

DSP Hardware Lab

Session 5

09/29/2009

FIR Solution Discussion
Optimizing C Code
Lab 5 Walkthrough

Agenda

- Lab3 (FIR-1) Solution
- Optimizing C Code
- Lab 5 Walkthrough
- Lab Session in K223

- **READING :**
 - Chassaing (no new reading assigned)
 - SPRA666.PDF : Hand Tuning Loops and Control Code on the TMS320C6000
 - SPRAA14.PDF : Introduction to Compiler Consultant

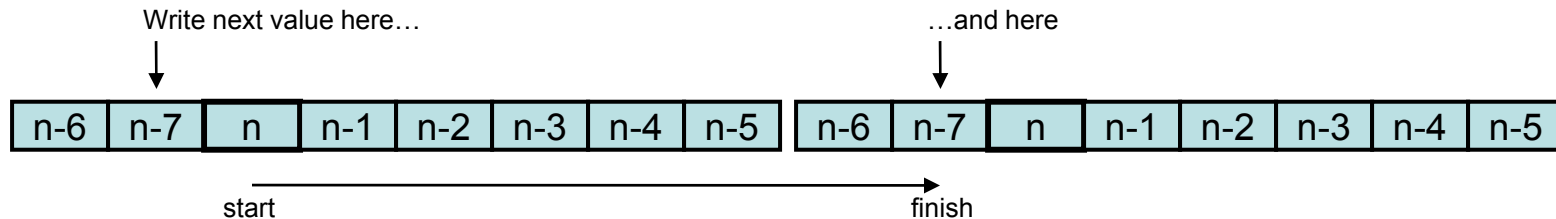
Review : Computing Convolution

- Hold Last N samples $x[n]$, $x[n-1]$...
- For each new input sample $x[n]$, compute a $y[n]$
 - $h[0] * x[n] + h[1]*x[n-1] + \dots + h[N-1]*x[n-N+1]$
- Shifting Data
- Circular Buffer

- Lets proceed to use Convolution Sum as a simple example for the wide variety of performance that can be seen simply by different coding methods

Test Case : “Double Circular Buffer”

- Using “double circular buffer” technique will allow us to simplify our convolution to a simple dot product calculation on linear vectors.
 - Good example for class



- How can we do this as fast as possible?
 - Faster = More Taps = Higher Performance
 - Faster = More CPU cycles available for other things
 - Faster = Lower Power
 - Allows same functionality with lower CPU clock rates
- Option 1 : Pure C
- Option 2 : C with Hand Written Assembly Subroutine
- Option 3 : DSPLib (Really just Option 2 where someone else wrote it)
- Option 4 : Optimized C
- Lets do some benchmarks!

Benchmarks

```
float convolve( float * hptr, Int16 * data, int
               newdataindex, int numtaps)
{
  int i; float dout = 0.0;
  Int16 *dptr = data+newdataindex;

  for (i=0;i<numtaps;i++)
    dout+= *hptr-- * *dptr--;
  return dout;
}
```

1024 taps = 288 us

Gut instinct – is this reasonable?

Convolution Calculation

- Processor runs at 300 MHz
- This is 300 million instructions per second
- That is > 6000 instructions per 1/48kHz
 - Doesn't seem right that we can only do 64 taps!
- What could be some issues?
- We know we could do better with assembly right? (and DSPLIB)

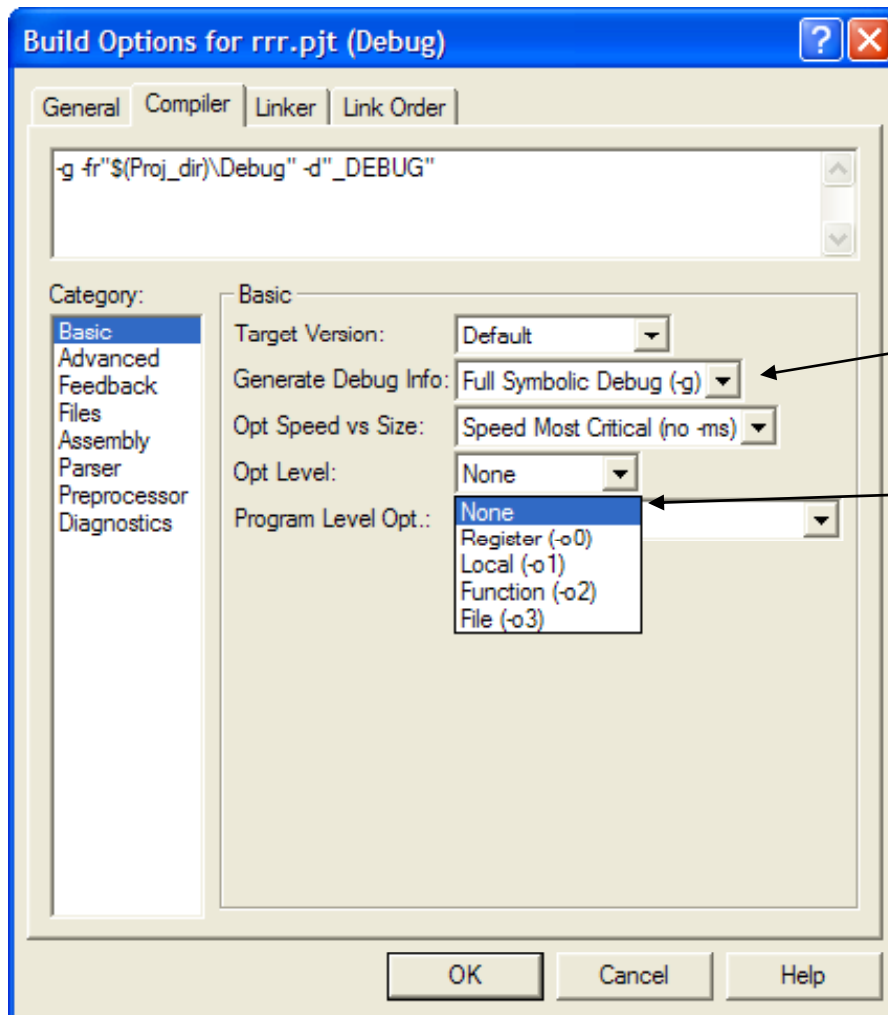
Benchmark from DSPLib programmer's ref

Cycles $(4 * \text{floor}((nh-1)/2) + 14) * (\text{ceil}(nr/4)) + 8$

e.g., $nh=10$, $nr=100$, cycles=758 cycles

So 1024 taps, 100 samples = $526 * 4 * 25 + 8 = 5268$

Setting Compiler Optimization



Include support for debugging
(obviously nice 'till production)

Compartments across which
To optimize

-O2 : Time for 1024 taps = 22 us!

Optimization

- Optimization should almost always be used in production code
- Correctly written code should still work with the optimizer on
 - Incorrectly written code may behave differently in the optimized code than in unoptimized, but this is not a good reason for its non-use
 - Optimizer can make debug much tougher.
 - Statements executed out of order
 - Variables don't necessarily exist
 - Code that you think is running gets “optimized away” if you don't use its result.
 - Nice balance is to debug functions with optimizer off, optimize them, and leave them alone.

Tools for Further Optimization

We chose an extreme example, and one that is simple to understand. There will be many cases where C code does not optimize as smoothly, and raw assembly seems far better.

- **One** reason that Assembly code performs better is that often the person writing assembly is able to make assumptions that the compiler is not allowed:
 - Locations of data
 - Alignment of parameters
 - Number of iterations through loops...etc
- Another is that often the person writing the assembly becomes aware of the bottlenecks, and can change the requirements / assumptions in order to remove those bottlenecks

Tools for Further Optimization

C Compiler tries to have free flow of information between programmer and compiler in order to more duplicate these benefits of assembly language programming.

- Information **FROM Programmer TO Compiler**
 - “Pragma” and other keywords allow programmer to tell the compiler some ways in which the code will be used.
 - In other words, the user can place restrictions on the function which may increase its speed.
- Information **FROM Compiler TO Programmer**
 - Compiler tries to communicate the bottlenecks and difficulties in its compilation by:
 - Describing the various time spent in loops
 - Showing the assembly code that it generates
 - Providing comments with that assembly to enable programmer to understand what it is doing.
 - “Compiler Consultant”

Some Tips

After this slide, much will be processor specific – however, these tips are pretty general for processors with long pipelines.

- Effectively using the capabilities of the DSP absolutely requires making as much use of parallelism as possible. (note : this is hard with the capability to execute 8 instructions in parallel)
 - Branches can be expensive
 - Don't process large data arrays in-place if possible
 - Compiler can more easily optimize if it knows that its instructions won't affect the input array. Otherwise parallelism becomes impossible
- Interactively engage the compiler by inspecting its assembly code.
- Always keep an idea of how fast something SHOULD go in your mind. When things deviate greatly from this, investigate!

Source : SPRA666.PDF

- Guide to many of these things in far more detail than presented here
- Please Read!
- Lab session can be arranged to talk about some of these things with the instructors, since lecture will not be comprehensive
- Try some examples!

Basic Guidelines

For simplicity, it is highly recommended to follow these guidelines before modifying source code:

- Choose performance oriented options (Section 3.1).
- Make sure index variables are signed integers. This communicates to the compiler that increments are linear (do not wrap around). This includes index variables used to do counting in loops. It also includes index variables used for subscripting. When the compiler knows that index variables are linear, there are more opportunities for optimization.
- Provide as much information as possible to the compiler. Use restrict qualifiers, MUST_ITERATE pragmas, and _nasserts() whenever possible (Sections 4.1 and 5.1).
- Align data on cache line boundaries and/or double-word boundaries when possible using the DATA_ALIGN and STRUCT_ALIGN pragmas. See reference [1] for more detail.
- Follow the advice in the Compiler Consultant (--consultant) and .nfo files (-on2 -o3).

Compiler Consultant

- Operates on loops
- Goal : make simple linear code more pipelined.
- Premise was that compiler can only do this sort of optimization if it has the right information.
 - “Effective software pipelining requires a great deal of loop analysis information. In many cases, not all of the information required is available in the source code. Compiler Consultant analyzes the loops in the application in order to determine what information may be missing, and then issues advice on how to address the information gap.”

Pseudocode Example

```
loop:
    load
    operate
    store
    bdec loop, count          ; decrement count, branch if != 0
```

Cycles = 4*count

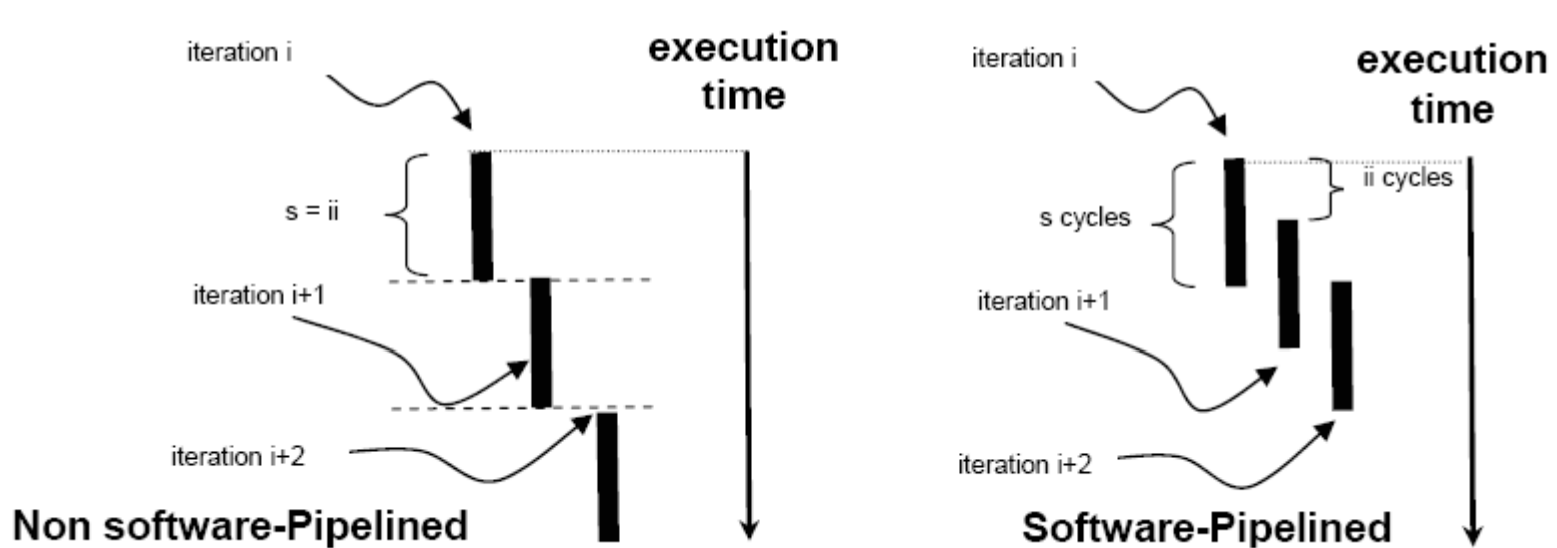
```
load(n)
load(n+1) || operate(n)
load(n+2) || operate(n+1) || store(n)
loop:
load(n+3) || operate(n+2) || store(n+1) || bdec loop, count(n)
operate(n+3) || store(n+2)
store(n+3)
```

Cycles = count + 2

Example

```
float test (float *in1, float *in2, int n) {  
    for (i=0;i<n;i++)  
        {  
            sum += *in1 * *in2 ;  
            *in2++ = *in1++;  
        }  
    return sum;  
}
```

Software Pipelining



Focus of all this optimization will generally be loop constructs

ii = "initiation interval"

S = # of cycles to complete an iteration

Consultant Advice

Function: test

Lines: 468-472

Analysis:

- Dependence constraints

increased minimum cycles/iteration

from 5 to 7.

Advice:

- [Alias](#)
- [Options](#)

Problem:

The compiler cannot determine whether

[in1\(C:\CCStudio_v3.3\examples\dsk6713\cs1\mcbsp\mcbsp1\main_mcbsp1.c:464\)](#)

might access the same memory locations as

[in2\(C:\CCStudio_v3.3\examples\dsk6713\cs1\mcbsp\mcbsp1\main_mcbsp1.c:464\)](#)

within the loop.

Suggestion:

Restrict qualify the definition of **in1**, if the safety criteria

1st Alternate Suggestion:

Restrict qualify the definition of **in2**, if the safety criteria

Also gave advice about not using `-g` for debug info generation. That didn't make a difference

Restrict keyword

In reality, experienced programmers generally write code so that input parameter arrays and output parameter arrays are independent. The reason is that this makes their algorithm much more parallel, which in turn leads to better performance. Suppose that, across all call sites, neither “input1” nor “input2” ever accesses the same memory locations as “output”. Tell this to the compiler and the back edges from the store to the loads will be eliminated. This is done either by using the `-mt` option (Section 3.1) or by using the `restrict` keyword.

```
void BasicLoop(int *restrict output,
               int *restrict input1,
               int *restrict input2,
               int n)
{
    int i;
    #pragma MUST_ITERATE(1)
    for (i=0; i<n; i++)
        output[i] = input1[i] + input2[i];
}
```

While it is sufficient for this example to restrict qualify either both loads or the single store, it is recommended to restrict qualify all parameters that can be restrict qualified (and local pointer variables as well). First, this is usually quicker than determining which actually need to be restrict-qualified. Second, this provides information to other programmers who might maintain or modify this code base in the future. However, before inserting restrict keywords, be sure that pointers being restrict-qualified cannot overlap with any other pointers. When writing a library routine and using restrict, be sure to document parameter restrictions for library users.

After adding the restrict keyword, rebuild the function. Observe that the loop-carried dependency bound has disappeared. This means that each iteration is independent. New iterations are now initiated as soon as resources are available.

First tuning iteration

```
int test (int * restrict in1, int *restrict in2, int n)
```

```
    ;*   Searching for software pipeline schedule at ...  
;*   ii = 3  Schedule found with 4 iterations in parallel  
        ;*   Done
```

more advice : use “must iterate” to specify bounds of n.

MUST_ITERATE

```
#pragma MUST_ITERATE(lower_bound, upper_bound, factor)
```

The lower bound is the lowest possible value for “n”. The upper bound is maximum possible value for “n”. The factor is a number that always divides n. Any of these parameters can be omitted.⁷

We can tell the compiler how many times that a loop will be executed. We know this by knowing how the code will be used. The compiler can use this for things such as :

- unrolling loops
- avoiding a check for a zero-trip loop

If compiler knows that loop will always be executed a factor of 2 times, it can Perhaps make some optimizations

Adding pragma before loop

```
#pragma MUST_ITERATE(20, , 2);
```

```
;* Loop Unroll Multiple : 2x
```

New number became 2.5, which is an ii of 5, but now doing 2 per iteration.

`_nassert`

The C64x and C64x+ processors support both aligned and non-aligned double-word loads and stores.⁸ If it is known that the function parameters are double-word aligned, switching to aligned, double-word memory accesses saves both D units and T address paths.

How does one get the compiler to select the double-word versions of the memory access instructions? There are two options: (1) use intrinsics, or (2) tell the compiler that the memory accesses are aligned. The second method is simpler, so it is best to try that first.

To tell the compiler that the memory accesses are aligned on double-word (64-bit) boundaries, use `_nasserts()` inside the function *just prior* to the loop of interest:⁹

```
_nassert((int) input1 % 8 == 0); // input1 is 64-bit aligned
_nassert((int) input2 % 8 == 0); // input2 is 64-bit aligned
_nassert((int) output % 8 == 0); // output is 64-bit aligned
```

Of course this should be true if we assert it, so we should make sure that data is in fact aligned by using `DATA_ALIGN` pragma in order to make sure that our variables are aligned on the right boundary.

Info FROM Compiler

Options that reduce tuning time by providing additional analysis information, while having no effect on performance or code size:

- **-s [-k|-al] -o[2|3]**. Output a copy of what the source code looks like after high-level optimization (Figure 1). This output, known as optimizer comments, looks much like the original C/C++, except that all inlining, transformations and other optimizations from this phase have been applied. Optimizer comments are interlisted with assembly code in the assembly file (with -k) and/or listing file (with -al). *This option is incredibly helpful in understanding the compiler-generated assembly.* See Section 3.3 for more detail.
- **-mw or -mw -al**. Output extra information about software-pipelined loops, including the *single scheduled iteration* (Section 4) of the loop. This information is used in the loop tuning examples presented later in this document.
- **-on2 -o3**. Create a .nfo file with the same base name as the .obj file. This file contains summary information regarding the high-level optimizations that have been applied, as well as advice.
- **--consultant**. Generates information to be used with the CCStudio Compiler Consultant. Provides tuning advice on a loop-by-loop level. Must be used in conjunction with CCStudio version 3.0 or higher.

The Compiler Consultant provides beginning to intermediate-level tuning advice. Following the advice yields performance improvement in most cases, but improvement is not guaranteed. Sometimes it is necessary to iterate or apply multiple pieces of advice before seeing any improvement. See reference [3] for more detail.