C28x Compiler Optimization

George Mock

August 2019

Agenda

- Enable Optimization
- Prefer Signed Loop Counters
- Background
- Restrict
- Dual MAC – C Code Method
- Dual MAC – Intrinsic Method

Enable Optimization

- Focus on inner loop
- How many assembly instructions?
- --opt_level=off
- --opt_level=2

```
void f_auto_correlation(AUTO_CORR *f)
{
   int i, k;
   float sum;

   for (i=0; i < f->lag; i++)
   {
      sum = 0;

      /* ----- INNER LOOP ----- */
      for (k = f->lag; k < (f->len+f->lag); k++)
         sum += f->input[k] * f->input[k-i];

      f->output[i] = sum;
   }
}
```

Inner Loop Comparison

--opt_level=off (default)

```
$C$L2:
        MOV       ACC,*-SP[6] << 1
        ADDL      ACC,*+XAR5[2]
        MOVL      XAR4,ACC
        MOV       AL,*-SP[6]
        SUB       AL,*-SP[5]
        MOV32     R0H,*+XAR4[0]
        MOV       ACC,AL << 1
        ADDL      ACC,*+XAR5[2]
        MOVL      XAR4,ACC
        MOV32     R1H,*+XAR4[0]
        MPYF32    R1H,R1H,R0H
        MOV32     R3H,*-SP[4]
        ADDF32    R0H,R1H,R3H
        NOP
        MOV32     *-SP[4],R0H
        INC       *-SP[6]
$C$L3:
        MOVL      XAR5,*-SP[2]
        MOVL      XAR4,*-SP[2]
        MOV       AL,*+XAR5[1]
        ADD       AL,*+XAR4[0]
        CMP       AL,*-SP[6]
        B         $C$L2,GT
```

--opt_level=2

```
        RPT       AR5
||      MACF32    R7H,R3H,*XAR6++,*XAR7++
        ADDF32    R3H,R3H,R2H
        ADDF32    R2H,R7H,R6H
        NOP
        ADDF32    R3H,R3H,R2H
```

Agenda

- Enable Optimization
- **Prefer Signed Loop Counters**
- Background
- Restrict
- Dual MAC – C Code Method
- Dual MAC – Intrinsic Method

Prefer Signed Loop Counters

```
int i, j;   /* LOOP COUNTERS */
float sum;
for (i=0; i < N; i++)
    for (j=0; j < M; j++)
        …
```

- Prefer signed over unsigned
  - **`int`** instead of **`unsigned int`**
- Compiler must presume unsigned counters may *wrap around* from 0xffff to 0
- Compiler assumes signed counters never wrap
  - If they do wrap, that is user error
- When **`i`** is signed, the compiler often changes **`array[i]`** into **`*XAR3++`**

Agenda

- Enable Optimization
- Prefer Signed Loop Counters
- **Background**
- Restrict
- Dual MAC – C Code Method
- Dual MAC – Intrinsic Method

Do NOT Lie to the Compiler

- Many methods for giving the compiler extra information
  - restrict
  - #pragma
  - intrinsic
- Verify the extra information is always correct
  - Under every circumstance
  - For every call
- Document constraints to those who call your functions
- Lying can cause bugs that are very hard to find
  - No diagnostics
  - Program silently does the wrong thing!

What is an Intrinsic?

- Similar to a function call
- Not implemented by calling a function
- Instead, a few instructions are emitted
  - Often, just one instruction
- An effective way to access unusual HW features of C28x
- Examples

```
long_var1 = __rol(long_var2);                  /* rotate left */
long_var1 = __sat(long_var2);                  /* saturate    */
__dmac(p1_32[i], p2_32[i], acc1, acc2, 0);     /* dual MAC    */
```

Agenda

- Enable Optimization
- Prefer Signed Loop Counters
- Background
- **Restrict**
- Dual MAC – C Code Method
- Dual MAC – Intrinsic Method

Restrict Example

```
typedef struct {
   float *p1;
   float *p2;
   float prod, G;
   int length;
} VECTOR;

void vector_dot_product_squared(VECTOR *vec)
{
   int i;
   for (i = 0; i < vec->length; i++)
   {
      vec->prod += vec->p1[i] * vec->p2[i];
      vec->G    += vec->p2[i] * vec->p2[i];
   }
}
```

Restrict Details

```
typedef struct {
   float * restrict p1;
   float * restrict p2;
   float prod, G;
   int length;
} VECTOR;
```

- Put **`restrict`** between **`*`** and the name of the pointer variable
- Property of the pointer, not the memory locations accessed through the pointer
- Data accessed through a restrict pointer is never accessed another way
  - During the scope of the pointer
  - Conservatively correct definition
  - Full definition allows corner cases that rarely matter in practice

Restrict Example

- Could **`p1`** point to **`prod`** or **`G`**?
- Could **`p2`** point to **`prod`** or **`G`**?
- Default answer is yes
- Restrict changes the answer to no
- What effect does that have?

```
for (i = 0; i < vec->length; i++)
{
   vec->prod += vec->p1[i] * vec->p2[i];
   vec->G    += vec->p2[i] * vec->p2[i];
}
```

Restrict Comparison

No restrict (default)

```
        ; repeat block starts      ; loop starts
$C$L1:
        MOV32     R1H,*XAR7++      ; read
        MOV32     R2H,*+XAR5[0]    ; read
        MPYF32    R2H,R2H,R1H
        NOP
        ADDF32    R0H,R0H,R2H
        NOP
        MOV32     *+XAR4[4],R0H    ; write
        MOV32     R1H,*XAR5++      ; read
        MPYF32    R2H,R1H,R1H
        NOP
        ADDF32    R3H,R3H,R2H
        NOP
        MOV32     *+XAR4[6],R3H    ; write
        ; repeat block ends        ; loop ends
$C$L2:
        LRETR
```

Add restrict

```
        ; repeat block starts      ; loop starts
$C$L1:
        MOV32     R2H,*XAR7++      ; read

        MPYF32    R1H,R2H,R2H
||      MOV32     R4H,*XAR5++      ; read

        MPYF32    R2H,R2H,R4H
        ADDF32    R0H,R0H,R1H
        ADDF32    R3H,R3H,R2H
        NOP
        ; repeat block ends        ; loop ends
$C$L2:
        MOV32     *+XAR4[6],R0H    ; write
        MOV32     *+XAR4[4],R3H    ; write
$C$L3:
        MOV32     R4H,*--SP
        LRETR
```

Agenda

- Enable Optimization
- Prefer Signed Loop Counters
- Background
- Restrict
- **Dual MAC – C Code Method**
- Dual MAC – Intrinsic Method

Dual MAC

- DMAC is most powerful instruction on C28x
- How can it be generated from C code?
- Two methods
  - C code with extra information
  - Intrinsic

Dual Mac – C Code Method

```
#include <stdint.h>

int32_t compute_mac(int16_t *p1,
    int16_t *p2, int_fast16_t length)
{
   int_fast16_t i;
   int32_t result = 0;

   _nassert((intptr_t)p1 % 2 == 0);
   _nassert((intptr_t)p2 % 2 == 0);

   #pragma MUST_ITERATE(,,2)
   for (i = 0; i < length; i++)
      result += (int32_t) p1[i] * p2[i];

   return result;
}
```

- Method: C code with extra information
- The next few slides repeat key lines from this example, then explain them

Include <stdint.h>

```
#include <stdint.h>
```

- Use standardized type names from <stdint.h>

Type	Means
int32_t	signed, exactly 32-bits
int16_t	signed, exactly 16-bits
int_fast16_t	signed, fastest type that is at least 16-bits
intptr_t	signed, wide enough to hold a pointer

Pointers Are 32-bit Aligned

```
_nassert((intptr_t)p1 % 2 == 0);
_nassert((intptr_t)p2 % 2 == 0);
```

- **`_nassert`** is similar to an intrinsic, but generates no code
- Means the expression is always true
- Expression says the pointer is aligned to a 32-bit boundary
  - C28x addresses are counted in 16-bit words
  - Aligned to an even word boundary

Loop Runs an Even Number of Times

```
#pragma MUST_ITERATE(,,2)
```

- #pragma is preprocessor directive
- MUST_ITERATE describes behavior of the next loop
- MUST_ITERATE(min, max, multiple)
  - min: minimum number of times the loop runs
  - max: maximum number of times the loop runs
  - multiple: loop runs a multiple of this many times
- Can leave arguments blank
- `(,,2)` means no minimum, no maximum, multiple of 2

16x16 to 32 Multiply

```
result += (int32_t) p1[i] * p2[i];
```

- Without the cast, the upper bits of the multiply are lost
- Default behavior of multiply in C works this way
  - **`(int32_t) ((int16_t) p1[i] * (int16_t) p2[i])`**
  - 16x16 multiply, keep lower 16-bits, signed extend to 32-bits
- The cast changes behavior from default to this
  - **`(int32_t) p1[i] * (int32_t) p2[i]`**
  - Each operand is signed extended to 32-bits, then multiplied
- C28x compiler recognizes this idiom, and maps to MAC-style instructions
- Application Note: How to Write Multiplies Correctly in C Code
  - http://www.ti.com/lit/pdf/spra683 (link)

Dual MAC Required Build Options

- --opt_level=2 or higher
- --unified_memory
  - Indicates data memory bus and program memory bus are connected to the same blocks of memory
  - Very rare for these buses to be connected to different blocks of memory
  - If there is any doubt, check the data sheet

Dual MAC Final Result

```
    for (i = 0; i < length; i++)
       result += (int32_t) p1[i] * p2[i];
```

Becomes …

```
        RPT       AR5
||      DMAC      ACC:P,*XAR4++,*XAR7++
        ADDL      P,ACC
```

Dual MAC How to Align Data

- Data passed to **`compute_mac`** function must be aligned to 32-bits
- How is that done?
- Global or static data, use #pragma DATA_ALIGN

```
#define N 256
#pragma DATA_ALIGN(ar1, 2)
#pragma DATA_ALIGN(ar2, 2)
int16_t ar1[N];
int16_t ar2[N];
...

     compute_mac(ar1, ar2, N);
```

- Addresses returned by RTS function **`malloc`** are 32-bit aligned

Agenda

- Enable Optimization
- Prefer Signed Loop Counters
- Background
- Restrict
- Dual MAC – C Code Method
- **Dual MAC – Intrinsic Method**

Dual MAC – Intrinsic Method

- Second method for implementing dual MAC
- Uses intrinsic **`__dmac`**
- Guarantees DMAC is emitted
- Use the method you prefer

Dual MAC – Intrinsic Method

```
#include <stdint.h>

int32_t intrinsic_mac(int16_t *p1, int16_t *p2, int_fast16_t length)
{
   int_fast16_t i;
   int32_t *p1_32 = (int32_t *) p1;
   int32_t *p2_32 = (int32_t *) p2;
   int32_t acc1, acc2;

   acc1 = acc2 = 0;
   length >>= 1;

   for (i = 0; i < length; i++)
      __dmac(p1_32[i], p2_32[i], acc1, acc2, 0);

   return acc1 + acc2;
}
```

Pointers Must Be 32-bit Aligned

```
int32_t *p1_32 = (int32_t *) p1;
int32_t *p2_32 = (int32_t *) p2;
```

- **__dmac** requires 32-bit memory operands
- These lines presumes **p1** and **p2** are aligned to 32-bits
- Because of optimization, these assignments result in no code

Two Accumulators

```
int32_t acc1, acc2;
```

- **__dmac** requires two 32-bit wide variables
- Accumulate results across multiple calls to **__dmac**
- Passed by reference
  - Concept borrowed from C++
  - Internet search on *c++ reference*
  - Means the variable is modified by **__dmac**

```
return acc1 + acc2;
```

- Final result requires adding the accumulators

Halve the Loop Count

```
length >>= 1;
```

- It is called dual MAC because it computes two multiply-accumulates at once
- Thus, run the loop half as many times
- Right-shift by 1 is the same as divide by 2
- Presumes **`length`** is even

References

- C2000 Performance Tips and Tricks

  http://processors.wiki.ti.com/index.php/C2000_Performance_Tips_and_Tricks (link)
- TMS320C28x Optimizing C/C++ Compiler User's Guide

  http://www.ti.com/lit/pdf/spru514 (link)

Questions?