C6000 Compiler Tutorial

Compiler Tutorial: Goals and Capabilities

- Teach new customers the easiest way to get good out-of-the-box performance
- Prevent simple user errors from creating bad first impressions
- Provide instructions on how to tune C code in an easy way
- Provide new tuning advice with the Feedback Solution Table to aid both the novice and experienced user
- Provide an example that walks the user through 4 easy steps to tune their code

Available on external web at:
http://www.ti.com/sc/c6000compiler

Compiler Tutorial Sections

- Getting Started
  - Provides instructions on how to get up and running
  - Tips on data types
  - Compiler switches section - very important
- Refining C code
  - Basically a collection of different C optimization steps already included in various places in the user's guides
- Feedback Solution Table
  - Helps user to tune C code somewhat interactively
- C Tuning Tutorial
  - Walks user through 4 key areas of simple C tuning

Getting Started - Tips on Data Types

- Goal - Avoid out-of-the-box mistakes from new users
- Not all architectures have the same data widths for C types (long, int, short and byte)
- Standard 16 x 16 multiply on 'C5000 require int data types
- An integer multiply on the 'C6000 requires 32 x 32 operation
- Tutorial provides detail on 'C6000 sizes for long, int, short and byte

```
    C6000 Types
long    40 bits
int     32 bits
short   16 bits
byte     8 bits
```

Getting Started - Compiler Switches

- Goal - Avoid out-of-the-box mistakes from new users
- Gives user “preferred option set” for best performance

  -o3 -pm -op2 -oi0 -k -mw -mh -mi -mt
- Warns user about options to avoid for best performance

  -g -s -ss -mu -o0/o1 -mz
- Gives user alternative control code option set

  -o3 -pm -op2 -oi0 -ms2

Understanding Feedback

- Unique in industry - a real differentiator!
- Used with Feedback Solutions to tune C code
- Provides detailed feedback on each loop
  - Dependency graph info
  - Resource requirements
  - How well the compiler did

Software Pipeline Feedback

```
Loop label: LOOP
Known Minimum Trip Count:                    16
Known Max Trip Count Factor:                  4
Loop Carried Dependency Bound(^):             8
Unpartitioned Resource Bound:                10
Partitioned Resource Bound(*):               11
Resource Partition:                 A-side   B-side
.L units                              6        4
.S units                              3        6
.D units                              8        8
.M units                             11*       9
.X cross paths                        7        7
.T address paths                      8        8
Long read paths                       4        4
Long write paths                      0        0
Logical ops (.LS)                     0        0   (.L or .S unit)
Addition ops (.LSD)                  11       12   (.L or .S or .D unit)
Bound(.L .S .LS)                      5       15
Bound(.L .S .D .LS .LSD)             10       10
Searching for software pipeline schedule at ...
     ii = 11 Schedule found with 3 iterations in parallel

Done
Speculative load threshold            : 12
Collapsed Epilog Stages               : 3
Prolog not entirely removed           : Stage contains branch
Collapsed Prolog Stages               : 1
```

Unique in Industry

Key Information for Loops

Resource Utilization Information

ii - iteration interval # cycles in loop

Compiler Tutorial Example

- Demonstrates 'C6000 C compiler optimization
- Single example that steps through 4 key areas of optimization
- It's all about passing more information to the compiler
- There are 4 key areas:
  - Pointer Aliasing Info
  - Loop Count Info
    - Minimum loop count info
    - Loop count factor - ex: count is a multiple of 2 or 4
  - Pointer Alignment Info - example: word alignment
  - Program Level Optimization

Lesson 1 - Pointer Aliasing Info

```
void lesson_c(short *xptr, short *yptr, short *zptr,
                    short *w_sum, int N) {
    int i, w_vec1, w_vec2;
    short w1,w2;

    w1 = zptr[0];
    w2 = zptr[1];
    for (i = 0; i < N; i++){
        w_vec1 = xptr[i] * w1;
        w_vec2 = yptr[i] * w2;
        w_sum[i] = (w_vec1 + w_vec2) >> 15;
        }
    }
```

Inner Loop Requires:

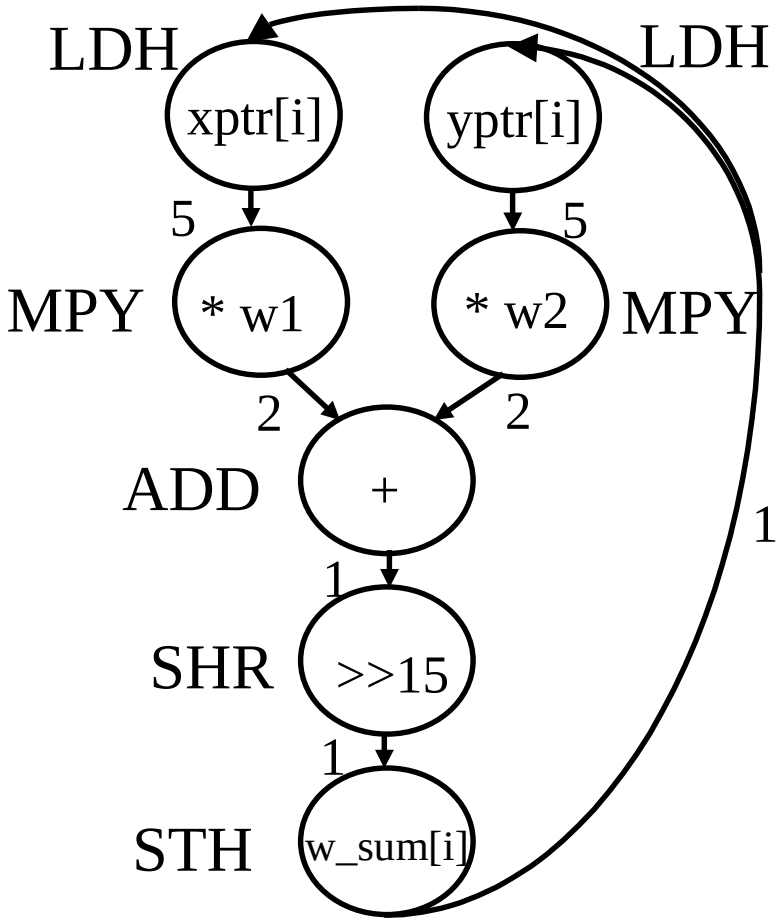
2 LDs from mem
2 MPYs
1 ADD
1 SHR
1 ST to mem

Lesson 1 - Software Pipeline Feedback

```
Known Minimum Trip Count          : 1
Known Max Trip Count Factor       : 1
Loop Carried Dependency Bound(^)  : 10
Unpartitioned Resource Bound      : 2
Partitioned Resource Bound(*)     : 2
Resource Partition:
                              A-side   B-side
.L units                        0        0
.S units                        1        1
.D units                        2*       1
.M units                        1        1
.X cross paths                  1        0
.T address paths                2*       1
Long read paths                 1        0
Long write paths                0        0
Logical  ops (.LS)              1        0    (.L or .S unit)
Addition ops (.LSD)             0        1    (.L or .S or .D unit)
Bound(.L .S .LS)                1        1
Bound(.L .S .D .LS .LSD)        2*       1

Searching for software pipeline schedule at ...
   ii = 10 Schedule found with 1 iterations in parallel
Done
```

Lesson 1 - Loop Carry Path



Assembly created by Compiler

```
        LDH   *A4++,A0 ;^ |32|
||      LDH   *B4++,B5 ;^ |32|

        NOP   2

[B0]    SUB   B0,1,B0  ; |33|

[B0]    B     L2       ; |33|

        MPY   A0,A5,A0 ;^ |32|
||      MPY   B5,B6,B  ;^ |32|

        NOP

        ADD   B5,A0,A0 ;^ |32|

        SHR   A0,15,A0 ;^ |32|
        STH   A0,*A3++ ;^ |32|
```

5+2+1+1+1 = 10 cycle loop carry path

Lesson 1 - Pointer Aliasing Info

```
void lesson1_c(short *restrict xptr, short *restrict yptr, short *zptr,
               short *w_sum, int N) {
    int i, w_vec1, w_vec2;
    short w1,w2;

    w1 = zptr[0];
    w2 = zptr[1];
    for (i = 0; i < N; i++){
        w_vec1 = xptr[i] * w1;
        w_vec2 = yptr[i] * w2;
        w_sum[i] = (w_vec1 + w_vec2) >> 15;
        }
    }
```

Adding restrict removes dependency between xptr/yptr and w_sum:

restrict says that no two pointers with a different name will alias the same memory location

Lesson 1 - Software Pipeline Feedback

```
Known Minimum Trip Count          : 1
Known Max Trip Count Factor       : 1
Loop Carried Dependency Bound(^)  : 0
Unpartitioned Resource Bound      : 2
Partitioned Resource Bound(*)     : 2
Resource Partition:
                            A-side   B-side
.L units                      0        0
.S units                      1        1
.D units                      2*       1
.M units                      1        1
.X cross paths                1        0
.T address paths              2*       1
Long read paths               1        0
Long write paths              0        0
Logical  ops (.LS)            1        0    (.L or .S unit)
Addition ops (.LSD)           0        1    (.L or .S or .D unit)
Bound(.L .S .LS)              1        1
Bound(.L .S .D .LS .LSD)      2*       1

Searching for software pipeline schedule at ...
   ii = 2 Schedule found with 3 iterations in parallel
Done
```

Loop carried dependency bound now equal to 0.

.D and .T are bottlenecks and are unbalanced between A and B side

Lesson 2 - Loop Count Info

```
void lesson2_c(const short *xptr, const short *yptr, short *zptr,
               short *w_sum, int N) {
    int i, w_vec1, w_vec2;
    short w1,w2;

#pragma MUST_ITERATE(10,40,2);

    w1 = zptr[0];
    w2 = zptr[1];
    for (i = 0; i < N; i++){
        w_vec1 = xptr[i] * w1;
        w_vec2 = yptr[i] * w2;
        w_sum[i] = (w_vec1 + w_vec2) >> 15;
        }
    }
```

Allows compiler to unroll and balance resources

MUST_ITERATE is a way to pass more info to compiler

Compiler must know loop count is a multiple of 2 to unroll

Compiler must know loop count is large enough to unroll and still be efficient

Lesson 2 - Software Pipeline Feedback

```
Loop Unroll Multiple                : 2x
Known Minimum Trip Count            : 11
Known Max Trip Count Factor         : 1
Loop Carried Dependency Bound(^)    : 0
Unpartitioned Resource Bound        : 3
Partitioned Resource Bound(*)       : 3
Resource Partition:
                            A-side   B-side
.L units                      0        0
.S units                      2        1
.D units                      3*       3*
.M units                      2        2
.X cross paths                1        1
.T address paths              3*       3*
Long read paths               1        1
Long write paths              0        0
Logical  ops (.LS)            1        1   (.L or .S unit)
Addition ops (.LSD)           0        1   (.L or .S or .D unit)
Bound(.L .S .LS)              2        1
Bound(.L .S .D .LS .LSD)      2        2

Searching for software pipeline schedule at ...
   ii = 3  Schedule found with 5 iterations in parallel
Done
```

.D and .T are balanced between A and B side because the loop has been unrolled.

.D and .T are the bottleneck of the loop with 6 memory accesses

Lesson 3 - Pointer Alignment Info

```
void lesson3_c(const short *xptr, const short *yptr, short *zptr,
               short *w_sum, int N) {
    int i, w_vec1, w_vec2;
    short w1,w2;

    #pragma MUST_ITERATE(10,40,2);

_nassert((int)(xptr) % 4) == 0);
_nassert((int)(yptr) % 4) == 0);

    w1 = zptr[0];
    w2 = zptr[1];
    for (i = 0; i < N; i++){
        w_vec1 = xptr[i] * w1;
        w_vec2 = yptr[i] * w2;
        w_sum[i] = (w_vec1 + w_vec2) >> 15;
        }
    }
```

Allows compiler to use LDW for two accesses

_nassert is used to tell the compiler that xptr and yptr are word aligned.

Compiler can now use LDW to load two xptr and two yptr values at a time

Continuation of Speaker Notes for Previous Slide,

Lesson 3, Pointer Alignment Info

Lesson 3 - Software Pipeline Feedback

```
Loop Unroll Multiple                : 2x
Known Minimum Trip Count            : 12
Known Max Trip Count Factor         : 2
Loop Carried Dependency Bound(^)    : 0
Unpartitioned Resource Bound        : 2
Partitioned Resource Bound(*)       : 2
Resource Partition:
                              A-side   B-side
.L units                        0        0
.S units                        2*       1
.D units                        2*       2*
.M units                        2*       2*
.X cross paths                  1        1
.T address paths                2*       2*
Long read paths                 1        1
Long write paths                0        0
Logical  ops (.LS)              1        1   (.L or .S unit)
Addition ops (.LSD)             0        1   (.L or .S or .D unit)
Bound(.L .S .LS)                2*       1
Bound(.L .S .D .LS .LSD)        2*       2*

Searching for software pipeline schedule at ...
   ii = 2  Schedule found with 6 iterations in parallel
Done
```

.D and .T are now only needing 4 memory accesses due to LDWs

Lesson 4 - Program Level Optimization

- Compiler option -pm enables program level optimization
- Previous three key areas for C tuning:
  - Pointer Aliasing info to reduce loop carry paths
  - Loop count info
    - Minimum loop count info
    - Loop count factor - ex: count is a multiple of 2 or 4
  - Pointer alignment info - ex: word alignment
- Program level optimization automates all three key areas
  - Gives compiler a full program view
  - Automatically extracts pointer, loop count, and alignment info
  - Key for large applications

Lesson 4 - Program Level Optimization

```
Loop Unroll Multiple                : 2x
Known Minimum Trip Count            : 12
Known Max Trip Count Factor         : 2
Loop Carried Dependency Bound(^)    : 0
Unpartitioned Resource Bound        : 2
Partitioned Resource Bound(*)       : 2
Resource Partition:
                              A-side   B-side
.L units                        0        0
.S units                        2*       1
.D units                        2*       2*
.M units                        2*       2*
.X cross paths                  1        1
.T address paths                2*       2*
Long read paths                 1        1
Long write paths                0        0
Logical  ops (.LS)              1        1   (.L or .S unit)
Addition ops (.LSD)             0        1   (.L or .S or .D unit)
Bound(.L .S .LS)                2*       1
Bound(.L .S .D .LS .LSD)        2*       2*

Searching for software pipeline schedule at ...
   ii = 2  Schedule found with 6 iterations in parallel
Done
```

Feedback of original unmodified C Code

- **No intrinsics**
- **No pragmas**
- **No nassert statements**
- **No restrict qualifiers**
- **Nothing C6000 specific**

Performance Summary

Tutorial Example	Initial Lesson	Lesson 1	Lesson 2	Lesson 3
Pointer Aliasing Info	×	✓	✓	✓
Loop Count Info	×	×	✓	✓
Pointer Alignment Info	×	×	×	✓
Cycles per Iteration	10	2	1.5	1
Cycles per Iteration w/ Program Level Optimization	1	1	1	1

Unrolling Example: FIR Filter C Function (FIR.C)

```
void firFilter(short *x, int f, short *y, int N, int M, QScale)

{ int i, j, sum;

    for (j = 0; j < M; j++) {

        sum = 0;

        for (i = 0; i < N; i++)

            sum += x[i + j] * filterCoeff[f][i];

        y[j] = sum >> QScale;

        y[j] &= 0xfffe;

    }

}
```

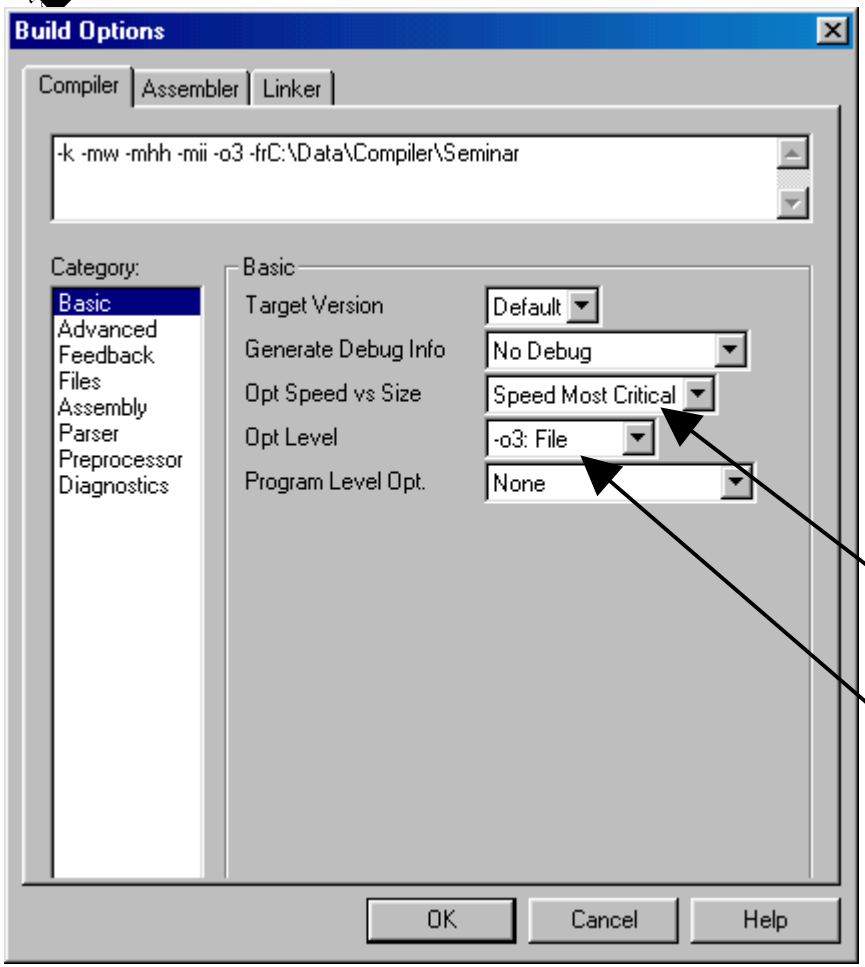
Number of Output Samples

Number of Coefficients in Filter

Multiply and Accumulate (Convolution)

Scale and Tailor Filter Output

Selecting Compiler Options



High MIPS code requires high performance options

Speed most critical

Highest optimization - level 3

Compiler Feedback

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count          : 1
;*      Known Max Trip Count Factor       : 1
;*      Loop Carried Dependency Bound(^)  : 0
;*      Unpartitioned Resource Bound      : 1
;*      Partitioned Resource Bound(*)     : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                    0        0
;*      .S units                    0        1*
;*      .D units                    1*       1*
;*      .M units                    1*       0
;*      .X cross paths              1*       0
;*      .T address paths            1*       1*
;*      Long read paths             0        0
;*      Long write paths            0        0
;*      Logical  ops (.LS)          0        0     (.
;*      Addition ops (.LSD)         1        1     (.
;*      Bound(.L .S .LS)            0        1*
;*      Bound(.L .S .D .LS .LSD)    1*       1*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 1  Schedule found with 8 iterations in parallel
;*      done
;*
;*      Collapsed epilog stages     : 7
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 2
;*
;*----------------------------------------------------------------------------*
```

- **Unique in Industry**
- **Shows Performance Achieved**

Key Information for Our Loop

`ii = 1` (iteration interval = 1 cycle)
Means: Single Cycle Inner Loop

`B side .M unit not used`
Means: Only one MPY per cycle

Full details on compiler feedback covered in Compiler Tutorial at:
www.ti.com/sc/c6000compiler

Resulting Inner Loop

- **Performs All Inner Loop Instructions in parallel**
- **Achieves one multiply accumulate every cycle**
- **C62x Maximum of 2 Loads performed every cycle**

Compiler Output

```
        LDH  *A3++,A4 ;  Load x input data
||      LDH  *B4++,B5 ;  Load coefficient
||      MPY  B5,A4,A5 ;  Multiply x and coeff
||      ADD  A5,A0,A0 ;  Accumulate result
|| [B0] SUB  B0,1,B0  ;  Decrement loop counter
|| [B0] B    L3       ;  Branch inner loop
```

Since the C62x has 2 Multipliers, can we do better?

Yes, Let's Unroll the Loop...

Imposed Unroll of Loops

Loop unrolling is usually automatically implemented, but can be forced with the UNROLL pragma, as seen below

```
void firFilter(short *x, int f, short *y, int N, int M, QScale)
{ int i, j, sum;
    #pragma UNROLL(2)
    for (j = 0; j < M; j++) {
        sum = 0;
        #pragma UNROLL(2)
        for (i = 0; i < N; i++)
            sum += x[i + j] * filterCoeff[f][i];
        y[j] = sum >> QScale;
        y[j] &= 0xfffe;
    }}
```

Unroll outer loop

Unroll inner loop

New Unrolled Inner Loop

Achieves 2 multiply accumulates every cycle

```
   [!B1] ADD    A5,A4,A4      ; running accumulator 1
||[!B1] ADD    B8,B4,B4      ; running accumulator 2
||      MPYHL  B0,A6,A5      ; h1*x1
||      MPY    A6,B0,B8      ; h0*x1
|| [A1] B      L14           ; Branch for inner loop
||      LDH    *B7++(4),B9   ; Load x0
||      LDH    *++A3(4),A5   ; Load x2

    [B1] SUB    B1,1,B1      ; dec conditional counter
||[!B1] ADD    B8,B5,B5      ; running accumulator 3
||[!B1] ADD    A6,A0,A0      ; running accumulator 4
||      MPY    B9,B0,B8      ; h0*x0
||      MPYHL  B0,A5,A6      ; h1*x2
|| [A1] SUB    A1,1,A1       ; dec loop counter
||      LDW    *B6++,B0      ; Load h0 & h1
||      LDH    *+A3(2),A6    ; Load x1
```

C6000 Benchmarks (on the TI Website)

Algorithm	***Used in***	***Assembly Cycles***	***Assembly Time (µs)***	***C Cycles*** (Rel 4.0)	***C Time (µs)***	***% Efficiency vs Hand Coded***
Block Mean Square Error *MSE of a 20 column image matrix*	For motion compensation of image data	348	1.16	402	1.34	**87%**
Codebook Search	CELP based voice coders	977	3.26	961	3.20	**100+%**
Vector Max *40 element input vector*	Search Algorithms	61	0.20	59	0.20	**100+%**
All-zero FIR Filter *40 samples, 10 coefficients*	VSELP based voice coders	238	0.79	280	0.93	**85%**
Minimum Error Search *Table Size = 2304*	Search Algorithms	1185	3.95	1318	4.39	**90%**
IIR Filter *16 coefficients*	Filter	43	0.14	38	0.13	**100+%**
IIR – cascaded biquads *10 Cascaded biquads (Direct Form II)*	Filter	70	0.23	75	0.25	**93%**
MAC *Two 40 samples vector*	VSELP based voice coders	61	0.20	58	0.19	**100+%**
Vector Sum *Two 44 sample vectors*		51	0.17	47	0.16	**100+%**
MSE *MSE between two 256 element vectors*	Mean Square Error computation in Vector Quantizer	279	0.93	274	0.91	**100+%**

TI ‘C62x Compiler Performance Rel 4.0 : Execution Time in µs @ 300 MHz