floating-Point Capabilities
• Examples of New Instructions
• Matrix Multiply Example
Introduction

• Introduction
• Increased SIMD Capabilities
• C66x Floating-Point Capabilities
• Examples of New Instructions
• Matrix Multiply Example
Enhanced DSP Core

C66x CorePac
- 100% upward object code compatible
- 4x performance improvement for multiply operation
- 32 16-bit MACs
- Improved support for complex arithmetic and matrix computation

C64x+
- SPLOOP and 16-bit instructions for smaller code size
- Flexible level one memory architecture
- iDMA for rapid data transfers between local memories
- Four 16-bit or eight 8-bit MACs
- Two-level cache

C64x
- Advanced fixed-point instructions

C67x+
- 100% upward object code compatible with C64x, C64x+, C67x and c67x+
- Best of fixed-point and floating-point architecture for better system performance and faster time-to-market

C67x
- Native instructions for IEEE 754, SP&DP
- Advanced VLIW architecture
- 2x registers
- Enhanced floating-point add capabilities

Performance improvement
CPU Modifications

- Datapaths of the .L and .S units have been increased from 32-bit to 64-bit.
- Datapaths of the .M units have been increased from 64-bit to 128-bit.
- The cross-path between the register files has been increased from 32-bit to 64-bit.
- Register file quadruplets are used to create 128-bit values.
- No changes in D datapath.
Core Evolution – Unified Architecture

- Increased Performance
- Fixed/Floating Unification

C64x+

- C64x+ multiplier unit contains four 16-bit multipliers (per side)
- 16 fixed multiplies per cycle (per side)

C66x

- Four floating multiplies per cycle (per side)

Diagram Key
- .D = Data Unit
- .M = Multiplier Unit
- .L = Logical Unit
- .S = Shifter Unit
Increased Performance

• Floating-point and fixed-point performance is significantly increased.
  – 4x increased in the number of MAC
• Fixed point:
  – The core can now multiply up to 32 (16x16-bit) multiplies per cycle.
  – Eight complex MACs per cycle
• Floating point:
  – Eight single-precision multiplies per cycle
  – Four single-precision MACs per cycle
  – Two double-precision MACs per cycle
  – SIMD (Single Instruction Multiple Data) support
  – Additional resource flexibility (e.g. the INT to/from SP conversion operations can now be executed on .L and .S units).
• Optimized for complex arithmetic and linear algebra (matrix processing)
  – L1 and L2 processing is highly dominated by complex arithmetic and linear algebra (matrix processing).
## Performance Improvement Overview

<table>
<thead>
<tr>
<th></th>
<th>C64x+</th>
<th>C674x</th>
<th>C66x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed point 16x16 MACs per cycle</td>
<td>8</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Fixed point 32x32 MACs per cycle</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Floating point single-precision MACs per cycle</td>
<td>n/a</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Arithmetic floating-point operations per cycle</td>
<td>n/a</td>
<td>6&lt;sup&gt;1&lt;/sup&gt;</td>
<td>16&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Load/store width</td>
<td>2 x 64-bit</td>
<td>2 x 64-bit</td>
<td>2 x 64-bit</td>
</tr>
<tr>
<td>Vector size (SIMD capability)</td>
<td>32-bit (2 x 16-bit, 4 x 8-bit)</td>
<td>32-bit (2 x 16-bit, 4 x 8-bit)</td>
<td>128-bit&lt;sup&gt;3&lt;/sup&gt; (4 x 32-bit, 4 x 16-bit, 4 x 8-bit)</td>
</tr>
</tbody>
</table>

<sup>1</sup> One operation per .L, .S, .M units for each side (A and B)

<sup>2</sup> Two-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).

<sup>3</sup> 128-bit SIMD for the M unit. 64-bit SIMD for the L and S units.
Increased SIMD Capabilities

• Introduction

• Increased SIMD Capabilities

• C66x Floating-Point Capabilities

• Examples of New Instructions

• Matrix Multiply Example
SIMD Instructions

• C64x and C674x support 32-bit SIMD:
  – 2 x 16-bit
    • Syntax: `<instruction_name>2 .<unit> <operand>`
    • Example: MPY2
  – 4 x 8-bit
    • Syntax: `<instruction_name>4 .<unit> <operand>`
    • Example: AVGU4

• C66x improves SIMD support:
  – Two-way SIMD version of existing instruction:
    • Syntax: `D<instruction_name> .<unit> <operand>`
    • Example: DMPY2, DADD
SIMD Instructions

- C66x supports various SIMD data types:
  - 2 x 16-bit
    - Two-way SIMD operations for 16-bit elements
      - Example: ADD2
  - 2 x 32-bit
    - Two-way SIMD operations for 32-bit elements
      - Example: DSUB
    - Two-way SIMD operations for complex (16-bit I / 16-bit Q) elements
      - Example: DCMPY
    - Two-way SIMD operations for single-precision floating elements
      - Example: DMPYS
  - 4 x 16-bit
    - Four-way SIMD operations for 16-bit elements
      - Example: DMAX2, DADD2
  - 4 x 32-bit
    - Four-way SIMD operations for 32-bit elements
      - Example: QMPY32R1
    - Four-way SIMD operations for complex (16-bit I / 16-bit Q) elements
      - Example: CMATMPY
    - Four-way SIMD operations for single-precision floating elements
      - Example: QMPYSP
  - 8 x 8-bit
    - Eight-way SIMD operations for 8-bit elements
      - Example: DMINU4
SIMD Operations (1/2)

• Same precision
  – Examples:
    • MAX2
    • DADD2
    • DCMPYR1

• Increased/narrowed precision
  – Example: DCMPY
SIMD Operations (2/2)

• Reduction
  – Example:
    • DCMPY, DDOTP4H

• Complex instructions
  – Example:
    • DDOTPxx, CMATMPY
Registers and Data Types

• Introduction
• Increased SIMD Capabilities

• Registers and Data Types

• C66x Floating-Point Capabilities
• Examples of New Instructions
• Matrix Multiply Example
Registers

- C66x provides a total of 64 32-bit registers, which are organized in two general purpose register files (A and B) of 32 registers each.

- Registers can be accessed as follows:
  - Registers (32-bit)
  - Register pairs (64-bit)
  - Register quads (128-bit)

- C66x provides explicit aliased views.

<table>
<thead>
<tr>
<th>Register File</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1:A0</td>
<td>B1:B0</td>
<td></td>
</tr>
<tr>
<td>A3:A2</td>
<td>B3:B2</td>
<td></td>
</tr>
<tr>
<td>A5:A4</td>
<td>B5:B4</td>
<td></td>
</tr>
<tr>
<td>A7:A6</td>
<td>B7:B6</td>
<td></td>
</tr>
<tr>
<td>A9:A8</td>
<td>B9:B8</td>
<td></td>
</tr>
<tr>
<td>A11:A10</td>
<td>B11:B10</td>
<td></td>
</tr>
<tr>
<td>A13:A12</td>
<td>B13:B12</td>
<td></td>
</tr>
<tr>
<td>A15:A14</td>
<td>B15:B14</td>
<td></td>
</tr>
<tr>
<td>A17:A16</td>
<td>B17:B16</td>
<td></td>
</tr>
<tr>
<td>A19:A18</td>
<td>B19:B18</td>
<td></td>
</tr>
<tr>
<td>A21:A20</td>
<td>B21:B20</td>
<td></td>
</tr>
<tr>
<td>A23:A22</td>
<td>B23:B22</td>
<td></td>
</tr>
<tr>
<td>A27:A26</td>
<td>B27:B26</td>
<td></td>
</tr>
<tr>
<td>A29:A28</td>
<td>B29:B28</td>
<td></td>
</tr>
<tr>
<td>A31:A30</td>
<td>B31:B30</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register File</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
</table>
The \texttt{__x128\_t} Container Type (1/2)

- To manipulate 128-bit vectors, a new data type has been created in the C compiler: \texttt{__x128\_t}.
- C compiler defines some intrinsic to create 128-bit vectors and to extract elements from a 128-bit vector.
The __x128_t Container Type (2/2)

• Example:

Extraction

\_get32\_128(src, 0)

\_hi128(src)

Creation

\_llto128(src1,src2)

\_ito128(src1,src2,src3, src4)

• Refer to the [TMS320C6000 Optimizing Compiler User Guide](https://www.ti.com) for a complete list of available intrinsics to create 128-bit vectors and extract elements from a 128-bit vector.
The `__float2_t` Container Type

- C66x ISA supports floating-point SIMD operations.
- `__float2_t` is a container type to store two single precision floats.
- On previous architectures (C67x, C674x), the `double` data type was used as a container for SIMD float numbers. While all old instructions can still use the `double` data type, all new C66x instructions will have to use the new data type: `__float2_t`.
- C compiler defines some intrinsic to create vectors of floating-point elements and to extract floating-point elements from a floating-point vector.

![Extraction and Creation Diagram]

<table>
<thead>
<tr>
<th>Extraction</th>
<th>Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_lof2(src)</code></td>
<td><code>_ftof2(src1,src2)</code></td>
</tr>
</tbody>
</table>
C66x Floating Point Capabilities

• Introduction
• Increased SIMD Capabilities
• Register

• C66x Floating-Point Capabilities
• Examples of New Instructions
• Matrix Multiply Example
Support for Floating Point in C66x

Floating point enables efficient MIMO processing and LTE scheduling:

- C66x core supports floating point at full clock speed resulting in 20 GFlops per core @ 1.2GHz.
- Floating point enables rapid algorithm prototyping and quick SW redesigns, thus there is no need for normalization and scaling.
- Use Case: LTE MMSE MIMO receiver kernel with matrix inversion
  - Performs up to 5x faster than fixed-point implementation
  - Significantly reduces development and debug cycle time

Floating point significantly reduces design cycle time with increased performance
C66x Floating-Point Compatibility

- C66x is 100% object code compatible with C674x.
- A new version of each basic floating-point instruction has been implemented.

<table>
<thead>
<tr>
<th></th>
<th>C674x</th>
<th>C66x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Delay Slots</td>
<td>Delay Slot</td>
</tr>
<tr>
<td>MPYSP</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>MPYDP</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>ADDSP / SUBSP</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>ADDDP/ SUBDP</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Functional Unit Latency</td>
<td>Functional Unit Latency</td>
</tr>
<tr>
<td>MPYSP</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MPYDP</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>ADDSP / SUBSP</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADDDP / SUBDP</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- The C compiler automatically selects the new C66x instructions.
- When writing in hand-coded assembly, the Fast version has to be specifically used.
  - FADDSP / FSUBSP / FMPYSP / FADDSP / FSUBSP / FMPYSP
C66x Floating Point

- C66x ISA includes a complex arithmetic multiply instruction, CMPYSP.
  - CMPYSP computes the four partial products of the complex multiply.
  - To complete a full complex multiply in floating point, the following code has to be executed:

```
```
Examples of New Instructions

• Introduction
• Increased SIMD Capabilities
• C66x Floating-Point Capabilities

• Examples of New Instructions
• Matrix Multiply Example
# New Instructions on .M Unit

<table>
<thead>
<tr>
<th>C/C++ Compiler Intrinsic</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__x128_t _dcmpy(long long src1, long long src2);</td>
<td>DCMPY</td>
<td>Two-way SIMD complex multiply operations on two sets of packed numbers.</td>
</tr>
<tr>
<td>__x128_t _dccmpy(long long src1, long long src2);</td>
<td>DCCMPY</td>
<td>Two-way SIMD complex multiply operations on two sets of packed numbers with complex conjugate of src2.</td>
</tr>
<tr>
<td>long long _dcmpy(long long src1, long long src2);</td>
<td>DCMPYR1</td>
<td>Two-way SIMD complex multiply operations on two sets of packed numbers with rounding.</td>
</tr>
<tr>
<td>long long _dccmpy(long long src1, long long src2);</td>
<td>DCCMPYR1</td>
<td>Two-way SIMD complex multiply operations on two sets of packed numbers with rounding and complex conjugate of src2.</td>
</tr>
<tr>
<td>__x128_t _cmatmpy(long long src1, __x128_t src2);</td>
<td>CMATMPY</td>
<td>Multiply a 1x2 vector by one 2x2 complex matrix, producing two 32-bit complex numbers.</td>
</tr>
<tr>
<td>__x128_t _ccmatmpy(long long src1, __x128_t src2);</td>
<td>CCMATMPY</td>
<td>Multiply the conjugate of a 1x2 vector by one 2x2 complex matrix, producing two 32-bit complex numbers.</td>
</tr>
<tr>
<td>long long _cmatmpyr1(long long src1, __x128_t src2);</td>
<td>CMATMPYR1</td>
<td>Multiply a 1x2 vector by one 2x2 complex matrix, producing two 32-bit complex numbers with rounding.</td>
</tr>
<tr>
<td>long long _ccmatmpyr1(long long src1, __x128_t src2);</td>
<td>CCMATMPYR1</td>
<td>Multiply the conjugate of a 1x2 vector by one 2x2 complex matrix, producing two 32-bit complex numbers with rounding.</td>
</tr>
<tr>
<td>__x128_t _dmpy2 (long long src1, long long src2);</td>
<td>DMPY2</td>
<td>Four-way SIMD multiply, packed signed 16-bit</td>
</tr>
<tr>
<td>__x128_t _dsmpy2 (long long src1, long long src2);</td>
<td>DSMPY2</td>
<td>Four-way SIMD multiply signed by signed with left shift and saturation, packed signed 16-bit</td>
</tr>
</tbody>
</table>
## New Instructions on .M Unit

<table>
<thead>
<tr>
<th>C/C++ Compiler Intrinsic</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long long _dxpnd2 (unsigned src);</td>
<td>DXPND2</td>
<td>Expands bits to packed 16-bit masks</td>
</tr>
<tr>
<td>long long _ccmpy32r1 (long long src1, long long src2);</td>
<td>CCMPY32R1</td>
<td>32-bit complex conjugate multiply of Q31 numbers with Rounding</td>
</tr>
<tr>
<td>__x128_t __qmpysp (__x128_t src1, __x128_t src2);</td>
<td>QMPYSP</td>
<td>Four-way SIMD 32-bit single precision multiply producing four 32-bit single precision results.</td>
</tr>
<tr>
<td>__x128_t __qmpy32 (__x128_t src1, __x128_t src2);</td>
<td>QMPY32</td>
<td>Four-way SIMD multiply of signed 32-bit values producing four 32-bit results. (Four-way _mpy32).</td>
</tr>
<tr>
<td>__x128_t __qsmtpy32r1 (__x128_t src1, __x128_t src2);</td>
<td>QSMTPY32R1</td>
<td>4-way SIMD fractional 32-bit by 32-bit multiply where each result value is shifted right by 31 bits and rounded. This normalizes the result to lie within -1 and 1 in a Q31 fractional number system.</td>
</tr>
</tbody>
</table>
# New Instructions on .L Unit

<table>
<thead>
<tr>
<th>C/C++ Compiler Intrinsic</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long long _dshr(long long src1, unsigned src2);</td>
<td>DSHR</td>
<td>Shift-right of two signed 32-bit values by a single value in the src2 argument.</td>
</tr>
<tr>
<td>long long _dshru(long long src1, unsigned src2);</td>
<td>DSHRU</td>
<td>Shift-right of two unsigned 32-bit values by a single value in the src2 argument.</td>
</tr>
<tr>
<td>long long _dshl(long long src1, unsigned src2);</td>
<td>DSHL</td>
<td>Shift-left of two signed 32-bit values by a single value in the src2 argument.</td>
</tr>
<tr>
<td>long long _dshr2(long long src1, unsigned src2);</td>
<td>DSHR2</td>
<td>Shift-right of four signed 16-bit values by a single value in the src2 argument (two way _shr2(), four way SHR).</td>
</tr>
<tr>
<td>long long _dshru2(long long src1, unsigned src2);</td>
<td>DSHRU2</td>
<td>Shift-right of four unsigned 16-bit values by a single value in the src2 argument (two way _shru2(), four way SHRU).</td>
</tr>
<tr>
<td>unsigned _shl2(unsigned src1, unsigned src2);</td>
<td>SHL2</td>
<td>Shift-left of two signed 16-bit values by a single value in the src2 argument.</td>
</tr>
<tr>
<td>long long _dshl2(long long src1, unsigned src2);</td>
<td>DSHL2</td>
<td>Shift-left of two signed 16-bit values by a single value in the src2 argument (two way _shl2(), four way SHL).</td>
</tr>
<tr>
<td>unsigned _dcmpgt2(long long src1, long long src2);</td>
<td>DCMPGT2</td>
<td>Four-way SIMD comparison of signed 16-bit values. Results are packed into the four least significant bits of the return value.</td>
</tr>
<tr>
<td>unsigned _dcmeq2(long long src1, long long src2);</td>
<td>DCMPEQ2</td>
<td>Four-way SIMD comparison of signed 16-bit values. Results are packed into the four least significant bits of the return value.</td>
</tr>
<tr>
<td>void _mfence();</td>
<td>MFENCE</td>
<td>Stall CPU while memory system is busy.</td>
</tr>
</tbody>
</table>
## New Instructions on .L/.S Unit

<table>
<thead>
<tr>
<th>C/C++ Compiler Intrinsic</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double _daddsp(double src1, double src2);</td>
<td>DADDSP</td>
<td>Two-way SIMD addition of 32-bit single precision numbers.</td>
</tr>
<tr>
<td>Double _dsubsp(double src1, double src2);</td>
<td>DSUBSP</td>
<td>Two-way SIMD subtraction of 32-bit single precision numbers.</td>
</tr>
<tr>
<td>__float2_t _dintsp(long long src);</td>
<td>DINTSP</td>
<td>Converts two 32-bit signed integers to two single-precision float point values.</td>
</tr>
<tr>
<td>long long _dspint (__float2_t src);</td>
<td>DSPINT</td>
<td>Converts two packed single-precision floating point values to two signed 32-bit values.</td>
</tr>
</tbody>
</table>
Other New Instructions

• For an exhaustive list of the C66x instructions, please refer to the Instruction Descriptions in the TMS320C66x DSP CPU and Instruction Set.

• For an exhaustive list of the new C66x instructions and their associated C intrinsics, please refer to the Vector-in-Scalar Support C/C++ Compiler v7.2 Intrinsics table in the TMS320C6000 Optimizing Compiler User Guide.
Matrix Multiply Example

• Introduction
• Increased SIMD Capabilities
• C66x Floating-Point Capabilities
• Examples of New Instructions

• Matrix Multiply Example
Matrix Multiply

for (i=0; i<NRA; i++)
{
    for (j=0; j<NCB; j++)
    {
        sum_16.real = 0;
        sum_16.imag = 0;
        for (k=0; k<NCA; k++)
        {
            sum_16.real += ((a[i][k].real * b[k][j].real) - (a[i][k].imag * b[k][j].imag)) + 0x00004000) >> 15;
            sum_16.imag += ((a[i][k].real * b[k][j].imag) + (a[i][k].imag * b[k][j].real)) + 0x00004000) >> 15;
        }
        c_ref[i][j].real = sum_16.real;
        c_ref[i][j].imag = sum_16.imag;
    }
}
Matrix Multiply

- CMATMPY instruction performs the basic operation:
  \[
  \begin{bmatrix}
  C_{11} & C_{12} \\
  \end{bmatrix} = \begin{bmatrix}
  A_{11} & A_{12} \\
  \end{bmatrix} \begin{bmatrix}
  B_{11} & B_{12} \\
  B_{21} & B_{22} \\
  \end{bmatrix}
  \]

- Multiple CMATMPY instructions can be used to compute larger matrices.
Matrix Multiply

- C66x C + intrinsic code:
  - Use of the __x128_t type
  - Use of some conversion intrinsic
  - Use of _cmatmpyr1() intrinsic

```c
for (k=0; k<nco; k++) {
    a_i_0 = *ptrA0++;   a_j1_i0 = *ptrAl++;
    a_i_2 = *ptrA2++;   a_j3_i2 = *ptrA3++;

    b_i_0 = _amem8((void *) &b_1 + k*nco);
    b_i_2 = _amem8((void *) &b_2 + k*nco + 2);
    b_k1_k0 = _amem8((void *) &b[1 + (k+1)*nco]);
    b_k3_k2 = _amem8((void *) &b[1 + (k+1)*nco + 2]);

    __x128_t b_k1_k0_b_1_0 = _l1to128(b_k1_k0, b_i_0);
    __x128_t b_k3_k2_b_3_2 = _l1to128(b_k3_k2, b_i_2);

    sum0 = _dsadd2(_cmatmpyr1(a_i_0), b_k1_k0_b_1_0, sum0);
    sum1 = _dsadd2(_cmatmpyr1(a_j1_i0), b_k1_k0_b_1_0, sum1);
    sum2 = _dsadd2(_cmatmpyr1(a_i_2), b_k1_k0_b_1_0, sum2);
    sum3 = _dsadd2(_cmatmpyr1(a_j3_i2), b_k1_k0_b_1_0, sum3);
    sum4 = _dsadd2(_cmatmpyr1(a_i_0), b_k3_k2_b_3_2, sum4);
    sum5 = _dsadd2(_cmatmpyr1(a_j1_i0), b_k3_k2_b_3_2, sum5);
    sum6 = _dsadd2(_cmatmpyr1(a_i_2), b_k3_k2_b_3_2, sum6);
    sum7 = _dsadd2(_cmatmpyr1(a_j3_i2), b_k3_k2_b_3_2, sum7);
}

} /* End of loop on Nca */
```
Matrix Multiply C66x Implementation Description

- C66x C + intrinsic code:

```c
for (k=0; k<nca; k+=2) {
  a_1_0 = *ptrA0++;   a_j1_j0 = *ptrA1++;
  a_3_2 = *ptrA2++;   a_j3_j2 = *ptrA3++;
  b_1_0 = _amem8((void *) &b[i + k*ncb]);
  b_3_2 = _amem8((void *) &b[i + k*ncb + 2]);
  b_k1_k0 = _amem8((void *) &b[i + (k+1)*ncb]);
  b_k3_k2 = _amem8((void *) &b[i + (k+1)*ncb + 2]);
  __x128_t b_k1_k0_b_1_0 = _llto128(b_k1_k0, b_1_0);
  __x128_t b_k3_k2_b_3_2 = _llto128(b_k3_k2, b_3_2);
  sum0 = _dsadd2(_cmatmpyr1(a_1_0, b_k1_k0_b_1_0), sum0);
  sum1 = _dsadd2(_cmatmpyr1(a_j1_j0, b_k1_k0_b_1_0), sum1);
  sum2 = _dsadd2(_cmatmpyr1(a_3_2, b_k1_k0_b_1_0), sum2);
  sum3 = _dsadd2(_cmatmpyr1(a_j3_j2, b_k1_k0_b_1_0), sum3);
  sum4 = _dsadd2(_cmatmpyr1(a_1_0, b_k3_k2_b_3_2), sum4);
  sum5 = _dsadd2(_cmatmpyr1(a_j1_j0, b_k3_k2_b_3_2), sum5);
  sum6 = _dsadd2(_cmatmpyr1(a_3_2, b_k3_k2_b_3_2), sum6);
  sum7 = _dsadd2(_cmatmpyr1(a_j3_j2, b_k3_k2_b_3_2), sum7);
} /* End of loop on Nca */
```

- Most inner loop unrolled
- Construct a 128-bit vector from two 64-bit
- 128-bit vector data type
- Four-way SIMD saturated addition
- Matrix multiply operation with rounding
Matrix Multiply C66x Resources Utilization

C compiler software pipelining feedback:
• The TI C66x C compiler optimizes this loop in four cycles.
• Perfect balance in the CPU resources utilization:
  – Two 64-bit loads per cycle
  – Two CMATMPY per cycle
    • i.e. 32 16-bit x 16-bit multiplies per cycle
  – Eight saturated additions per cycle.

Additional examples are described in the application report, Optimizing Loops on the C66x DSP.
For More Information

• For more information, refer to the **C66x DSP CPU and Instruction Set Reference Guide**.

• For a list of intrinsics, refer to the **TMS320C6000 Optimizing Compiler User Guide**.

• For questions regarding topics covered in this training, visit the C66x support forums at the **TI E2E Community** website.