C6000 Compiler Optimization

George Mock

August 2019

Agenda

- Background
- Example Function
- Build Options
- Types
- Unroll and Jam
- SIMD: Single Instruction Multiple Data
- References

Software Pipelining

- C6000 processor family improves loop performance by using software pipelining
- Without software pipelining, loop iteration i completes before iteration i+1 begins. Software pipelining overlaps iterations.
- May improve loop performance 20x!

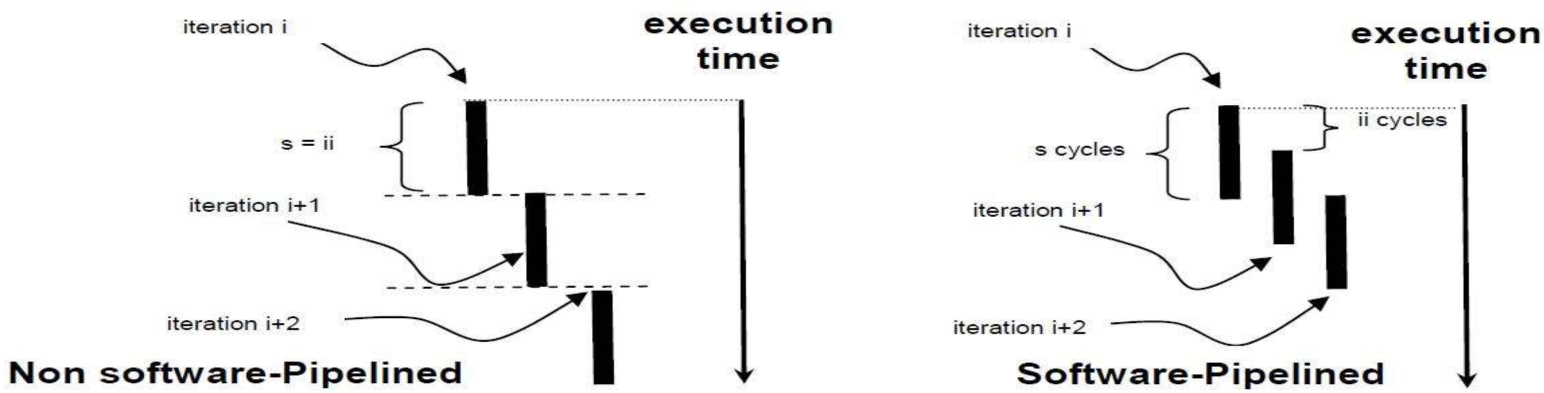

Figure 2. Software Pipelining

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!

Agenda

- Background
- **Example Function**
- Build Options
- Types
- Unroll and Jam
- SIMD: Single Instruction Multiple Data
- References

Example Function

```
void Xcorr(short *pd1,
           int len1,
           short *pd2,
           int len2)
{
    int i, j;
    long long sum;

    for(i=0; i < len1; i++)
    {
        sum = 0;
        for(j=0; j < len2; j++)
        {
            sum += pd1[i+j]*pd2[j];
        }
        pixCorrResult[i] = sum;
    }
}
```

Agenda

- Background
- Example Function
- **Build Options**
- Types
- Unroll and Jam
- SIMD: Single Instruction Multiple Data
- References

Recommended Build Options

- --silicon_version=6740
  - Controls use of CPU specific instructions
  - CPU’s supported: 6400+, 6740, 6600
- --opt_level=2
  - Level of optimization
- --src_interlist
  - Compiler generated assembly file is not deleted
  - Comments are added which make assembly easier to understand
- --debug_software_pipeline
  - Every software pipelined loop is preceded by a block comment
  - Shows information about the loop
  - Makes the block comment much more verbose

Optimization

Option	Range of Optimization
--opt_level=off	None
--opt_level=0	Statements
--opt_level=1	Blocks
--opt_level=2	Functions
--opt_level=3	Files

- Only a rough summary
- Some level 0 and 1 optimizations range farther

Optimization Level – C6000

- Default: --opt_level=off
- Must use at least --opt_level=2 to get software pipelining

Agenda

- Background
- Example Function
- Build Options
- **Types**
- Unroll and Jam
- SIMD: Single Instruction Multiple Data
- References

Change Types

FROM: Built-in types

```
void Xcorr(short *pd1,
           int len1,
           short *pd2,
           int len2)
{
    int i, j;
    long long sum;
```

TO: <stdint.h> types

```
#include <stdint.h>
void Xcorr(int16_t *pd1,
           int_fast32_t len1,
           int16_t *pd2,
           int_fast32_t len2)
{
    int_fast32_t i, j;
    int40_t sum;        /* smaller! */
```

Include <stdint.h>

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

- Use standardized type names from <stdint.h>

Type	Means
int16_t	signed, exactly 16-bits
int_fast32_t	signed, fastest type that is at least 32-bits
int40_t	signed, exactly 40-bits
intptr_t	signed, wide enough to hold a pointer

Change Type of sum

- The type of sum changes from long long to int40_t
  - Changes size from 64-bits to 40-bits
- Accumulates multiply-accumulate of inner loop
- Presumes sum never exceeds 40-bits
- Compiler cannot automatically change sum to a smaller type
- Immediate effect is small
- When combined with later changes, the effect is larger
- Overall point: Don’t compute more bits than you need

Agenda

- Background
- Example Function
- Build Options
- Types
- **Unroll and Jam**
- SIMD: Single Instruction Multiple Data
- References

Unroll and Jam: Source Changes

FROM

```
void Xcorr(int16_t *pd1,
           int_fast32_t len1,
           int16_t *pd2,
           int_fast32_t len2)
{
    int_fast32_t i, j;
    int40_t sum;


    for(i=0; i < len1; i++)
    {
        sum = 0;

        for(j=0; j < len2; j++)
        {
            sum += pd1[i+j]*pd2[j];
```

TO

```
void Xcorr(int16_t * restrict pd1,
           int_fast32_t len1,
           int16_t * restrict pd2,
           int_fast32_t len2)
{
    int_fast32_t i, j;
    int40_t sum;

    #pragma MUST_ITERATE(2,,2)
    for(i=0; i < len1; i++)
    {
        sum = 0;
        #pragma MUST_ITERATE(1)
        for(j=0; j < len2; j++)
        {
            sum += pd1[i+j]*pd2[j];
```

Restrict Details

```
int16_t * restrict pd1
```

- 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
- The effect of restrict is shown later

#pragma MUST_ITERATE

```
#pragma MUST_ITERATE(2,,2)
/* outer loop */
    #pragma MUST_ITERATE(1)
    /* inner loop */
```

- Iterate means the number of times the loop executes
- #pragma is preprocessor directive
- MUST_ITERATE describes behavior of the next loop
- #pragma MUST_ITERATE(min, max, multiple)
  - min: minimum number of times the loop iterates
  - max: maximum number of times the loop iterates
  - multiple: loop iterates a multiple of this many times
- Can omit arguments

Effect of MUST_ITERATE

```
#pragma MUST_ITERATE(2,,2)
/* outer loop */
    #pragma MUST_ITERATE(1)
    /* inner loop */
```

- For unroll and jam, user must tell compiler the following
  - Outer loop iterates at least two times, and a multiple of 2 times
  - Inner loop iterates at least one time
- **`(2,,2)`** means at least 2 times, no maximum, multiple of 2
- **`(1)`** means at least 1 time, no maximum, no multiple

Unroll and Jam

- These source changes enable an optimization named unroll and jam
- Compiler performs this optimization automatically
- The following code examples demonstrate unroll and jam
- Do NOT make these changes in your code

Unroll and Jam: Before

```
for(i=0; i < len1; i++)
{
    sum = 0;
    for(j=0; j < len2; j++)
    {
        sum += pd1[i+j]*pd2[j];
    }
    pixCorrResult[i] = sum;
}
```

Unroll and Jam: Unroll Outer Loop One Time

```
for(i=0; i < len1; i += 2)
{
    sum = 0;
    for(j=0; j < len2; j++)
    {
        sum += pd1[i+j]*pd2[j];
    }
    pixCorrResult[i] = sum;            /* AAA */

    sum = 0;
    for(j=0; j < len2; j++)
    {
        sum += pd1[i+1+j]*pd2[j];    /* BBB */
    }
    pixCorrResult[i+1] = sum;
}
```

Unroll and Jam: Jam Inner Loops Together

```
for(i=0; i < len1; i += 2)
{
    sum0 = sum1 = 0;
    for(j=0; j < len2; j++)
    {
        sum0 += pd1[i+j]*pd2[j];
        sum1 += pd1[i+1+j]*pd2[j];   /* BBB */
    }
    pixCorrResult[i] = sum0;          /* AAA */
    pixCorrResult[i+1] = sum1;
}
```

Effect of Restrict on Memory References

Before

```
pixCorrResult[i] = sum;         /* AAA */
…
sum += pd1[i+j]*pd2[j];         /* BBB */
```

After

```
sum1 += pd1[i+1+j]*pd2[j];   /* BBB */
…
pixCorrResult[i] = sum0;     /* AAA */
```

- AAA and BBB mark memory reference to focus on
- Note how they change order
- Restrict enables this change
- Without restrict, no unroll and jam

Agenda

- Background
- Example Function
- Build Options
- Types
- Unroll and Jam
- **SIMD: <u>S</u>ingle <u>I</u>nstruction <u>M</u>ultiple <u>D</u>ata**
- References

SIMD: Source Changes

FROM

```
#pragma MUST_ITERATE(2,,2)
for(i=0; i < len1; i++)
{
    sum = 0;
    #pragma MUST_ITERATE(1)
    for(j=0; j < len2; j++)
    {
        sum += pd1[i+j]*pd2[j];
    }
```

TO

```
_nassert(((intptr_t)pd1 % 8) == 0);
_nassert(((intptr_t)pd2 % 8) == 0);

#pragma MUST_ITERATE(8,,8)
for(i=0; i < len1; i++)
{
    sum = 0;
    #pragma MUST_ITERATE(2,,2)
    for(j=0; j < len2; j++)
    {
        sum += pd1[i+j]*pd2[j];
    }
```

SIMD Overview

- Single Instruction Multiple Data
- One instruction performs multiple operations
- Examples: LDDW, LDNDW, DOTP2
- Arrange code to put 8 bytes worth of similar operations close together
- Align pointers to 8 byte boundaries
- Insure loops iterate an even multiple of times
  - char operations? 8 times
  - short operations? 4 times
  - int operations? 2 times

Pointers Are 8-Byte Aligned

```
_nassert((intptr_t)pd1 % 8 == 0);
_nassert((intptr_t)pd2 % 8 == 0);
```

- **`_nassert`** is similar to an function, but generates no code
- Means the expression is always true
- Expression says the pointer is aligned to an 8-byte boundary

SIMD: Inner Loop

```
#pragma MUST_ITERATE(2,,2)
for(j=0; j < len2; j++)
{
    sum += pd1[i+j]*pd2[j];
}
```

- (2,,2) means inner loop iterates at least 2 times, and a multiple of 2 times
- Recall outer loop is unrolled 2 times
- Thus, the inner loop is guaranteed to run 2*2=4 times overall
- Each memory read is 2 bytes
- Now 4*2=8 bytes of similar operations are close together

SIMD: Outer Loop

```
#pragma MUST_ITERATE(8,,8)
for(i=0; i < len1; i++)
```

- (8,,8) means loop iterates at least 8 times, and a multiple of 8 times
- For unroll and jam, the only requirement is a multiple of 2 times
- At values less than 8, compiler generates multiple copies of the loop
  - Chooses between these loops dynamically at run time
  - Each loop optimized for a range of iteration counts
  - Since this loop will always run some multiple of 8 times, that wastes code space
- Feel free to experiment with other iteration counts

Agenda

- Background
- Example Function
- Build Options
- Types
- Unroll and Jam
- SIMD: Single Instruction Multiple Data
- **References**

References

- Article: C6000 CGT Optimization Lab
  - These slides are based on this article
  - http://processors.wiki.ti.com/index.php/C6000_CGT_Optimization_Lab_-_1 (link)
- Article: Optimization Techniques for the TI C6000 Compiler
  - Collection of links to articles, workshops, programmer's guides, etc.
  - http://processors.wiki.ti.com/index.php/Optimization_Techniques_for_the_TI_C6000_Compiler (link)
- Manual: TMS320C6000 Optimizing Compiler User's Guide
  - http://www.ti.com/lit/pdf/sprui04 (link)

Questions?