Texas Instruments

TI C7000 C/C++ Optimization Guide

v4.1

Copyright © 2024, Texas Instruments Incorporated

Online HTML version available here

CONTENTS

1 C7000 Software Development Flow 2

2 C7000 DSP CPU Architecture 6

3 Profiling Code 9

4 Basic Code Optimization Strategies and Techniques 15

5 Understanding Compiler Optimization 31

6 Advanced Code Optimization Techniques 60

7 Support 66

8 Revision History 67

9 IMPORTANT NOTICE AND DISCLAIMER 68

10 About This Document 69

11 Related Documents 70

12 Related Collateral 71

13 Legal Information 72

This guide describes the C7000™ DSP architecture and optimization techniques that are used to craft high-performance code that runs on a C7000 DSP core. It also describes tools and resources you may find useful in developing source code to run on C7000 DSPs.

Readers of this document should have the following:

- Knowledge of C and C++.
- Experience invoking the C7000 compiler using compiler options.
- Knowledge of basic assembly language concepts.
- Knowledge of CPU architectural features, such as registers, caches, and functional units.

This guide is not intended to help optimize code for the memory/cache hierarchy, MSMC, DMA, or Matrix Multiply Accelerator (MMA).

Contents:

CHAPTER

ONE

C7000 SOFTWARE DEVELOPMENT FLOW

This chapter outlines a high-level software development process that many developers have found useful in developing high-performance code for the C7000 DSP core.

Figure 1.1 shows a possible software development strategy for the C7000 DSP core. There are three phases to this strategy; you may need to iterate on phase 1 and between phases 2 and 3. After adding an optimization, measure the resulting performance/code-size and repeat.

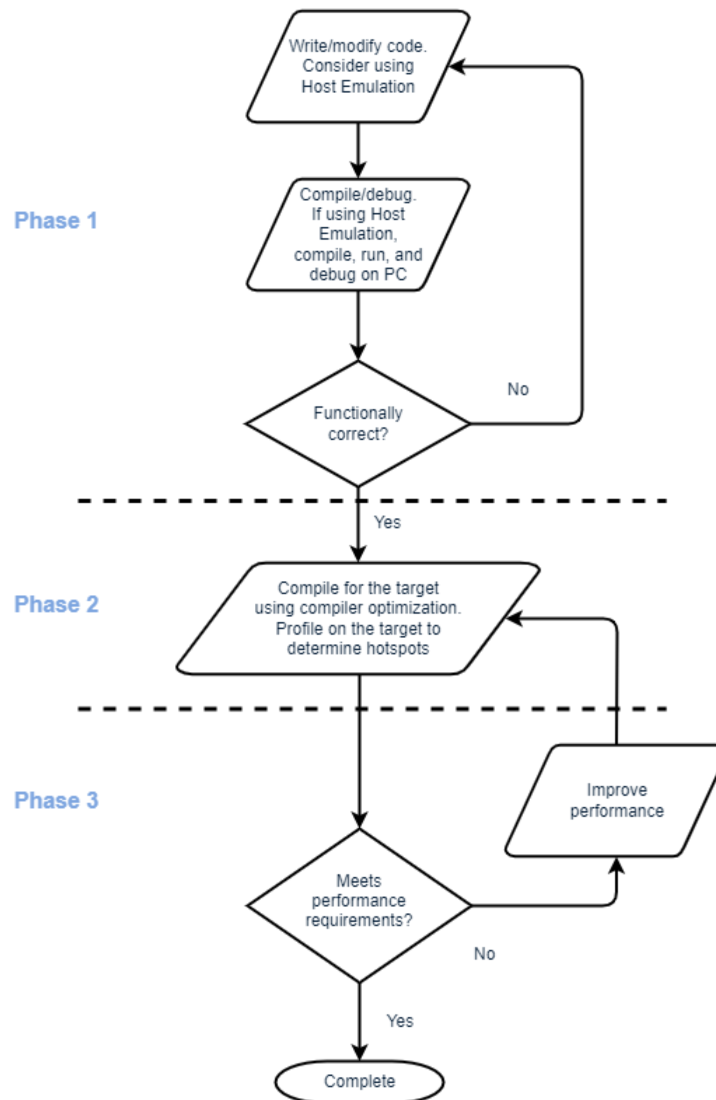

Figure 1.1: Software Development - Profiling and Optimization

It is a good idea to set up a self-checking application, so that its correctness can be checked during optimizations.

1.1 Phase 1: Create Functionally Correct Code

During the first phase of code development, concentrate on developing code that is functionally correct. A second aim of this phase is to use C7000 DSP programming model constructs and idioms in performance-critical areas that may help performance. These strategies and techniques are covered in *Basic Code Optimization Strategies and Techniques*. The developer should write, test, and revise code, iterating as necessary to produce functionally correct code before moving to the next phases of development.

Depending on the nature of the application and your testing infrastructure, it may or may not be necessary to test the code on the C7000 DSP. If testing the code on a personal computer is possible, the C7000 Compiler Tools provide a "Host Emulation" infrastructure, which allows you to use C7000 compiler intrinsics and native vector types while developing and debugging on a PC. This allows you to use different debugging tools and programming environments to prototype programs targeted for C7000 hardware before using the C7000 compiler. The Host Emulation package does not attempt to simulate the C7000 CPU. See the *C7000 Host Emulation Users Guide* (SPRUIG6) for more details.

Once the application is functionally correct, move to Phase 2.

1.2 Phase 2: Profile Code on the Target

During this phase, focus on invoking the C7000 compiler using the appropriate compiler optimization options. Run and profile the code on the target to determine:

- Does the code meet performance expectations?
- Which parts of the code consume the most time?

Perform this phase with the code running on the C7000 DSP core, typically with an emulator attached to the EVM or device on which the system-on-chip and C7000 DSP core resides.

Running and profiling the code can give you a good idea of where cycles are being spent and thus which portions of the code may need further optimization work. For example, you may find that the application spends most of its time in one or two ISRs. In such scenarios, you would focus optimization efforts on those ISRs.

Profiling Code provides an overview of profiling tools.

1.3 Phase 3: Improve Performance

In Phase 2, you use profiling information to identify sections of code that need improvement. If the performance and memory use meet requirements and testing reveals no issues, the development process can end. If performance and memory use do not meet requirements, concentrate your optimization efforts on sections of code that consume the most cycles.

Often, profiling in Phase 2 reveals memory system bottlenecks that must be addressed first. For example, some algorithms require a large amount of data in a short amount of time. If the data is placed in DDR, the algorithm could suffer from lots of cold cache misses and may run slowly. This kind of data often should be placed into fast, on-chip memory before that data is consumed. Such memory needs to be set up appropriately. The DMA features of your device can help with moving data to appropriate locations. Detailed analysis of memory bottlenecks, setting up caches and memories, and using DMA is outside the scope of this document. Please consult your Texas

Instruments Field Applications Engineer or the Texas Instruments E2E forums for guidance and more information.

Once the memories and DMA transfers are set up appropriately, further profiling can reveal hot spots that need further attention at the algorithm level. Typical steps may include:

- Enabling the appropriate compiler options, which typically include:
  - Options to take advantage of optimization passes within the compiler, such as inlining and loop optimization. Note that loop optimization is especially important when optimizing performance on C7000 devices. Options to consider are covered in *Selecting Compiler Options for Performance*.
  - Options to take advantage of hardware features.
- Where possible, using optimized libraries from TI.
- Providing more information to the compiler to help its optimizations (for example: pragmas and the restrict keyword).

Basic Code Optimization Strategies and Techniques and *Advanced Code Optimization Techniques* describe these and more strategies to optimize your code.

If the application does not yet meet performance requirements, make improvements and return to Phase 2 to collect new profiling data.

CHAPTER

TWO

C7000 DSP CPU ARCHITECTURE

Before describing compiler options and source code strategies you can use to make code more efficient, it is necessary to review some information about the C7000 Digital Signal Processor and instruction set. This chapter provides an overview of the C7000 architecture, datapath, and functional units.

Contents:

2.1 C7000 Digital Signal Processor CPU Architecture

The C7000 CPU DSP architecture is the latest high-performance digital signal processor (DSP) from Texas Instruments. It is featured in some Texas Instruments Keystone 3 devices. This Very-Long-Instruction-Word (VLIW) DSP has significant mathematical processing capabilities, due to its wide vector instructions and multiple functional units. This optimization guide can help developers get the most performance of the C7000 DSPs.

When integrated into a larger TI device, such as some Keystone 3 devices, the C7000 is often paired with a Matrix Multiply Accelerator (MMA), which can significantly improve the performance of certain machine learning networks. We recommend use of the TI Deep Learning library, which has been optimized to use the Matrix Multiply Accelerator. The TI Deep Learning library is part of the Processor SDK.

The C7000 DSP has vector (SIMD) instructions that are capable of performing up to 64 operations in a single instruction, depending on the data type and version of the C7000 CPU. Nearly all computational instructions on C7000 DSP cores are fully pipelined, which means independent instructions can be started on every clock cycle. This combination of vector instructions and pipelined behavior allows you to perform a large number of computations per cycle. The C7000 DSP cores feature both fixed-point and floating-point vector instructions.

Each C7000 DSP core has several functional units. On each clock cycle, each functional unit can be executing an independent instruction. Because the C7000 DSP cores have 13 fully-pipelined functional units, up to 13 instructions can start executtion on every clock cycle. In reality, some of the functional units are specialized for certain kinds of instructions, so for this and other reasons, it is common that not all 13 functional units execute an instruction every cycle.

For more information on the C7000 instruction set, please see the *C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator Technical Reference Manual* (SPRUIP0), which can be obtained through your TI Field Application Engineer.

2.2 C7000 Split Datapath and Functional Units

Figure 2.1 shows the datapath split on the C7100 DSP CPU. There is an A-side datapath and a B-side datapath. The diagram shows the functional units and multiple, heterogeneous register files. The A-side datapath is responsible for scalar computation, loading and storing scalars and vectors to and from memory, and control-flow (branches, calls). The B-side datapath handles vector math operations, permutations of data, and vector predication operations.

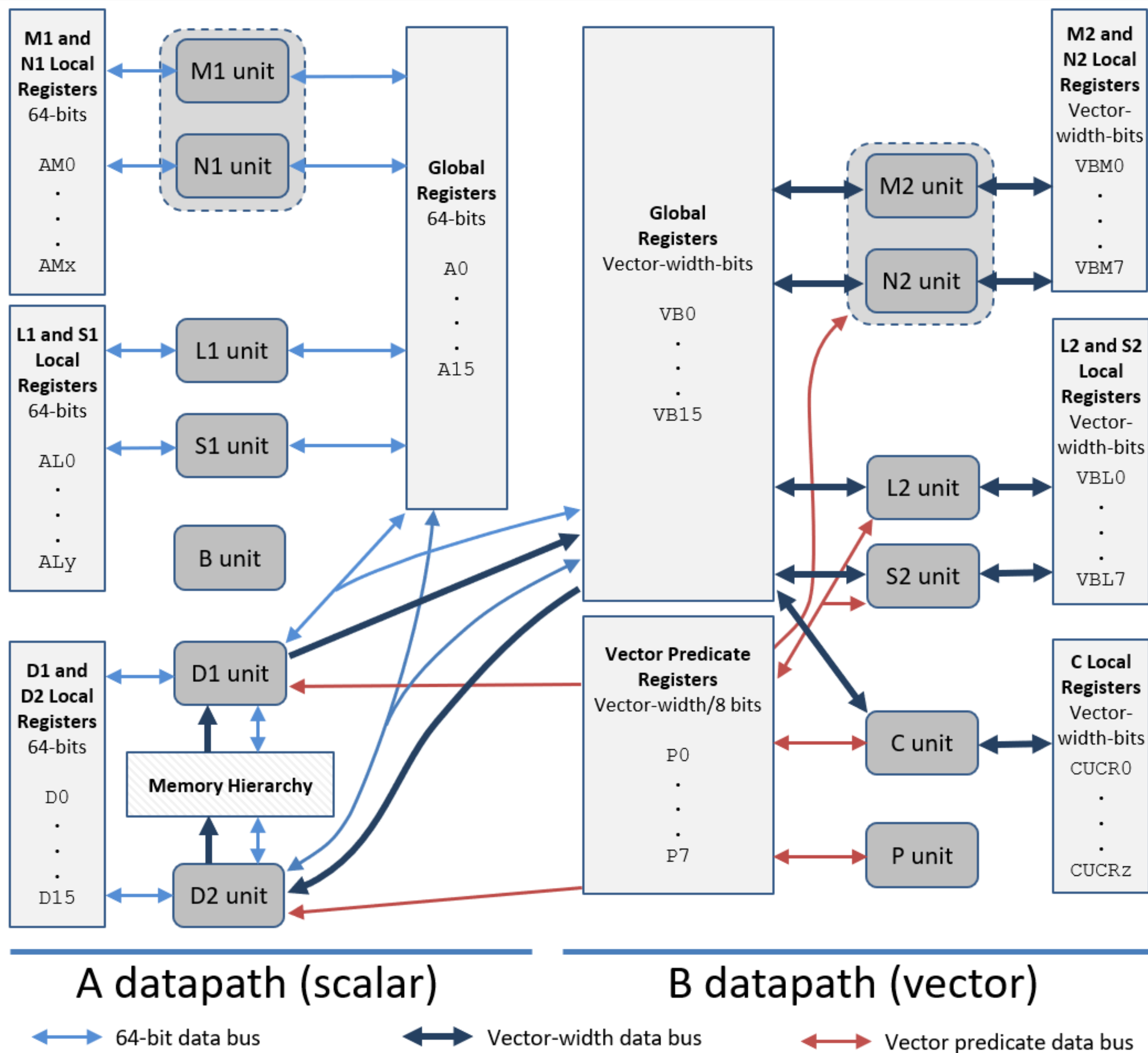

Figure 2.1: C7000 Datapath Block Diagram

To simplify Figure 2.1, some data movement capabilities and data paths are not shown in this figure.

- In general, a functional unit can write to any register file on the same datapath.
- Most functional units can obtain data from one or both of the streaming engines.
- There is one 64-bit cross path per datapath (A/B). Each cross path allows one read per cycle from the opposite side global register file.

C7100 and C7120 cores have a 512-bit vector width. C7504 and C7524 cores have a 256-bit vector width. Registers have 64 bits per register ("scalar") or a "vector-width" number of bits per register. Thus, C7100 and C7120 cores have 512-bit vector registers, while C7504 and C7524 cores have 256-bit vector registers.

On a given datapath, there are several different kinds of register files. On a given datapath, each functional unit can write to the global register file on that datapath and most of the *local* register files on that datapath. However, only some functional units can read from a *local* register file.

- **D1 and D2 units:** These reside on the A-side datapath and can load from and store to memory. Two 64-bit loads can execute in parallel. Two 64-bit stores can execute in parallel. A 64-bit and vector-width load can execute in parallel with a 64-bit or vector-width store. It is not possible for two vector-width stores to execute in parallel or two vector-width loads to execute in parallel.
- **L1, S1, M1, and N1 units:** These are general-purpose functional units, handling a varied mix of scalar and small vector computation. The M1 and N1 functional units perform various multiplication instructions.
- **L2, S2, M2, and N2 units:** These are also general-purpose functional units, and can operate on full-width vector data. The M2 and N2 functional units perform various multiplication instructions.
- **B unit:** This unit handles indirect branches and calls.
- **C unit:** This unit performs permutations and shuffles of data.
- **P unit:** This unit computes predicates used to mask off vector lanes so particular lanes are not computed or are not stored to memory.

In addition to the D1 and D2 units providing CPU access to the memory hierarchy, the C7100 DSP has two *streaming engines* that facilitate a fast path to obtain data from memory. A *streaming engine* is a hardware feature that allows you (or the compiler) to specify a pattern of memory addresses to obtain from memory. The streaming engine will do its best to pre-fetch that data from the memory hierarchy into a scratchpad memory close to the CPU, to minimize CPU stalls due to cold cache misses.

CHAPTER

THREE

PROFILING CODE

Determining which areas of the code consume the most cycles and then optimizing those areas, is an iterative process that should be performed after the code is functionally correct. Determining which areas of the code consume the most cycles is done via a technique called profiling. This chapter provides an overview of various profiling tools that may be available to you.

Contents:

3.1 Using Trace Analyzer

Trace Analyzer is provided by Code Composer Studio (CCS) to enable non-intrusive debug and analysis of system activity.

The TI target device sends data about the target's actions over a dedicated port. A Trace Receiver, JTAG emulator, or Embedded Trace Buffer (ETB) must be connected to or embedded within the device.

Many different kinds of system activity can be monitored, such as function duration, reads and writes to memory, and cache stalls. More information can be found by searching for *Trace Analyzer* and *Function Profiling* in the CCS help. Also consult your device documentation.

3.2 Using the Profile Clock Feature in CCS

The Profile Clock feature in CCS allows you to measure cycles between two points. The following links provide details on using this feature:

- CCS Debug Overview
- Counting Cycles

3.3 Using the Time-Stamp Counter (TSC) Register

The C7000 Compiler Tools provide a way to access the Time-Stamp Counter (TSC) register on the device. The value in the 64-bit Time-Stamp Counter Register is incremented by one on every clock cycle. The following program shows use of the `__TSC` register.

```
#include <stdio.h>
#include <c7x.h>

int main()
{
    unsigned long start_time = __TSC;

    // Print out numbers 1 through 5
    for (int i = 1; i <= 5; i++)
        printf("%d\n", i);

    unsigned long stop_time  = __TSC;

    // Calculate and display the # cycles taken
    printf("Number of clock cycles elapsed is %lu\n",
           stop_time - start_time);

    return 0;
}
```

3.4 Using the Time-Stamp Counter (TSC) Register with Function Entry/Exit Hooks

The Time-Stamp Counter can be combined with the Function Entry/Exit Hooks feature available in the compiler to generate a quick profiler.

When the Entry/Exit hooks feature is enabled using the `--entry_hook` and `--exit_hook` options, the compiler inserts a call to an entry hook on entry to each function in the program. The compiler also inserts a call to an exit hook on exit of each function. Refer to the *C7000 Optimizing C/C++ Compiler User's Guide* (SPRUIG8) for details, and search for the section on *Enabling Entry Hook and Exit Hook Functions*.

The following example uses entry and exit hooks with the `__TSC` register to implement a simple profiler.

hook.h

```
// hook.h

#ifndef __hook__
#define __hook__

#include <stdint.h>
#include <c7x.h>

// An enum to indicate if a timestamp is associated with
// function entry or exit
typedef enum { PD_ENTRY=0, PD_EXIT } PD_Mode;

// Struct for data associated with a single timestamp
typedef struct {
    uint64_t function_address;
    uint64_t timestamp;
    PD_Mode  mode;
} ProfileData;

void __entry_hook(void (*addr)());
void __entry_hook(void (*addr)());
void ProfileData_init();
void ProfileData_print();

static inline uint64_t get_tsc()
{
    // The TSC (Time Stamp Counter) is a read-only register that
    // is set to 0 on device reset and increments by 1 for each
    // C7000 CPU clock cycle. It is readable in all supervisor
    // and user modes.
    return __TSC;
}

#endif /* __hook__ */
```

hook.cpp

```
// hook.cpp

#include "hook.h"
#include <stdio.h>

#define MAX_ENTRIES (64)
```

(continues on next page)

(continued from previous page)

```

// Array to store profile data
ProfileData table[MAX_ENTRIES];
int         index = 0;

// Entry hook function used to record cycle count on entry into function
void __entry_hook(void (*addr)())
{
    if (index >= MAX_ENTRIES) return;

    table[index].function_address = (uint64_t) addr;
    table[index].mode             = PD_ENTRY;
    table[index].timestamp        = get_tsc();
    index++;
}

// Exit hook function used to record cycle count on exit from function
void __exit_hook(void (*addr)())
{
    if (index >= MAX_ENTRIES) return;

    table[index].timestamp        = get_tsc();
    table[index].function_address = (uint64_t) addr;
    table[index].mode             = PD_EXIT;
    index++;
}

void ProfileData_init()
{
    for (int i = 0; i < MAX_ENTRIES; i++)
    {
        table[i].function_address = 0;
        table[i].mode = PD_ENTRY;
        table[i].timestamp = 0;
    }
}

void ProfileData_print()
{
    for (int i = 0; i < MAX_ENTRIES; i++)
```

(continues on next page)

(continued from previous page)

```
        if (table[i].function_address != 0)
        {
            printf("0x%lx, %d, %lu\n",
                   table[i].function_address,
                   table[i].mode,
                   table[i].timestamp);
        }
}
```

Files with functions that need to be profiled are compiled using the `--entry_hook ``, ``--entry_parm=address`, `--exit_hook`, and `--exit_parm=address` options.

When an application is built with the entry/exit hooks in this example, the table is populated with profile data. For example, execution of the code snippet below results in the following table being emitted:

main.cpp

```
// hook.cpp

#include "hook.h"
#include <stdio.h>

#define MAX_ENTRIES (64)

// Array to store profile data
ProfileData table[MAX_ENTRIES];
int         index = 0;

// Entry hook function used to record cycle count on entry into␣
↪function
void __entry_hook(void (*addr)())
{
    if (index >= MAX_ENTRIES) return;

    table[index].function_address = (uint64_t) addr;
    table[index].mode             = PD_ENTRY;
    table[index].timestamp        = get_tsc();
    index++;
}

// Exit hook function used to record cycle count on exit from␣
↪function
void __exit_hook(void (*addr)())
```

(continues on next page)

(continued from previous page)

```
{
    if (index >= MAX_ENTRIES) return;

    table[index].timestamp        = get_tsc();
    table[index].function_address = (uint64_t) addr;
    table[index].mode             = PD_EXIT;
    index++;
}

void ProfileData_init()
{
    for (int i = 0; i < MAX_ENTRIES; i++)
    {
        table[i].function_address = 0;
        table[i].mode = PD_ENTRY;
        table[i].timestamp = 0;
    }
}

void ProfileData_print()
{
    for (int i = 0; i < MAX_ENTRIES; i++)
        if (table[i].function_address != 0)
        {
            printf("0x%lx, %d, %lu\n",
                   table[i].function_address,
                   table[i].mode,
                   table[i].timestamp);
        }
}
```

Table output:

```
0x7004, 0, 8100
0x6fc0, 0, 8516
0x6fc0, 1, 8931
0x6fc0, 0, 9061
0x6fc0, 1, 9476
0x7004, 1, 9603
```

CHAPTER

FOUR

BASIC CODE OPTIMIZATION STRATEGIES AND TECHNIQUES

This section discusses basic code optimization techniques that can be applied to C/C++ code that will run on the C7000 DSP core.

Contents:

4.1 Overview of the Compilation Process

The Texas Instruments C7000 compiler accepts C or C++ source input. When compiling, the compiler proceeds through several stages, as shown in Figure 4.1.

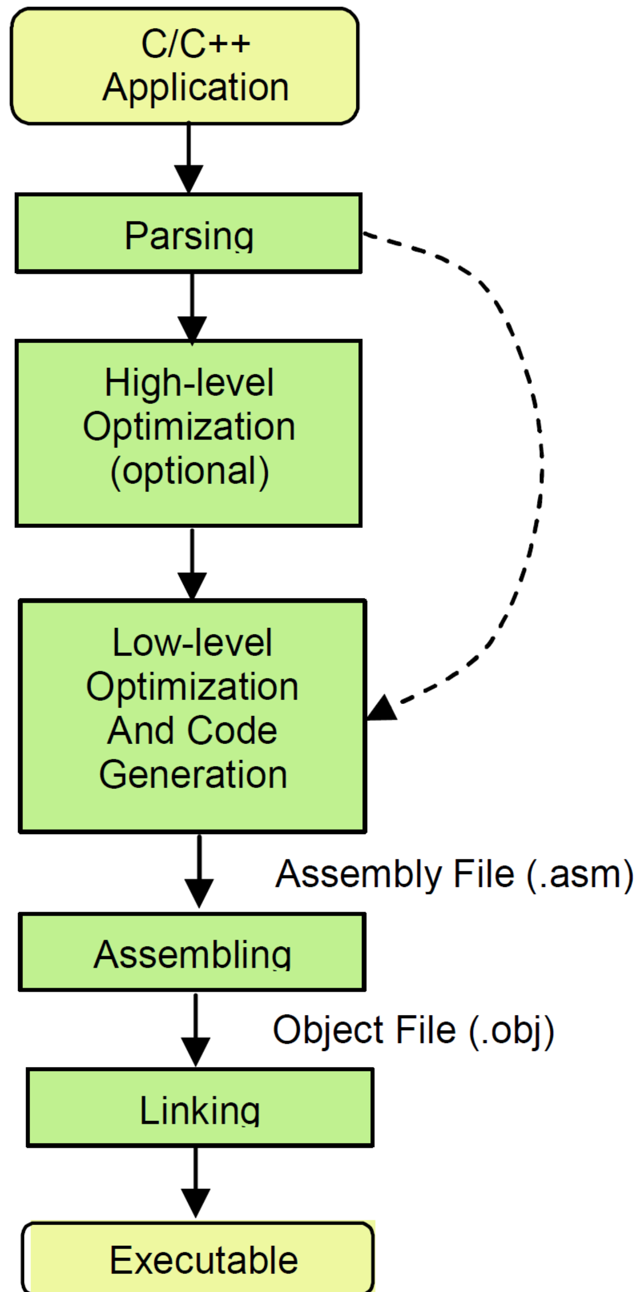

Figure 4.1: C7000 Compiler Processing Stages

First, the source file is parsed to create a high-level intermediate representation that closely resembles the source language, but is more tailored for optimization transformations.

Files and functions (optionally) compiled with some level of optimization pass through the high-level optimizer, which performs function inlining, loop transformations, and other code optimizations.

Next, the high-level intermediate language is translated into a low-level intermediate language,

which closely resembles assembly. The low-level optimizer and code generation pass performs partitioning, register allocation, software pipelining, instruction scheduling, and other optimizations.

The output of the code generation pass is the assembly file, which is assembled into an object file by the assembler and then linked into a library or executable by the linker.

4.2 Selecting Compiler Options for Performance

After your application has been fully debugged and is working properly, it is time to begin the optimization process. First, you need to select appropriate compiler options. The following compiler options affect performance. See the *C7000 C/C++ Compiler User's Guide* (SPRUIG8) for more details on command-line options.

- **--opt_level=3 (-o3)** The compiler performs function-level optimization at `-opt_level=2` and file-level optimization and function inlining at `--opt_level=3`. At both `--opt_level=2` and `--opt_level=3`, the compiler performs various loop optimizations, such as software pipelining, vectorization, and loop coalescing. By default, the `--opt_level` switches optimize for performance. Such optimization can increase code size. If code size is an issue, do not reduce the level of optimization. Instead use the `--opt_for_speed` (`-mf`) switch to change the optimization goal (performance versus code size) and the `-oi` option to control the amount of automatic inlining.
- **--opt_level=4 (-o4)** Consider using this option to perform optimizations across all files at link-time. Using this option can increase compile time significantly. If used for any step, this option must be used at all compilation and linking steps. Source files can be compiled separately, as long as they are all compiled with `--opt_level=4`. This optimization level cannot be used with `--program_level_compile` (`-pm`).
- **--gen_func_subsections (-mo)** Consider using this option if the source code uses many functions that are never called. This option places each function in its own input subsection, so the linker can exclude that function from the executable if it is never referenced. However, this optimization can increase code size, because there are minimum section alignment requirements the compiler must apply.
- **--opt_for_speed=0 (-mf0) or --opt_for_speed=1 (-mf1)** If code size is a concern, use these options when compiling files with functions that are not executed often or are not critical to performance. This tells the compiler to optimize for code size instead of performance. Do not lower the optimization level (`--opt_level`) in an attempt to lower code size.

Do not use the `--disable_software_pipelining` (`-mu`) option if you are concerned about performance. This option turns off software pipelining. Software pipelining is critical to achieving high performance on most inner loops. This option can be a debugging tool, as it makes the assembly code easier to understand. See *Software Pipelining* to learn more about pipelining.

The following options provide additional information for debugging and performance evaluation

purposes:

- **--src_interlist (-s)** This option causes the compiler to emit into the compiler-generated assembly files a copy of what the source code looks like after high-level optimization. This output is placed in the assembly files as comments among the assembly code. The comments output from the optimizer look like C code and show the high-level transformations that have been applied such as inlining, loop coalescing, and vectorization. This option can be useful in helping you understand the assembly code and some of what the compiler is doing to optimize the performance of the code. This option turns on the `--keep_asm` (`-k`) option, so the compiler-generated assembly (.asm) files will not be deleted.
- **--debug_software_pipeline (-mw)** This option emits extra information about software-pipelined loops into the compiler-generated assembly file, including the single-scheduled iteration of the loop. This information is used in loop tuning examples presented later in this document. This option turns on the `--keep_asm` (`-k`) option, so the compiler-generated assembly (.asm) files will not be deleted.
- **--gen_opt_info=2 (-on2)** This option creates 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 providing advice.

4.3 Automatic Use of the Streaming Engine and Streaming Address Generator

The compiler can use the Streaming Engine (SE) and/or the Streaming Address Generator (SA) automatically if the `--auto_stream=no_saving` option is used on C7100 and C7120 devices or the `--auto_stream=saving` option is used on C7504, C7524, and later devices.

If the weighted_vector_sum_v3.cpp example in section *Software Pipelining Example* is compiled with the `--auto_stream=no_saving` option, the following software pipeline information block is generated. (The generated assembly in this example is for C7100.)

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file                : weighted_vector_sum_
↪v2.cpp
;*      Loop source line                  : 7
;*      Loop opening brace source line    : 7
;*      Loop closing brace source line    : 9
;*      Loop Unroll Multiple              : 32x
;*      Known Minimum Iteration Count     : 32
;*      Known Max Iteration Count Factor  : 1
;*      Loop Carried Dependency Bound(^)  : 2
;*      Unpartitioned Resource Bound      : 2
```

(continues on next page)

(continued from previous page)

```
;*      Partitioned Resource Bound       : 2 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*        ii = 2  Schedule found with 4 iterations in parallel
. . .
;*----------------------------------------------------------------
↪-------------*
;*        SINGLE SCHEDULED ITERATION
;*
;*        ||$C$C36||:
;*   0              TICK    ; [A_U]
;*   1              VMPYWW  .N2     VBM1,SE0++,VBL0   ; [B_N2]␣
↪|8|  ^
;*     ||           VMPYWW  .M2     VBM0,SE1++,VBL1   ; [B_M2]␣
↪|8|  ^
;*   2              VMPYWW  .N2     VBM1,SE0++,VBL0   ; [B_N2]␣
↪|8|  ^
;*     ||           VMPYWW  .M2     VBM0,SE1++,VBL1   ; [B_M2]␣
↪|8|  ^
;*   3              NOP     0x2       ; [A_B]
;*   5              VADDW   .L2     VBL1,VBL0,VB0     ; [B_L2]␣
↪|8|
;*   6              VST16W  .D2     VB0,*D0(0)        ; [A_D2]␣
↪|8|
;*     ||           VADDW   .L2     VBL1,VBL0,VB0     ; [B_L2]␣
↪|8|
;*   7              VST16W  .D2     VB0,*D0(64)       ; [A_D2]␣
↪|8| [C0]
;*     ||           ADDD    .D1     D0,0x80,D0        ; [A_D1]␣
↪|7| [C1]
;*     ||           BNL     .B1     ||$C$C36||        ; [A_B] |7|
;*   8              ; BRANCHCC OCCURS {||$C$C36||}    ; [] |7|
```

In this case, the compiler uses SE0 and SE1 to replace the loads that previously set a lower ii bound of 4. With these loads instead being performed with SEs, an ii of 2 is achieved. To use the SEs in the above example, the compiler must configure and open them. The configuration and open actions are shown in comments added by the `--src_interlist` option before the loop:

```
;***    ----------------------    S$1 = __internal_SE_TEMPLATE_
↪1_i_1_i_d_i_d_i_d_i_d_i_d_2_4;
;***    ----------------------    S$1.ICNT0 = C$5 =␣
↪(unsigned)(n+15&0xfffffff0);
;***    ----------------------    __se_open_V0_U32_O(*__se_
↪mem((packed void *)a), 0, S$1);
```

(continues on next page)

(continued from previous page)

```
;***    ----------------------    S$3 = __internal_SE_TEMPLATE_
↪1_i_1_i_d_i_d_i_d_i_d_i_d_2_4;
;***    ----------------------    S$3.ICNT0 = C$5;
;***    ----------------------    __se_open_V0_U32_0(*__se_
↪mem((packed void *)b), 1, S$3);
```

By default, the compiler uses the SE or SA only if using them appears to be profitable and legal.

For profitability, a key consideration is that using the SEs or SAs comes with a processing overhead; the compiler does not necessarily know whether this overhead is profitable. In the example, the MUST_ITERATE pragma indicates the minimum iteration count is 1024, which convinces the compiler that use of SEs or SAs is likely profitable, so the compiler performs the transformation. If the compiler is not using the SE or SA and you want to cause the compiler to use them, indicating the number of iterations with the MUST_ITERATE or PROB_ITERATE pragma can help.

For legality, most reasons for not using the SE or the SA relate to whether an addressing pattern can always be mapped to an SE or SA. These reasons include, but are not limited to:

- Iteration counter (ICNT) values that exceed the range of an unsigned 32-bit type. For example, this occurs in `for (i = 0; i < icnt; i++)` when `i` and `icnt` are 64-bit types.
- DIM values that exceed the range of a signed 32-bit type. For example, this occurs in `data_in[i*dim]` when `dim` is a 64-bit type.
- Additions or multiplies in addressing that exceed the range of a signed 32-bit type. For example, this occurs in `data_in[i*dim]` when `i` or `dim` is a 64-bit type.
- Addressing exceeding the range of INT_MIN to INT_MAX elements. For example, in `int16_ptr[i]` when `int16_ptr` is an `int16 *` and `i` is an `int`, the maximum range is INT_MIN*16 elements to INT_MAX*16 elements.

Each of these are edge cases are unlikely to occur in practice. To allow the compiler to ignore them, use the `--assume_addresses_ok_for_stream` option.

If using the SE or SA is not profitable in practice, you can override the `--auto_stream` and/or `--assume_addresses_ok_for_stream` options for a single function using the FUNCTION_OPTIONS pragma.

If the code explicitly uses the SE or SA in a function, the compiler does not choose to use either the SE or the SA for optimization. In this case, the compiler assumes that the code handles all aspects of optimization with the SE and SA within that function.

For further information on automatic use of the SE and SA and the associated compiler options, see the *C7000 C/C++ Compiler User's Guide* (SPRUIG8).

4.4 Using Existing Optimized Libraries

The Processor SDK for your device likely contains a set of libraries with functions that are optimized for the C7000 DSP core. You should examine the contents of these libraries to determine if an algorithm you need has already been implemented and expertly optimized for the C7000 DSP. Some optimized libraries include:

- **MATHLIB.** Common floating-point math functions, optimized for speed. Contains functions such as cosine, square-root, and divide.
- **DSPLIB.** Common DSP functions, optimized for speed. Contains functions such as FIR, MAX, and cascade-biquad.
- **VXLIB.** Common vision-processing functions, optimized for speed. Contains functions in domains such as image filtering, feature detection, and low-level pixel processing.

See your Processor SDK documentation for more information about the available libraries.

4.5 Signed Types for Iteration Counters and Limits

In order for automatic vectorization to occur, the iteration counters and iteration limits for loops should have signed types. In other words, use `int` rather than `unsigned int`.

The C language standard defines the behavior for unsigned arithmetic overflow, but not for signed arithmetic overflow.

In the unsigned case, an overflowing value will "wrap-around". Therefore, the compiler must assume (in certain cases) that the loop counter may loop around and thus cannot make certain necessary conclusions about the behavior of the loop.

In the signed type case, the compiler can assume the iteration counter will not overflow, because that has undefined behavior according to the C-standard. Thus, the compiler can make certain conclusions about the behavior of the loop and from there may be able to vectorize the loop.

4.6 Use of the restrict Keyword

Careful use of the restrict keyword is a common and important technique when programming the C7000 DSP core. Using the restrict keyword can dramatically improve the performance of an inner loop and is often required to achieve acceptable performance results of an inner loop. However, the restrict keyword imposes a significant responsibility for the user, because misuse of this technique can lead to incorrect results.

This section discusses the restrict keyword and how the use of the restrict keyword is often necessary for optimal performance.

4.6.1 The Restrict Keyword

Many common digital signal processing loops contain one or more load operations, some computation, and then a store operation. Typically, the loads read from an array and the stores store values to a different array. If the compiler cannot prove that the arrays are separate (or at least the accesses do not overlap), the compiler must be conservative and assume that the value written by a store in iteration `i` may be read by a load in iteration `i+1` or `i+2`, etc. The compiler cannot safely issue the load until the store has completed, which prevents the from aggressively overlapping the loop iterations, which can lead to poorly performing inner loops. Therefore, it is important to tell the compiler when the memory accesses cannot refer to the same location; that is, that the object/arrays pointed to by those pointers do not overlap.

We can do this with the use of the `restrict` keyword. This keyword tells the compiler that throughout the lifetime of the variable (array name or pointer name used to access the array), accesses to that object or array will only be made through that array name or pointer name.

Note: Note: This description of the restrict keyword is conservatively simplified. Following this conservatively safe description can never cause incorrect execution, and it is easier to learn and use. The precise definition of the restrict keyword in the C standard is a bit more flexible, but it is subtle and difficult to understand, and situations where the additional utility is appropriate are rarely seen in actual practice. If you wish to learn more about the restrict keyword, see the C standard or Demystifying The Restrict Keyword.

In the load-compute-store case described above, use of the restrict keyword allows the compiler to conclude that the store to memory will not write to the same place where the next iterations' loads will read from. Thus, memory operations in successive iterations can be overlapped, and this often allows the generated code to run faster.

The performance of a software-pipelined loop is dominated by how tightly the compiler is able to overlap successive iterations of the loop. In general, the smaller the *initiation interval* (ii), the tighter and faster the loop will be, because we're keeping more functional units busy simultaneously, maximizing work performed per cycle.

The C function below shows the use of the restrict keyword. After this C function is compiled, the resulting Software Pipeline Information comment block in the generated assembly code will show that when the restrict keyword is used, the initiation interval is two cycles. Without the *restrict* keyword, the initiation interval is 12 cycles. (The reader is encouraged to review the section *Loop-Carried Dependencies* for an explanation why a metric called the *loop-carried dependency bound* is reduced in this case and thus why the initiation interval is reduced.)

Note that the use of the restrict keyword is legal if *a*, *b*, and *out* do not overlap. If they do overlap, the use of the restrict keyword may be incorrect and can lead to incorrect code being generated by the compiler. Therefore, it is critically important that the programmer only use the restrict keyword when it is safe to do so.

```
void weighted_sum(int * restrict a, int * restrict b, int *␣
↪restrict out, int weight_a, int weight_b, int n)
{
    #pragma UNROLL(1)
    #pragma MUST_ITERATE(1024, ,32)
    for (int i = 0; i < n; i++)
    {
        out[i] = a[i] * weight_a + b[i] * weight_b;
    }
}
```

The Texas Instruments C7000 C/C++ Compiler allows the restrict keyword to be used in both C and C++ modes, despite the restrict keyword not being part of the C++ standards.

Note: Note: The use of the restrict keyword is a promise from the programmer. If you use the restrict keyword where the promise isn't true, the compiler will often produce code with undefined behavior--meaning that the code generated by the compiler will produce an incorrect result.

For more information on the restrict keyword, see "The Restrict Keyword" section in the C7000 C/C++ Optimizing Compiler User's Guide.

4.6.2 Compiler-Emitted Restrict Advice

In certain situations, the C7000 C/C++ compiler may detect a loop-carried dependency that is negatively affecting performance. The compiler will emit a performance remark when it detects that an inner loop that is a candidate for software pipelining has a loop-carried dependency that is the result of unknown aliasing between an input pointer (a load) and an output pointer (a store). In these situations, the compiler will emit a performance remark to the console noting the issue, where it occurs, and where the addition of the retrict keyword may help. *It is up to the user to determine if the addition of the restrict keyword is legal in that particular situation.*

4.6.3 Run-Time Alias Disambiguation

Under certain limited circumstances, the compiler may generate two loops: one that assumes two pointers are not aliased and one that assumes the two pointers are aliased. It generates a run-time check to determine if the two pointers alias. This optimization is called *run-time alias disambiguation*. The advantage is that the loop that assumes no-aliased pointers can usually software pipeline at a much smaller initiation interval, leading to improved performance of the loop.

The compiler cannot always perform run-time alias disambiguation due to considerations that are too technical to describe here. In addition, certain further optimizations such as nested loop coa-

lescing are inhibited when the compiler produces two different loops with a run-time alias check, so it is best to use the restrict keyword whenever legally possible.

For further discussion and details regarding identifying and eliminating loop-carried dependencies, consult the following references:

- TMS320C6000 Programmer's Guide (SPRU198K), Section 2.2.2 "Memory Dependencies"
- Hand-Tuning Loops and Control Code on the TMS320C6000 (SPRA666), Section 4.1, "Using restrict qualifiers, MUST_ITERATE pragmas, and _nasserts()"

4.6.4 Loop-Carried Dependencies

The performance of a software-pipelined loop is dominated by how tightly the compiler is able to overlap successive iterations of the loop. In general, the smaller the initiation interval (ii), the tighter and faster the loop will be, because we're keeping more functional units busy simultaneously, maximizing work performed per cycle. Ideally, the ii is no larger than the partitioned resource bound.

When the compiler is not able to optimally overlap successive iterations of the loop, performance suffers. In almost all cases, the culprit is a *loop-carried dependency*. A loop-carried dependency can prevent the execution of iteration i+1 from overlapping with iteration i as well as it otherwise could. When a loop-carried dependency gets too big, it can force the compiler to use a larger ii, hurting the performance of the loop.

When a loop-carried dependency contains too many cycles (relative to the *partitioned resource bound*), it negatively affects the performance of a loop.

A loop-carried dependency arises because there is a *cycle* in the ordering constraints (dependencies) for some set of instructions in a loop. In other words, there is a flow of data from one iteration to a later iteration, meaning that a future iteration is dependent on data that is generated from a previous iteration.

Out of all these cycles in the loop (loop-carried dependences), the *largest* loop-carried dependency cycle is called the *loop-carried dependency bound*. The compiler will state what the loop-carried dependency bound is for any software pipelined loop. This value can be found in the `SOFTWARE PIPELINE INFORMATION` comment block that appears next to the software pipelined loop in the assembly code.

To reduce or eliminate a problematic loop-carried dependency, one must identify the cycle and then find a way to shorten or break it. Let's walk through an example.

The following example shows a loop with a problematic loop-carried dependency.

```
void weighted_sum(int * a, int * b, int * out, int weight_a, int␣
↪weight_b, int n)
{
    #pragma UNROLL(1)
```

(continues on next page)

(continued from previous page)

```
    #pragma MUST_ITERATE(1024, ,32)
    for (int i = 0; i < n; i++)
    {
        out[i] = a[i] * weight_a + b[i] * weight_b;
    }
}
```

The compiler-generated assembly code for this example (shown below) shows that the Loop Carried Dependency Bound in the Software Pipeline Information section of the assembly code is 12 cycles.

```
;*----------------------------------------------------------------
↪-------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file               : weighted_vector_sum_
↪v3.cpp
;*      Loop source line                 : 10
;*      Loop opening brace source line   : 11
;*      Loop closing brace source line   : 13
;*      Known Minimum Iteration Count    : 1024
;*      Known Max Iteration Count Factor : 32
;*      Loop Carried Dependency Bound(^) : 12
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound       : 2 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*        ii = 12 Schedule found with 2 iterations in parallel
. . .
;*----------------------------------------------------------------
↪-------------*
;*        SINGLE SCHEDULED ITERATION
;*
;*        ||$C$C51||:
;*   0              TICK    ; [A_U]
;*   1              LDW     .D2     *D1++(4),BM0      ; [A_D2]␣
↪|12|  ^
;*     ||           LDW     .D1     *D2++(4),BM1      ; [A_D1]␣
↪|12|  ^
;*   2              NOP             0x5               ; [A_B]
;*   7              MPYWW   .M2     BM2,BM0,BL0       ; [B_M2]␣
↪|12|  ^
;*     ||           MPYWW   .N2     BM3,BM1,BL1       ; [B_N2]␣
↪|12|  ^
```

(continues on next page)

(continued from previous page)

```
;*    8                   NOP               0x3                 ; [A_B]
;*   11                   ADDW     .L2      BL1,BL0,B0          ; [B_L2]␣
↪|12|   ^
;*   12                   STW      .D1X     B0,*D0++(4)         ; [A_D1]␣
↪|12|   ^
;*      ||                BNL      .B1      ||$C$C51||          ; [A_B]␣
↪|10|
;*   13                   ; BRANCHCC OCCURS {||$C$C51||}        ; [] |10|
;*-----------------------------------------------------------------
↪-------------*
```

The final software pipelined initiation interval of a software pipelined loop can be no smaller than either the Loop Carried Dependency Bound or the Partitioned Resource Bound. In other words, when the loop-carried dependency bound is greater than the partitioned resource bound, the software pipelined loop could likely run faster if the loop-carried dependency bound is eliminated or reduced. Therefore, in this example, because the partitioned resource bound is 2 and the loop-carried dependency bound is 12, this code has an issue that should be investigated.

To identify the problem, we need to look at the instructions involved in the loop-carried dependency. These instructions are marked with the caret "^" symbol in the comment block in the compiler-generated assembly file. Notice that the two load instructions and the store instruction are marked with a caret. Because the load instructions appear first with a caret and the store instruction appears last with a caret, it's likely that the compiler thinks that the load instructions (LDW) depend on values stored (via the STW instruction) from a previous iteration. This is likely because the compiler cannot prove the stores are writing to an area of memory that is independent of the location from which the load instructions are loading values. In absence of information about the locations of the pointers, arrays and address access patterns, the compiler must assume that successive iterations may load from the location of the previous iteration's stores. As previously discussed, careful use of the *restrict* keyword may help.

Loop-carried dependencies can be grouped into two categories, memory and data. The previous example was a memory loop-carried dependence. If a value travels from one iteration to a subsequent iteration in a register, it is called a data loop-carried dependency. In the case of a data loop-carried dependency, the restrict keyword will not help. Eliminating data loop-carried dependences can be difficult, and involve finding ways to eliminate the data dependence from one iteration to another.

See *Hand-Tuning Loops and Control Code on the TMS320C6000* (SPRA666) for more about loop-carried dependencies and how to identify them.

4.7 Function Calls and Inlining

In some instances, functional calls inhibit optimization. For instance, a loop containing a function call will not be software pipelined if the compiler is not able to inline the called function. In order to enable optimizations such as software pipelining, it may be necessary to define the called function in one of these ways:

- In the same source file as the call with the "inline" keyword
- In a .h file included with the #include preprocessor directive, with the called function using the keywords "static inline"

The compiler performs some amount of automatic inlining at the --opt_level=3 and --opt_level=4 optimization levels.

See *Software Pipelining* to learn more about pipelining.

4.8 MUST_ITERATE and PROB_ITERATE Pragmas and Attributes

The compiler can often generate faster code when the compiler knows how many times a loop will execute. Adding this information via the MUST_ITERATE and PROB_ITERATE pragmas and the TI_must_iterate and TI_prob_iterate C++ attributes can help the compiler:

- Determine if it is profitable to vectorize a loop
- Determine if it is profitable to perform certain loop optimizations and loop-nest optimizations
- Determine if a redundant loop is needed (see Redundant Loops, below)

Before vectorizing a loop, the compiler tries to determine if the change will improve performance. It is helpful if the compiler has information about the iteration counts of the loop so the compiler can make better predictions about the profitability of vectorization. In the same way, the compiler also tries to determine if certain loop optimizations and loop-nest optimizations will be profitable and so information about the iterations counts of the loops can be helpful to the compiler.

Note: Note: Do not provide incorrect information about the iteration count in the MUST_ITERATE pragma or TI_must_iterate C++ attribute. If incorrect information is specified in this pragma/attribute, the compiler may create code that produces unexpected and incorrect behavior.

Redundant Loops: In some cases, if the compiler does not know how many times a loop will execute, the compiler generates two different versions of the loop. Software pipelined loops often must execute a certain minimum number of iterations to be legal to execute. If the iteration count of

the loop is less than this *minimum safe iteration count*, the compiler generates a run-time iteration count check and branches to either the software pipelined version of the loop, or a *duplicate loop*. That is, the compiler generates a "regular" version of the loop (that executes much more slowly).

The minimum safe iteration count depends on how many iterations were scheduled in parallel and how effectively the compiler was able to perform an optimization called *stage collapsing*. See the *Minimum Safe Iteration Count* and *Stage Collapsing and Load Speculation* sections for more information.

The Software Pipeline Information in the comment block in the assembly file specifies the minimum safe iteration count (iteration count) of the loop and states whether the compiler has generated a duplicate loop.

Because the compiler must sometimes generate a redundant loop and the control code necessary to choose between the two loops, it is helpful to tell the compiler the minimum iteration count of the loop with a MUST_ITERATE pragma or TI_must_iterate attribute when it is known, as the redundant loop may not be necessary. This can improve performance, especially when the loop is enclosed within an outer loop and if the compiler can then perform loop collapsing or other loop optimizations with the outer and inner loops.

The following example shows redundant loop generation information in the Software Pipeline Information section of the assembly comment block. See *Software Pipelining* for more information on the Software Pipeline Information comment block.

```
;*       Redundant loop generated
```

4.9 Floating-Point Division

Floating-point division operations can be costly. Often, a division operation results in a run-time-support call to a predefined function that implements floating-point division. Such calls prevent software pipelining.

If your code divides by a `constant` that is known at compile time, consider pre-calculating the `1/constant` value and replacing the division operation with a multiplication by `1/constant`. The compiler automatically performs this optimization only if the `1/constant` value can be precisely represented in an IEEE-754 float or double.

4.10 C++ Features to Use and Avoid

Some C++ features incur in a run-time penalty. Other features are handled completely at compile-time and thus do not cause a run-time penalty. A full discussion of which C++ features do and do not incur a run-time penalty is outside the scope of this document; discussion is available from several sources on the internet and in print.

Some features that do incur a run-time penalty are so useful in providing the desired level of abstraction and/or safety, that you should consider using them anyway. Here are some guidelines for some of the more commonly-used features:

These features have potential run-time overheads. Consider whether the benefits are worth the cost:

- Calls to `new()`, although this is essentially no more or less expensive than `malloc()`
- Use of the Standard Template Library (STL), mainly due to hidden calls to `new()`
- Exceptions / exception handling
- Run-Time Type Information (RTTI)
- Multiple inheritance
- Virtual functions (although the run-time cost is usually small)

Use these features freely, as they have little to no run-time overhead:

- Templates
- Operator overloading
- Function overloading
- Inlining
- Well-designed inheritance. In particular, calling a member function of a derived class incurs no penalty if the object type is known at compile-time.

The following features improve performance and should be used where possible:

- Use of `const`
- Use of `constexpr`
- Passing objects by-reference instead of passing objects by-value
- Constructs and expressions that can be evaluated at compile-time versus run-time

4.11 Memory Optimizations

Optimizations that improve the loading and storing of data are often crucial to the performance of an application. A detailed examination of useful memory optimizations on Keystone 3 devices is outside the scope of this document. However, the following are the most common optimizations used to aid memory system throughput and reduce memory hierarchy latency.

- **Blocking:** Input, output, and temporary arrays/objects are often too large to fit into Multicore Shared Memory Controller (MSMC) or L2 memory. For example, when performing an algorithm over an entire 1000x1000 pixel image, the image is too large to fit into most or all configurations of L2 memory, and the algorithm may thrash the caches, leading to poor performance. Keeping the data as close to the CPU as possible improves memory system performance, but how do we do this when the image is too large to fit into the L2 cache? Depending on the algorithm, it may be useful to use a technique called "blocking," in which the algorithm is modified to operate on only a portion of the data at a given time. Once that "block" of data is processed, the algorithm moves to the next block. This technique is often paired with the other techniques in this list.
- **Direct Memory Access (DMA):** Consider using the asynchronous DMA capabilities of the device to move new data into MSMC memory or L2 memory and DMA to move processed data out. This frees the C7000 CPU to perform computations while the DMA is readying data for the next frame, block, or layer.
- **Ping-Pong Buffers:** Consider using ping-pong memory buffers so that the C7000 CPU is processing data in one buffer while a DMA transfer is occurring to/from another buffer. When the C7000 CPU is finished processing the first buffer, the algorithm switches to the second buffer, which now has new data as a result of a DMA transfer. Consider placing these buffers in MSMC or L2 memory, which is much faster than DDR memory.

CHAPTER

FIVE

UNDERSTANDING COMPILER OPTIMIZATION

Before you can interpret the assembly code and the software pipelining information within, it helps to understand some of what the compiler is trying to do with the C/C++ source code as it compiles source code into assembly code.

Contents:

5.1 Software Pipelining

This chapter introduces the concept of software pipelining, which is a compiler optimization that is essential for achieving high-performance on a VLIW architecture like the C7000 DSP.

5.1.1 Software Pipelining Motivation

As described in *C7000 Digital Signal Processor CPU Architecture*, the C7000 DSP has a very-long instruction word (VLIW) architecture that contains 13 fully-pipelined functional units on which instructions can execute. Up to 13 instructions can start on any cycle. Keeping these functional units busy executing instructions is key to achieving performance on the C7000 DSP core.

Keeping all of these functional units busy can be challenging. *Software pipelining* is a technique used on inner loops by a compiler to attempt to keep the functional units as busy as possible. The C7000 compiler performs software pipelining automatically on inner loops, when appropriate compiler options are used.

Without software pipelining, inner loops will likely suffer from poor performance on a VLIW architecture. This is because without software pipelining, only a single iteration of the loop is executed at a time.

Figure 5.1 shows a conceptual representation of an inner loop that is not software pipelined. The loop in this example has five equal sized sections, A through E. The arrow represents repeated execution of the loop body (via a conditional branch instruction at the end of section E).

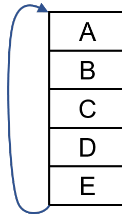

Figure 5.1: Inner Loop, Not Pipelined

Figure 5.2 shows that same loop's execution over time of three iterations. The y-axis represents time and the x-axis visualizes which iteration of the loop is being executed. The sections of the loop are executed sequentially, from top to bottom, and then jumps to the top of the loop body after the last part of the loop is completed. If *n* is the number of iterations the loop is to perform, the loop jumps back to the top *n-1* times.

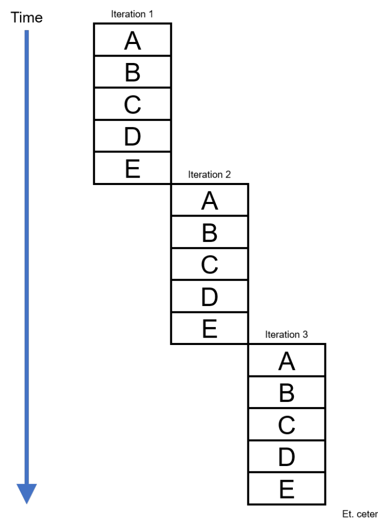

Figure 5.2: Inner Loop, Not Pipelined, 3 Iterations

In a normal loop, there are typically few instructions that can be executed in parallel. This is partly because of dependencies within the loop body from one instruction to the next, as one instruction in the loop is calculating a result that is likely consumed by a subsequent instruction in the loop. This can lead to resource under-utilization. In other words, during the execution of this loop, only one or two of the functional units would be used at any given time, and this would lead to many of the functional units being unused.

5.1.2 Software Pipelining Overview and Terminology

Software pipelining is a technique the compiler uses to overlap successive iterations of a loop in order to more fully utilize the multiple functional units on the CPU. In this manner, the loop executes much faster than a sequentially executed loop.

Figure 5.3 shows the execution of this same example loop after software pipelining has been performed. We can see that rather than waiting for an entire iteration of the loop to complete before starting the next iteration, in this example the 2nd iteration starts only after the 1st iteration's section `A` is complete. In other words, the 2nd iteration starts after only some of 1st iteration is complete. As long as correctness can be preserved, iteration i+1 can start before iteration i finishes. This generally permits a much higher utilization of the machine's resources than might be achieved from other scheduling techniques.

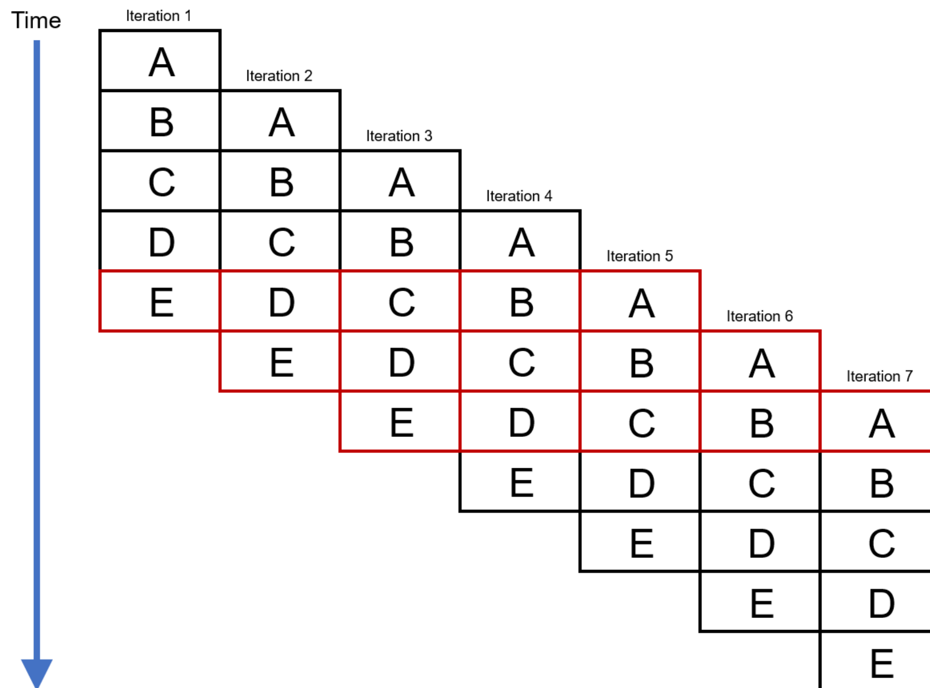

Figure 5.3: Inner Loop, Pipelined, 7 Iterations

Notice in Figure 5.3 that much more work is being performed per unit time in this example. Also note there is a repeating pattern (in the color red) once iteration 5 has started. This pattern continues until iteration 7. This pattern involves each section (A through E) of the loop body being executed at the same time (but from different iterations!). Software pipelining takes advantage of this pattern -- the compiler creates a *kernel* that repeats as often as it necessary to execute the proper number of iterations of the loop. The compiler also creates sections called the prolog and the epilog to start ("pipe-up") and finish ("pipe-down") the software pipeline execution.

Prolog, Kernel, and Epilog

Figure 5.4 shows a conventional diagram of a software pipelined loop with some of the parts identified.

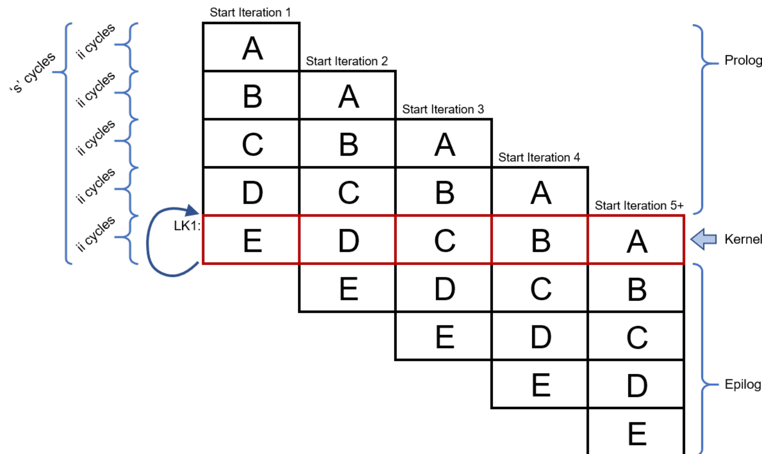

Figure 5.4: Software Pipelined Loop with Components Identified

After software pipelining, the software pipelined loop has three major phases, as shown in Figure 5.4:

- pipe-up (prolog) phase during which overlapped iterations are started.
- steady-state (kernel), repeating phase during which iterations may start, continue, and/or finish
- pipe-down (epilog) phase during which any unfinished iterations are completed

Let's review each phase in more detail.

The *prolog* portion of the software pipelined loop starts a number of iterations in order to get the software pipelined loop going. The number of iterations that the prolog starts is dependent on how many iterations were overlapped during the software pipelining process by the compiler. The prolog executes once.

The *kernel* is a repeating portion of the software pipelined loop. The compiler generates a branch instruction that jumps to the top of the kernel in order to execute as many repetitions of the kernel as is necessary to correctly execute all of the user-mandated iterations of the inner loop. The kernel starts, continues, and finishes iterations, all at the same time. The kernel in the above figure is colored red. The loop back to the top of the kernel is depicted by a curved arrow to the top of the kernel and the assembly label `LK1`.

After the correct number of iterations of the loop have been started (and some iterations likely have finished by this point), the kernel branch will not be taken and thus execution will fall-through instead of branching to the top of the kernel. Thus, execution will proceed into the *epilog* portion

of the software pipelined loop. This epilog section completes any unfinished loop iterations that were started by the prolog or the kernel. The epilog executes once.

Note that some iterations of the executing software pipelined loop may completely execute (start and finish) within the repeating kernel. Earlier iterations may start in the prolog and finish in the kernel. Later iterations may start in the kernel and finish in the epilog. Loops with small iteration counts may have iterations that start in the prolog, continue in the kernel, and finish in the epilog.

Initiation Interval and Single-Scheduled Iteration Length

As shown in Figure 5.4, the number of cycles between starting two successive iterations of the loop is called the *initiation interval*, or `ii`. In a software-pipelined loop, a new iteration is initiated every `ii` cycles. Because the ii has a large effect on a loop's performance, the compiler tries to schedule a loop at the lowest ii possible. The ii of a loop (and other information) is printed in a helpful comment section in the assembly file. See later sections of this document for information on how to keep the assembly file and generate these assembly comments.

As seen in the figure above, length `s` is the length of a *single scheduled iteration* of the software-pipelined loop. This is the amount of time it takes for one iteration of the loop to execute when the loop is software pipelined. A single iteration of the software pipelined loop (`s`) may be much more than `ii` cycles, but there will be multiple loop iterations executing at once, all at a different stage of completion. In an efficient software pipelined loop, `ii` is usually much less than the length `s`.

There are also two other blocks that the compiler adds, one above the prolog block, and one below the epilog block. Often during the process of software pipelining, the compiler needs to add various instructions to facilitate the setup and finalization of a software pipelined loop. These blocks are sometimes able to be merged into surrounding blocks after software pipelining by the compiler, so they may not appear in the final assembly.

The compiler attempts to software pipeline innermost loops. These are loops that do not have any other loops within them. Note that during the compilation process, software pipelining occurs after inlining and after loop transformations that may combine loops. So in certain cases you may see the compiler software pipelining more of your code than you expect.

5.1.3 Software Pipelining Example

The following example shows the source code for a simple weighted vector sum.

```
// weighted_vector_sum.cpp
// Compile with "cl7x -mv7100 --opt_level=3 --debug_software_
↪pipeline
// --src_interlist --symdebug:none weighted_vector_sum.cpp"

void weighted_sum(int * restrict a, int *restrict b, int␣
↪*restrict out,
```

(continues on next page)

(continued from previous page)

```
int weight_a, int weight_b, int n)
{
    #pragma UNROLL(1)
    #pragma MUST_ITERATE(1024, ,32)
    for (int i = 0; i < n; i++)
    {
        out[i] = a[i] * weight_a + b[i] * weight_b;
    }
}
```

To simplify this first software-pipelining example, two pragmas are used:

- The `UNROLL(1)` pragma tells the compiler not to perform vectorization, which is a transformation technique that is demonstrated in the next section.
- The `MUST_ITERATE` pragma conveys information on how many times the loop executes and is explained later in this document. The example uses this pragma to prevent a "duplicate loop" from being generated.

Then we compile this code with the following command:

```
cl7x --opt_level=3 --debug_software_pipeline --src_interlist --
↪symdebug:none weighted_vector_sum.cpp
```

The `--symdebug:none` option prevents the compiler from generating debug information and the associated debug directives in the assembly. This debug information is not relevant to the discussion in this document and if included, would unnecessarily lengthen the assembly output examples shown here. Normally, you would not turn off debug generation as the generation of debug information does not degrade performance.

Because the `--src_interlist` option is used, the compiler-generated assembly file is not deleted and has the following contents:

```
1   ;*----------------------------------------------------------------
    ↪----------------*
2   ;*   SOFTWARE PIPELINE INFORMATION
3   ;*
4   ;*      Loop found in file       : weighted_vector_sum.cpp
5   ;*      Loop source line         : 10
6   ;*      Loop opening brace source line   : 11
7   ;*      Loop closing brace source line   : 13
8   ;*      Known Minimum Iteration Count    : 1024
9   ;*      Known Max Iteration Count Factor : 32
10  ;*      Loop Carried Dependency Bound(^) : 0
11  ;*      Unpartitioned Resource Bound     : 2
```

(continues on next page)

(continued from previous page)

```
;*       Partitioned Resource Bound       : 2 (pre-sched)
;*
;*       Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 7 iterations in parallel
;*
;*       Partitioned Resource Bound(*)    : 2 (post-sched)
. . .
;*----------------------------------------------------------------------------*
;*         SINGLE SCHEDULED ITERATION
;*
;*         ||$C$C36||:
;*    0              TICK     ; [A_U]
;*    1              SLDW     .D1      *D1++(4),BM0       ; [A_D1] |12|
;*      ||           SLDW     .D2      *D2++(4),BM1       ; [A_D2] |12|
;*    2              NOP      0x5      ; [A_B]
;*    7              MPYWW    .N2      BM2,BM0,BL0        ; [B_N] |12|
;*      ||           MPYWW    .M2      BM3,BM1,BL1        ; [B_M2] |12|
;*    8              NOP      0x3      ; [A_B]
;*   11              ADDW     .L2      BL1,BL0,B0         ; [B_L2] |12|
;*   12              STW      .D1X     B0,*D0++(4)        ; [A_D1] |12|
;*      ||           BNL      .B1      ||$C$C36||         ; [A_B] |10|
;*   13      ; BRANCHCC OCCURS {||$C$C36||}     ; [] |10|
;*----------------------------------------------------------------------------*
||$C$L1||:    ; PIPED LOOP PROLOG
;          EXCLUSIVE CPU CYCLES: 8

           TICK    ; [A_U] (R) (SP) <1,0>
||         SLDW    .D1     *D1++(4),BM1       ; [A_D1] |12| (P) <1,1>
||         SLDW    .D2     *D2++(4),BM0       ; [A_D2] |12| (P) <1,1>


```

(continues on next page)

(continued from previous page)

```
            MV      .L2X    A7,B0    ; [B_L2] |7| (R)
    ||      TICK    ; [A_U] (P) <2,0>

            MV      .L2X    A8,B1    ; [B_L2] |7| (R)
    ||      SLDW    .D1     *D1++(4),BM0      ; [A_D1] |12| (P)
↪<2,1>
    ||      SLDW    .D2     *D2++(4),BM1      ; [A_D2] |12| (P)
↪<2,1>

            MV      .S2     B0,BM2   ; [B_S2] (R)
    ||      MV      .L2     B1,BM3   ; [B_L2] (R)
    ||      TICK    ; [A_U] (P) <3,0>


            MPYWW   .N2     BM2,BM1,BL0       ; [B_N] |12| (P)
↪<0,7>
    ||      MPYWW   .M2     BM3,BM0,BL1       ; [B_M2] |12| (P)
↪<0,7>
    ||      SLDW    .D1     *D1++(4),BM0      ; [A_D1] |12| (P)
↪<3,1>
    ||      SLDW    .D2     *D2++(4),BM1      ; [A_D2] |12| (P)
↪<3,1>

            TICK    ; [A_U] (P) <4,0>

            MPYWW   .N2     BM2,BM1,BL0       ; [B_N] |12| (P)
↪<1,7>
    ||      MPYWW   .M2     BM3,BM0,BL1       ; [B_M2] |12| (P)
↪<1,7>
    ||      SLDW    .D1     *D1++(4),BM0      ; [A_D1] |12| (P)
↪<4,1>
    ||      SLDW    .D2     *D2++(4),BM1      ; [A_D2] |12| (P)
↪<4,1>

            MV      .D2     A6,D0    ; [A_D2] (R)
    ||      ADDD    .D1     SP,0xfffffff8,SP  ; [A_D1] (R)
    ||      TICK    ; [A_U] (P) <5,0>

    ;** -----------------------------------------------------------
↪-----------------*
    ||$C$L2||:    ; PIPED LOOP KERNEL
    ;         EXCLUSIVE CPU CYCLES: 2
```

(continues on next page)

(continued from previous page)

```
74
75                 ADDW     .L2      BL1,BL0,B0          ; [B_L2] |12| <0,
↪11>
76     ||          MPYWW    .N2      BM2,BM0,BL0         ; [B_N] |12| <2,7>
↪
77     ||          MPYWW    .M2      BM3,BM1,BL1         ; [B_M2] |12| <2,
↪7>
78     ||          SLDW     .D1      *D1++(4),BM0        ; [A_D1] |12| <5,
↪1>
79     ||          SLDW     .D2      *D2++(4),BM1        ; [A_D2] |12| <5,
↪1>
80
81                 BNL      .B1      ||$C$L2||; [A_B] |10| <0,12>
82     ||          STW      .D1X     B0,*D0++(4)         ; [A_D1] |12| <0,
↪12>
83     ||          TICK     ; [A_U] <6,0>
84     ;** -----------------------------------------------------------
↪----------------*
85     ||$C$L3||:     ; PIPED LOOP EPILOG
86     ;           EXCLUSIVE CPU CYCLES: 7
87     ;**      -----------------------    return;
88
89                 ADDD     .D2      SP,0x8,SP; [A_D2] (O)
90     ||          LDD      .D1      *SP(16),A9          ; [A_D1] (O)
91     ||          ADDW     .L2      BL1,BL0,B0          ; [B_L2] |12| (E)
↪<4,11>
92     ||          MPYWW    .N2      BM2,BM0,BL1         ; [B_N] |12| (E)
↪<6,7>
93     ||          MPYWW    .M2      BM3,BM1,BL0         ; [B_M2] |12| (E)
↪<6,7>
94
95                 STW      .D1X     B0,*D0++(4)         ; [A_D1] |12| (E)
↪<4,12>
96                 ADDW     .L2      BL1,BL0,B0          ; [B_L2] |12| (E)
↪<5,11>
97                 STW      .D1X     B0,*D0++(4)         ; [A_D1] |12| (E)
↪<5,12>
98                 ADDW     .L2      BL0,BL1,B0          ; [B_L2] |12| (E)
↪<6,11>
99                 STW      .D1X     B0,*D0++(4)         ; [A_D1] |12| (E)
↪<6,12>
100
```

(continues on next page)

(continued from previous page)

```
101                 RET      .B1      ; [A_B] (O)
102     ||          PROT     ; [A_U] (E)
103
104                 ; RETURN OCCURS {RP}      ; [] (O)
```

This assembly output shows the software pipelined loop from the compiler-generated assembly file along with part of the software pipelining information comment block, which includes important information about various characteristics of the loop.

If the compiler successfully software pipelines a loop, the compiler-generated assembly code contains a software pipeline information comment block that contains a message about "ii = xx Schedule found with yy iterations in parallel". The *initiation interval*, (`ii`), is a measure of how often the software pipelined loop is able to start executing a new iteration of the loop. The smaller the initiation interval, the fewer cycles it will take to execute the entire loop. The software-pipelined loop information also includes the source lines from which the loop originates, a description of the resource and latency requirements for the loop, and whether the loop was unrolled, as well as other information described below.

In this example, the achieved initiation interval (ii) is 2 cycles, and the number of iterations that will run in parallel is 7.

The comment block finishes with a *single-scheduled iteration* view of the software pipelined loop. The single-scheduled iteration view of the software pipelined loop allows you to see how the compiler transformed the code and how the compiler scheduled one iteration of the software pipelined loop in order to overlap iterations in software pipelining. See *Software Pipeline Information Comment Block* for more information on how to interpret the information in this comment block.

5.1.4 Software Pipeline Information Comment Block

This section provides an explanation of the Software Pipeline Information comment block added to the assembly output for each software pipelined loop. In order to keep the assembly files, use the `--keep_assembly` or `--src_interlist` compiler options.

Some of the information below is placed in the assembly files by default. Other information is added only when the `--debug_software_pipeline` compiler option is specified.

By understanding the feedback that is generated when the compiler pipelines a loop, you may be able tune your C code to obtain better performance.

Dependency and Resource Bounds

The second stage of software pipelining involves collecting loop resource and dependency graph information. The results of Stage 2 are shown in the Software Pipeline Information comment block as follows:

```
;*      Loop Carried Dependency Bound(^) : 2
;*      Unpartitioned Resource Bound     : 12
;*      Partitioned Resource Bound       : 12 (pre-sched)
```

The statistics provided in this section of the block are:

- Loop Carried Dependency Bound: The distance of the largest loop carry path, if one exists. A loop carry path occurs when one iteration of a loop writes a value that must be read in a future iteration. Instructions that are part of the loop carry bound are marked with the ^ symbol. The number shown for the loop carried dependency bound is the minimum iteration interval due to a loop carry dependency bound for the loop.

  If the Loop Carried Dependency Bound is larger than the Resource Bounds, there may be an inefficiency in the loop, and you may be able to improve performance by conveying additional information to the compiler. Potential solutions for this are discussed in the section, *Use of the restrict Keyword*.

- Unpartitioned Resource Bound: The best-case resource bound minimum initiation interval (`mii`) before the compiler has partitioned each instruction to the A or B side. The unpartitioned resource bound is a measure of the minimum achievable ii when mapping the loop's instructions onto the available functional units of the CPU, ignoring dependences. The most used resource constrains the minimum initiation interval. For instance, if four instructions require a .D unit, they require at least two cycles to execute (four instructions / two parallel .D units = 2 cycles). The partitioned resource bound is a lower-bound on how few cycles it takes to make the loop's instructions onto the available functional units of the CPU, ignoring any dependences.

- Partitioned Resource Bound (pre-sched, post-sched): The minimum initiation interval (`mii`) after instructions are partitioned to the A and B sides, ignoring dependences. Some instructions can only execute on the B side. Fore example, a vector instruction with inputs and/or outputs greater than 64 bits must execute on the B side as only the B side has the capability to execute vector instructions that are wider than 64-bits. Pre-scheduling and post-scheduling partioned resource bound values are given. The post-scheduling value is the partitioned resource bound after scheduling occurs. Scheduling sometimes adds instructions, which may affect the resource bound.

The starting initiation interval of the compiled loop is always equal to the maximum of: - the largest loop carried dependence bound of the loop and - the partitioned resource bound.

The compiler will start scheduling attempts at the starting ii. However, the compiler may not *find* a schedule at the starting ii due to various factors and thus the scheduled ii may be higher than the starting ii.

Initiation Interval (ii) and Iterations

The following information is provided about software pipelining attempts:

- Initiation interval (ii): In the example, the compiler was able to construct a software pipelined loop that starts a new iteration every 13 cycles. The smaller the initiation interval, the fewer cycles it will take to execute the loop.
- Iterations in parallel: When in the steady-state (kernel), the example loop is executing different parts of three iterations at the same time. This means that before iteration `n` has completed, iterations `n+1` and `n+2` have begun.

```
;*       Searching for software pipeline schedule at ...
;*         ii = 12 Cannot allocate machine registers
. . .
;*         ii = 12 Register is live too long
;*         ii = 13 Schedule found with 3 iterations in parallel
```

Resources Used and Register Tables

The Resource Partition table summarizes how the instructions have been assigned to various machine resources and how they have been partitioned between the A and B side. Examples are shown below.

An asterisk (*) marks entries that determine the resource bound value (that is, the maximum `mii`). Because many C7000 instructions can execute on more than one functional unit, the table breaks the functional units into categories by possible resource combinations.

- Individual Functional Units (.L, .S, .D, .M, .C units, etc.) show the total number of instructions that specifically require that unit. Instructions that can operate on multiple functional units are not included in these counts.

```
;*       .S units 0         0
;*       .M units 4        12*
. . .
```

- Grouped Functional Units (.M/.N, .L/.S, .L/.S/.C, etc) show the total number of instructions that can execute on all of the listed functional units. For example, if the .L/.S line shows an A-side value of 14 and a B-side value of 12, it means that there are 14 instructions that will execute on either .L1 or .S1 and 12 instructions that will execute on either .L2 or .S2.

```
;*       .L/.S units        1         8
;*       .L/.S/.C units     0         0
. . .
```

- .X cross paths shows the number of cross path buses needed to move data from one datapath to another (A-to-B or B-to-A).

```
;*       .X cross paths    13*        0
```

- Bound: shows the minimum `ii` at which the loop can software pipeline when only considering instructions that can operate on the set of functional units listed on that line. For example, if the .L .S .LS line shows an A-side value of 3 and a B-side value of 2, it means that there are enough instructions that need to go on .L and .S that require .L1 and .S1 for three cycles in the software pipeline schedule and .L2 and .S2 for two cycles in the software pipeline schedule. Note that the .L .S .LS notation means we take into account instructions that can go only on the .L unit or can go only on .S or can go on either .L or .S.

```
;*       Bound(.L .S .LS)   1         4
```

- Register Usage Tables When the `--debug_software_pipeline` compiler option is specified, the software pipeline information comment block in the assembly code will show which CPU registers are used on each cycle of the software pipelined kernel. It is difficult to use this information to improve the performance of the loop, but the information can give you an idea of how many registers are active throughout the loop.

```
;*          Regs Live Always    :  6/ 1/ 4/
;*          Max Regs Live       : 56/26/29/
;*          Max Cond Regs Live  :  0/ 0/ 0/
```

Loop and Iteration Count Information

If the compiler qualifies the loop for software pipelining, the first few lines look like the following example:

```
;*----------------------------------------------------------------
↪-------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file       : s.cpp
;*      Loop source line         : 5
;*      Loop opening brace source line   : 6
;*      Loop closing brace source line   : 8
;*      Known Minimum Iteration Count    : 768
;*      Known Maximum Iteration Count    : 1024
;*      Known Max Iteration Count Factor : 256
```

The loop counter is called the "iteration counter" because it is the number of iterations through a loop. The statistics provided in this section of the block are:

- **Loop found in file, Loop source line, Loop opening brace source line, Loop closing brace source line:** Information about where the loop is located in the original C/C++ source code.
- **Known Minimum Iteration Count:** The minimum number of times the loop might execute given the amount of information available to the compiler.
- **Known Maximum Iteration Count:** The maximum number of times the loop might execute given the amount of information available to the compiler.
- **Known Max Iteration Count Factor:** The maximum number that will divide evenly into the iteration count. Even though the exact value of the iteration count is not deterministic, it may be known that the value is a multiple of 2, 4, etc., which may allow more aggressive packed data/SIMD optimization.

The compiler tries to identify information about the loop counter such as minimum value (known minimum iteration count), and whether it is a multiple of something (has a known maximum iteration count factor).

If a Max Iteration Count Factor greater than 1 is known, the compiler might be more aggressive in packed data processing and loop unrolling optimizations. For example, if the exact value of a loop counter is not known but it is known that the value is a multiple of some number, the compiler may be better able to unroll the loop to improve performance.

Minimum Safe Iteration Count

The *minimum safe iteration count* is always displayed in the software pipeline information comment block. This value is the minimum number of iterations that the loop must execute in order for the generated software pipeline loop to be safely used. If the value of the loop's run-time iteration count is less than the minimum safe trip count, a non-software pipelined version of the loop will be executed. This non-software pipelined version of the loop is called a *redundant loop*.

If the compiler doesn't know about the loop's minimum iteration count (either through analysis or through a user-supplied MUST_ITERATE pragma in the source code), the compiler may have to generate a redundant loop in addition to the software pipelined loop.

Note that the minimum safe iteration count is a value that is calculated after software pipelining has occurred. Thus, this value is affected by vectorization, loop unrolling, and other loop transformations that may have occurred before software pipelining. Therefore, this value may not represent the number of iterations of the original loop in the source code. See the rest of this chapter for more details on potential loop transformations the compiler may perform.

See the *MUST_ITERATE and PROB_ITERATE Pragmas and Attributes* section for more information on Redundant Loops and the use of the `MUST_ITERATE` pragma to potentially eliminate the need for the compiler to generate a redundant loop.

```
;*      Redundant loop generated

;*      Minimum safe iteration count   : 3 (after unrolling)
```

Stage Collapsing and Load Speculation

In some cases, the compiler can reduce the *minimum safe iteration count* (see previous section) of a software pipelined loop through a transformation called *stage collapsing*. Information on stage collapsing is displayed in the Software Pipeline Information comment block when the `--debug_software_pipeline` option is specified. An example is shown below.

Stage collapsing always helps reduce code size. Stage collapsing is usually beneficial for performance, because it can lower the minimum safe iteration count for the software pipelined loop so that when the loop executes only a small number of times, it is more likely the (faster) software pipelined loop can be executed and execution does not have to be transferred to the duplicate loop (which is slower and not-software pipelined).

```
;*      Epilog not entirely removed
;*      Collapsed epilog stages        : 2
;*
;*      Prolog not removed
;*      Collapsed prolog stages        : 0
;*
;*      Max amt of load speculation    : 128 bytes
```

The feedback in the example above shows that two epilog stages were collapsed. However, the compiler was not able to collapse any prolog stages and thus was not able to reduce the minimum safe iteration count of the software pipelined loop down to one (which is the best-case). There are complex technical reasons why a software pipelined loop prolog or epilog may not be removed, and it is difficult for a programmer to affect this outcome.

When performing stage collapsing, the compiler may generate code that executes load instructions *speculatively*, meaning that the result of the load might not be used. In cases where the compiler needs to speculatively execute load instructions, it only does so with load instructions that will not cause an exception if the address accessed is outside the range of legal memory. The feedback "Max amt of load speculation" specifies how far outside the range of normal address accesses the load speculation will access.

Constant Extensions

Each execute packet can hold up to two constant extensions.

```
;*       Constant Extension #0 Used [C0]   : 10
;*       Constant Extension #1 Used [C1]   : 10
```

Constant extension slots are for use by instructions in the execute packet if an instruction's operand constant is too large to fit in the encoding space within the instruction. For instructions that have a constant operand, the encoding space for the constant is usually only a few bits. If a constant will not fit in those few bits, the compiler may use a constant extension slot.

The "Constant Extension #n Used" feedback shows the number of constant extension slots used for each of the C0 and C1 slots.

Memory Bank Conflicts

The compiler has limited understanding of the memory bank structure of the cache hierarchy and the alignment of the objects being accessed via memory. Nevertheless, the compiler tries to estimate the effects on performance of an unlucky memory alignment due to memory bank conflicts stalls. It presents this information in the Software Pipeline Information comment block when the `--debug_software_pipeline` option is specified.

```
;*       Mem bank conflicts/iter(est.) : { min 0.000, est 0.000, max 0.000 }
;*       Mem bank perf. penalty (est.) : 0.0%
```

Loop Duration Formula

The compiler also emits a formula for the number of cycles it will take to execute the software pipelined loop in question when the `--debug_software_pipeline` option is specified. Because the compiler often schedules the prolog and/or epilog in parallel with some of the other code surrounding the loop, this formula is not precise when trying to compute the expected number of cycles for an entire function.

```
;*       Total cycles (est.): 13 + iteration_cnt * 4
```

Single Scheduled Iteration Comment Block

Because the iterations of a software-pipelined loop overlap, it can be difficult to understand the assembly code corresponding to the software pipelined loop. By default, a single-scheduled iteration comment block is added to the generated assembly source file at the end of the SOFTWARE PIPELINE INFORMATION comment block for each software pipelined loop. The single scheduled iteration notionally represents one of the vertical columns (thus, one of the overalapped iterations) in the software pipeline diagrams above. Examining this code can make it easier to understand what the compiler has done and in turn makes optimizing the loop easier.

```
;*----------------------------------------------------------------
↪-------------*
;*        SINGLE SCHEDULED ITERATION
;*
;*        ||$C$C51||:
;*    0             TICK    ; [A_U]
;*    1             LDW     .D2     *D1++(4),BM0        ; [A_D2]␣
↪|12|  ^
;*      ||          LDW     .D1     *D2++(4),BM1        ; [A_D1]␣
↪|12|  ^
;*    2             NOP     0x5     ; [A_B]
;*    7             MPYWW   .M2     BM2,BM0,BL0         ; [B_M2]␣
↪|12|  ^
;*      ||          MPYWW   .N2     BM3,BM1,BL1         ; [B_N2]␣
↪|12|  ^
;*    8             NOP     0x3     ; [A_B]
;*   11             ADDW    .L2     BL1,BL0,B0          ; [B_L2]␣
↪|12|  ^
;*   12             STW     .D1X    B0,*D0++(4)         ; [A_D1]␣
↪|12|  ^
;*      ||          BNL     .B1     ||$C$C51||          ; [A_B]␣
↪|10|
;*   13             ; BRANCHCC OCCURS {||$C$C51||}     ; [] |10|
;*----------------------------------------------------------------
↪-------------*
```

5.1.5 Software Pipeline Processing Steps

The C7000 compiler goes through three basic stages when software pipelining a loop. The three stages are:

1. Qualify the loop for software pipelining
2. Collect loop resource and dependency graph information
3. Attempt to software pipeline the loop

By the time the compiler tries to software pipeline an inner loop, the compiler may have applied certain transformations to the code in the loop, and also may have combined adjacent or nested loops.

Stage 1: Qualification

Several conditions must be met before software pipelining is attempted by the compiler. Two of the most common conditions that cause software pipelining to fail at this stage are:

- The loop cannot have too many instructions. Loops that are too big typically require more registers than are available and require a longer compilation time.
- Another function cannot be called from within the loop unless the called function is inlined. Any break in control flow makes it impossible to software pipeline, since multiple iterations are executing in parallel.

If any conditions for software pipelining are *not* met, qualification of the pipeline halts and a disqualification message appears in the assembly code. See the section *Issues that Prevent a Loop from Being Software Pipelined* for more information.

If all conditions for software pipelining are met, the compiler continues to Stage 2.

Stage 2: Collecting Loop and Dependency Information

The second stage of software pipelining involves collecting loop resource and dependency graph information. See the section *Dependency and Resource Bounds* for information about compiler output from this stage.

Stage 3: Software Pipelining Attempts

Once the compiler has qualified the loop for software pipelining, partitioned it, and analyzed the necessary loop carry and resource requirements, it can attempt software pipelining.

The compiler attempts to software pipeline a loop starting at a certain *initiation interval* (`ii`). The compiler may or may not find a legal schedule at a given ii. If the compiler fails to software pipelining at a particular initiation interval, the `ii` is increased, and another software pipelining attempt is made. The increase in the attempted ii can often be seen in the Software Pipeline Information comment block. This process continues until a software pipelining attempt succeeds or `ii` is equal to the length of a scheduled loop with no software pipelining. If `ii` reaches the length of a scheduled loop with no software pipelining, attempts to software pipeline stop and the

compiler generates a non-software pipelined loop. See the section *Software Pipeline Information Comment Block* for more about the information provided during this stage.

If a software pipelining attempt is not successful, the compiler provides additional feedback in the *Software Pipeline Information Comment Block* to help explain why. See the section *Software Pipeline Failure Messages* for a list of the most common software pipeline failures and strategies for mitigation.

Note that if the compiler is not successful in software pipelining a loop, a regular, non-software pipelined loop will be used instead. This means that the source loop will run correctly, but because it does not utilize a software pipelined loop, the loop may run inefficiently.

After a successful software pipeline schedule and register allocation is found at a particular initiation interval, more information about the loop is emitted to the assembly code. See the section *Software Pipeline Information Comment Block* for more information about the information in this section.

5.2 If Statements and Nested If Statements

In order for the compiler to software pipeline a loop (and thus improve performance), the only branch that may occur in a loop is a branch back to the top of the loop. Branches for if-then and if-then-else statements or for other control-flow constructs will prevent software pipelining.

The compiler tries to eliminate branches resulting from control-flow constructs when it can, in a process called *if-conversion*. If-conversion attempts to remove branches associated with if-then and if-then-else statements, by predicating instructions so that they conditionally execute depending on the test in the "if" statement. As long as there are not too many nesting levels, too many condition terms, or too many instructions in the if-then or if-then-else statements, if-conversion usually succeeds.

5.2.1 If-conversion Pseudo-Code Example

Let's walk through a pseudo-code example so we can better understand the if-conversion concept. Say we have the following code-snippet:

```
if (p) x=5; else x=7;
```

The compiler can if-convert this statement into the following:

```
[ p] x = 5
[!p] x = 7
```

Where the `[]` notation in this pseudo-code indicates conditional execution of the instruction. In this case, the condition `p` controls execution of the statement. Only when the condition is true (that is, not zero), the instruction will be executed. The condition is also called a *predicate*.

After if-conversion, the branches are eliminated and the compiler can schedule these statements in any order or in parallel:

```
[ p] x = 5          [!p] x = 7               [ p] x = 5
              or                  or
[!p] x = 7          [ p] x = 5            || [!p] x = 7
```

5.2.2 If-conversion Source Code Example

Now, let's walk through a real-life source code example that demonstrates if-conversion and look at the assembly code that is produced by the C7000 compiler. In order to software pipeline the "for" loop in this C++ code below, if-conversion must be performed.

Note that the pragmas in the code example below are used to prevent the compiler from vectorizing and generating additional code that is not important for this example.

```
// if_conversion.cpp
// Compile with "cl7x -mv7100 --opt_level=3 --debug_software_
↪pipeline
// --src_interlist --symdebug:none if_conversion.cpp"

void function_1(int * restrict a, int *restrict b, int *restrict␣
↪out, int n)
{
    #pragma UNROLL(1)
    #pragma MUST_ITERATE(1024, ,32)
    for (int i = 0; i < n; i++)
    {
        int result;
        if (a[i] < b[i])
            result = a[i] + b[i];
        else
            result = 0;

        out [i] = result;
    }
}
```

After compilation, the single-scheduled iteration of the loop in the software pipeline information comment block looks like the following:

```
;*----------------------------------------------------------------
↪-------------*
;*          SINGLE SCHEDULED ITERATION
```

(continues on next page)

(continued from previous page)

```
;*
;*          ||$C$C65||:
;*    0                 TICK     ; [A_U]
;*    1                 SLDW     .D1      *D2++(4),A1          ; [A_D1]␣
↪|17|   ^
;*      ||              SLDW     .D2      *D1++(4),A2          ; [A_D2]␣
↪|17|   ^
;*    2                 NOP      0x5      ; [A_B]
;*    7                 CMPGEW   .L1      A2,A1,A0 ; [A_L1] |17|   ^
;*    8         [!A0]   ADDW     .D2      A1,A2,D3 ; [A_D2] |17|   ^
;*    9         [ A0]   MVKU32   .S1      0,D3     ; [A_S1] |17|
;*   10                 STW      .D1      D3,*D0++(4)          ; [A_D1]␣
↪|17|
;*      ||              BNL      .B1      ||$C$C65||           ; [A_B] |9|
;*   11                 ; BRANCHCC OCCURS {||$C$C65||}     ; [] |9|
;*-----------------------------------------------------------------
↪-------------
```

The instruction `[!A0] ADDW.D2 A1,A2,D3` represents the "then" part of the if statement. The instruction `[A0] MVK32.S1 0,D3` represents the "else" part of the if statement. The CMPGEW instruction computes the if-condition and puts the result into a predicate register, which is used to conditionally execute the ADDW and MVKU32 instructions. Note that there is no branch associated with the if-then-else statement because it has been removed by the compiler when it performed if-conversion.

5.2.3 Benefits of if-conversion

If-conversion has a couple of benefits on the C7000 architecture. First, branches are eliminated. In general, if there are fewer branches in code executing on C7000, the resulting code will run faster. In addition, when branches are eliminated from inner loops, the compiler is able to software pipeline inner loops. Second, after if-conversion, the compiler can detect that statements from the body of the "then" have no ordering constraints with respect to statements in the body of the "else".

5.2.4 Which if-statements are if-converted

The C7000 compiler if-converts small and medium sized "if" statements. The compiler will if-convert larger if-then and if-then-else statements when an if-then or if-then-else statement is in an inner loop that might software pipeline. It will also if-convert larger if-then and if-then-else statements when larger values of the size/speed tradeoff option are used (`--opt_for_speed`).

The compiler does not if-convert large "if" statements ("if" statements where the "then" or "else" block is long). If a loop contains an "if" statement that was not converted, a message such as the

following is generated:

```
;*----------------------------------------------------------------
↪-------------*
;* SOFTWARE PIPELINE INFORMATION
;* Disqualified loop: Loop contains control code
;*----------------------------------------------------------------
↪-------------*
```

The reason that the compiler does always perform conversion is that when "if" statements are large, if-conversion is not always profitable. For example, consider the following loop containing an "if" statement:

```
for (i=0; i<n; i++)
{
    if (x[i])
    {
        <large "if" statement body>
    }
}
```

If "x[i]" is usually 0, "x" is sparse. If "x[i]" is usually non-zero, "x" is dense. If "x" is sparse and the body of the "if" statement is long, if-conversion is not profitable. However, if "x" is dense, then if-conversion is profitable. Since the compiler does not know anything about "x", it does not automatically if-convert this "if" statement.

See *If-Conversion Improvement* for more about if-conversion.

5.3 Vectorization and Vector Predication

The C7000 instruction set has many powerful single-instruction, multiple-data (SIMD) instructions that can perform multiple operations in a single instruction. To take advantage of this, the compiler tries to *vectorize* the source code when possible and profitable. Vectorization usually involves using vector (SIMD) instructions to perform an operation on several loop iterations of data at a time.

The following example removes the UNROLL pragma and the MUST_ITERATE pragma from the example in the previous section. The UNROLL(1) pragma prevented certain loop-transformation optimizations in the C7000 compiler.

```
// weighted_vector_sum_v2.cpp
// Compile with "cl7x -mv7100 --opt_level=3 --debug_software_
↪pipeline
// --src_interlist --symdebug:none weighted_vector_sum_v2.cpp"
void weighted_sum(int * restrict a, int *restrict b, int␣
↪*restrict out,
```

(continues on next page)

(continued from previous page)

```
                    int weight_a, int weight_b, int n)
{
    for (int i = 0; i < n; i++)
    {
        out[i] = a[i] * weight_a + b[i] * weight_b;
    }
}
```

The following shows the resulting internal compiler code, which has been vectorized. Vectorization by the compiler can be inferred by the "`+= 16`" address increments and "`32x16`" in the names of optimizer temporary variables (to indicate there are 16 32-bit elements in the temporary variable).

```
;***    -----------------------g3:
;*** 6  -----------------------    if ( !((d$1 == 1)&U$33) )␣
↪goto g5;
;*** 6  -----------------------    VP$25 = VP$24;
;***    -----------------------g5:
;*** 7  -----------------------    VP$20 = VP$25;
;*** 7  -----------------------    __vstore_pred_p_P64_S32(VP$20,
↪ &(*(packed int (*)<[16]>)U$47),
*(packed int (*)<[16]>)U$38*VRC$s32x16$001+*(packed int (*)<[16]>
↪)U$42*VRC$s32x16$002);
;*** 6  -----------------------    U$38 += 16;
;*** 6  -----------------------    U$42 += 16;
;*** 6  -----------------------    U$47 += 16;
;*** 6  -----------------------    --d$1;
;*** 6  -----------------------    if ( L$1 = L$1-1 ) goto g3;
```

The software pipeline information block from the resulting assembly file is as follows:

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file                 : weighted_vector_sum_
↪v2.cpp
;*      Loop source line                   : 6
;*      Loop opening brace source line     : 6
;*      Loop closing brace source line     : 8
;*      Loop Unroll Multiple               : 16x
;*      Known Minimum Iteration Count      : 1
;*      Known Max Iteration Count Factor   : 1
;*      Loop Carried Dependency Bound(^)   : 1
;*      Unpartitioned Resource Bound       : 2
```

(continues on next page)

(continued from previous page)

```
;*      Partitioned Resource Bound       : 2 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*        ii = 2  Schedule found with 7 iterations in parallel
...
;*----------------------------------------------------------------
↪-------------*
;*        SINGLE SCHEDULED ITERATION
;*
;*        ||$C$C41||:
;*   0              TICK    ; [A_U]
;*   1              VLD16W  .D1     *D0++(64),VBM0    ; [A_D1]␣
↪|7| [SI]
;*   2              VLD16W  .D1     *D1++(64),VBM0    ; [A_D1]␣
↪|7| [SI]
;*   3              NOP     0x4     ; [A_B]
;*   7              VMPYWW  .N2     VBM2,VBM0,VBL0    ; [B_N2]␣
↪|7|
;*   8              VMPYWW  .N2     VBM1,VBM0,VBL1    ; [B_N2]␣
↪|7|
;*   9              CMPEQW  .L1     AL0,0x1,D3        ; [A_L1]␣
↪|6|  ^
;*  10              ANDW    .D2     D2,D3,AL1; [A_D2] |6|
;*     ||           ADDW    .L1     AL0,0xffffffff,AL0 ; [A_L1]␣
↪|6|  ^
;*  11              CMPEQW  .S1     AL1,0,A0 ; [A_S1] |6|
;*  12      [!A0]   MV      .P2     P1,P0    ; [B_P] |6| CASE-1
;*     ||           VADDW   .L2     VBL1,VBL0,VB0     ; [B_L2]␣
↪|7|
;*  13              VSTP16W .D2     P0,VB0,*A1(0)     ; [A_D2]␣
↪|7|
;*     ||           ADDD    .M1     A1,0x40,A1        ; [A_M1]␣
↪|6| [C1]
;*     ||           BNL     .B1     ||$C$C41||        ; [A_B] |6|
;*  14              ; BRANCHCC OCCURS {||$C$C41||}    ; [] |6|
```

This example compares the output from that in the previous section to show these effects of vectorization:

- The "optimizer" code after several high-level optimization steps, including vectorization. (This "optimizer" code appears in the assembly when using the `-os` compiler option.) The address increments are by 16 and there are optimizer temporary variables with the partial name of 32x16, indicating 16 32-bit elements.

- The "SOFTWARE PIPELINE INFORMATION" comment block in the assembly file shows that the loop has been unrolled by 16x. This may or may not indicate vectorization has occurred, but is often associated with vectorization.
- The software pipelined loop now uses the VMPYWW and VADDW instructions. The 'V' in the instruction mnemonics often (but not always) indicates that the compiler has vectorized a code sequence (using vector/SIMD instructions).
- Larger address increments in load and store instructions can be another clue that vectorization has occurred.

In this loop, the compiler does not know how many times the loop will execute. Therefore in our example, the compiler must not store to memory an entire vector on the last loop iteration if the number of loop iterations is not a multiple of the number of elements in the vector width that was chosen. For example, if the original (unvectorized) loop will execute 40 iterations and the compiler vectorized the loop by 16, the last optimized iteration will compute 16 elements, but only 8 of them should be stored to memory.

The C7000 ISA has certain vector predication features, where a vector predicate affects which lanes of a vector operation should be performed. In this case, a BITXPND instruction generates a vector predicate that is used in a vector-predicate-aware store instruction. This vector store instruction (VSTP16W) uses the vector predicate to prevent storing to memory those elements on the last iteration that were computed only as a result of the vectorization process and would not have been computed or stored in the original loop. The compiler attempts to perform vector predication automatically during the vectorization process. Vector predication helps avoid the need for generating peeled loop iterations, which can inhibit loop nest optimizations.

Note: Note: Vector predicated stores may lead to page faults if the Corepac Memory Management Unit (CMMU) is enabled and the store overlaps an illegal memory page. Any memory range that will be within 63 bytes of an illegal memory page at run-time should be reduced in length in the linker command file. For more information, see the *C7000 C/C++ Compiler User's Guide* (SPRUIG8).

You can avoid vector prediction if you give the compiler information about the number of loop iterations using the MUST_ITERATE pragma. For example, if the loop in the previous example is known to execute only in multiples of 32 and the minimum iteration count is 1024, then the following example improves the generated assembly code:

```
// weighted_vector_sum_v3.cpp
// Compile with "cl7x -mv7100 --opt_level=3 --debug_software_
↪pipeline
// --src_interlist --symdebug:none weighted_vector_sum_v3.cpp"
void weighted_sum(int * restrict a,  int *restrict b,  int␣
↪*restrict out,
int weight_a,      int weight_b,     int n)
```

(continues on next page)

(continued from previous page)

```
{
    #pragma MUST_ITERATE(1024, ,32)
    for (int i = 0; i < n; i++) {
        out[i] = a[i] * weight_a + b[i] * weight_b;
    }
}
```

When compiled, this modified example generates the following software pipeline information block:

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file                 : weighted_vector_sum_
↪v3.cpp
;*      Loop source line                   : 7
;*      Loop opening brace source line     : 7
;*      Loop closing brace source line     : 9
;*      Loop Unroll Multiple               : 32x
;*      Known Minimum Iteration Count      : 32
;*      Known Max Iteration Count Factor   : 1
;*      Loop Carried Dependency Bound(^)   : 0
;*      Unpartitioned Resource Bound       : 4
;*      Partitioned Resource Bound         : 4 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*        ii = 4  Schedule found with 5 iterations in parallel
...
;*-----------------------------------------------------------------
↪-------------*
;*        SINGLE SCHEDULED ITERATION
;*
;*        ||$C$C36||:
;*   0              TICK    ; [A_U]
;*   1              VLD16W  .D1     *D1++(128),VBM0    ; [A_D1]␣
↪|8| [SI][C1]
;*   2              VLD16W  .D1     *D1(-64),VBM0      ; [A_D1]␣
↪|8| [C1]
;*   3              VLD16W  .D1     *D2++(128),VBM0    ; [A_D1]␣
↪|8| [SI][C1]
;*   4              VLD16W  .D1     *D2(-64),VBM0      ; [A_D1]␣
↪|8| [C1]
;*   5              NOP     0x2     ; [A_B]
;*   7              VMPYWW  .N2     VBM2,VBM0,VBL1     ; [B_N2]␣
↪|8|
```

(continues on next page)

(continued from previous page)

```
;*    8                VMPYWW   .N2      VBM2,VBM0,VBL0     ; [B_N2]␣
 ↪|8|
;*    9                VMPYWW   .N2      VBM1,VBM0,VBL2     ; [B_N2]␣
 ↪|8|
;*   10                VMPYWW   .N2      VBM1,VBM0,VBL1     ; [B_N2]␣
 ↪|8|
;*   11                NOP     0x2       ; [A_B]
;*   13                VADDW    .L2      VBL2,VBL1,VB0      ; [B_L2]␣
 ↪|8|
;*   14                VST16W   .D2      VB0,*D0(0)         ; [A_D2]␣
 ↪|8|
;*      ||             VADDW    .L2      VBL1,VBL0,VB0      ; [B_L2]␣
 ↪|8|
;*   15                VST16W   .D2      VB0,*D0(64)        ; [A_D2]␣
 ↪|8| [C0]
;*   16                ADDD     .D2      D0,0x80,D0         ; [A_D2]␣
 ↪|7| [C0]
;*      ||             BNL      .B1      ||$C$C36||         ; [A_B] |7|
;*   17                ; BRANCHCC OCCURS {||$C$C36||}       ; [] |7|
```

Due to the added MUST_ITERATE pragma, the compiler knows that vector predication is never needed and does not perform vector predication. As a result, the compiler removes the CMPEQW, ANDW, VSTP16W, and other instructions associated with the vector predication.

5.4 Automatic Inlining

The compiler sometimes takes functions defined in header files and places the code at the call site. This allows software pipelining in an enclosing loop and thus improves performance. The compiler may also do this to eliminate the cost of calling and returning from a function.

In the following example, the `add_and_saturate_to_255()` function sums two values and caps the sum at 255 if the sum is over 255. This function is called from a function in `inlining.cpp`, which includes the `inlining.h` file via a preprocessor #include directive.

```
// inlining.cpp
// Compile with "cl7x -mv7100 --opt_level=3
//   --debug_software_pipeline --src_interlist"
#include "inlining.h"

void saturated_vector_sum(int * restrict a, int * restrict b,
                          int * restrict out, int n)
{
```

(continues on next page)

(continued from previous page)

```
    #pragma MUST_ITERATE(1024,,)
    #pragma UNROLL(1)
    for (int i = 0; i < n; i++)
    {
        out[i] = add_and_saturate_to_255(a[i], b[i]);
    }
}

// inlining.h
int add_and_saturate_to_255(int a, int b)
{
    int sum = a + b;
    if (sum > 255) sum = 255;

    return sum;
}
```

In this case, the compiler will inline the call to `add_and_saturate_to_255()` so that software pipelining can be performed. You can determine that inlining has been performed by looking at the bottom of the generated assembly file. Here, the compiler places a comment that `add_and_saturate_to_255()` has been inlined. Note that the function's identifier has been modified due to C++ name mangling.

```
;; Inlined function references:
;; [0] _Z23add_and_saturate_to_255ii
```

The inlining can also be seen in the generated assembly code, because there is no CALL instruction to a function in the loop. In fact, because of the inlining (and thus the elimination of the call to a function), the loop can be software pipelined. Software pipelining cannot occur if there is a call to another function in the loop. Note that because of code size concerns, not every call that can be inlined will be inlined automatically. See the *C7000 Optimizing Compiler User's Guide* for more information on inlining.

```
;*----------------------------------------------------------------
↪-------------*
;*        SINGLE SCHEDULED ITERATION
;*
;*        ||$C$C44||:
;*   0                TICK     ; [A_U]
;*   1                SLDW     .D1      *D1++(4),BL0       ; [A_D1]␣
↪|5|
;*   2                SLDW     .D2      *D2++(4),BL1       ; [A_D2]␣
↪|5|
```

(continues on next page)

(continued from previous page)

```
;*    3              NOP     0x5     ; [A_B]
;*    8              ADDW    .L2     BL1,BL0,BL1         ; [B_L2]␣
↪|5|
;*    9              VMINW   .L2     BL2,BL1,B0          ; [B_L2]␣
↪|5|
;*   10              STW     .D1X    B0,*D0++(4)         ; [A_D1]␣
↪|5|
;*      ||           BNL     .B1     ||$C$C44||          ; [A_B]␣
↪|11|
;*   11              ; BRANCHCC OCCURS {||$C$C44||}      ; [] |11|
;*-----------------------------------------------------------------
↪-------------*
```

5.5 Loop Collapsing and Loop Coalescing

The compiler attempts to *collapse* or *coalesce* nested loops if it is legal and can improve performance. A *nested loop* is a set of two loops where one loop resides inside of another enclosing loop. Both collapsing and coalescing involve transforming a nested loop into a single loop. Collapsing takes place when there is no code in the outer loop. Coalescing takes place when there is code in the outer loop.

After the two nested loops are combined into one loop, the code that was in the body of the outer loop must be transformed so that it conditionally executes only when necessary. Collapsing and coalescing can have performance benefits because only one pipe-up and pipe-down are executed when the loop nest is executed, instead of a pipe-down and pipe-up of the inner loop every time the outer loop executes when loop coalescing/collapsing is not performed.

In order to perform loop collapsing or loop coalescing, the combined loop must be able to be software pipelined. This means that the loop nest must not contain function calls. The loops must each have a signed counting iterator that iterates a fixed amount each time. That is, the inner loop must not iterate a different number of times depending on which outer loop iteration execution is in. Also, the outer loop must not contain too much code, otherwise the transformation will not improve performance. If the outer loop carries a memory dependence, loop coalescing and loop collapsing likely will not be performed.

When loop collapsing or loop coalescing take place, the software pipelined loop indicates the beginning loop source line ("`Loop source line`") near the top of the software information comment block. When this source line number references an outer loop, this indicates that the inner loop has been fully unrolled or the compiler has performed loop coalescing or collapsing. In cases of loop coalescing, the compiler uses special instructions, such as NLCINIT, TICK, GETP, and BNL. A description of these hardware features, encompassing what is known as the "NLC", is beyond the scope of this document. More details of the NLC may be found in the *C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator Technical Reference Manual* (SPRUIP0).

CHAPTER

SIX

ADVANCED CODE OPTIMIZATION TECHNIQUES

This chapter discusses advanced code optimization techniques that may be needed to obtain maximum performance from a given section of code.

Contents:

6.1 Streaming Engine

The C7100 CPU has two *streaming engines*. A streaming engine is a feature of the C7000 CPU cores that aids in loading data from memory to the CPU. The streaming engines can significantly improve the performance of the memory hierarchy by prefetching data from memory to a location near the CPU. Prefetching data can significantly reduce the time needed to bring data into the CPU. It may also reduce the number of L1 data cache capacity misses as the L1 cache is bypassed for data accessed through the streaming engine.

The streaming engine supports up to a six-dimensional address access pattern. When the performance bottleneck involves reads from memory (if D unit resource bound dominates or cache misses dominate), consider using one or both of the streaming engines if the access pattern to the objects in memory is known in advance. Streaming engines have the greatest effect when used in conjunction with loops that are vectorized by hand. For more information on the streaming engine and code examples, please see the *C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator Technical Reference Manual* (SPRUIP0), the *C7000 Optimizing C/C++ Compiler User's Guide* (SPRUIG8), and the `c7x_strm.h` file in the `include` directory of the compiler's installation directory.

As of v4.0.0 of the C7000 compiler, the compiler may automatically use the streaming engine, depending on the situation. See the *C7000 Optimizing C/C++ Compiler User's Guide*, Section 4.14, for more information.

6.2 Streaming Address Generator

Use of a *streaming address generator* can help limit the number of instructions required to calculate an address used for a load or store instruction. This in turn can reduce the resource bound of the software-pipelined loop and so can positively affect the initiation interval of the loop (and thus improve performance of the loop). It can also allow loop collapsing or loop coalescing to occur, possibly leading to improved performance of the loop.

There are four streaming address generators on C7100 cores. For more information and code examples, please see the *C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator Technical Reference Manual* (SPRUIP0), the *C7000 Optimizing C/C++ Compiler User's Guide* (SPRUIG8) and the `c7x_strm.h` file in the `include` directory of the compiler installation directory.

As of v4.0.0 of the C7000 compiler, the compiler may automatically use the streaming address generator. See the *C7000 Optimizing C/C++ Compiler User's Guide* (SPRUIG8), Section 4.14, for more information.

6.3 Identifying Software Pipelining Failures and Performance Issues

The subsections that follow explain situations that may prevent loops from being optimized.

6.3.1 Issues that Prevent a Loop from Being Software Pipelined

The following situations may prevent a loop from being eligible for software pipelining. These can be detected by examining the assembly output and the Software Pipeline Information in the comment block.

- Loop contains function calls: Although a software pipelined loop can contain intrinsics, it cannot contain function calls. This includes code that will result in a call to un-inlinable run-time support routines, such as floating-point division. You may attempt to inline small, user-defined functions. See the section *Function Calls and Inlining* for more information.
- Loop contains control code: In some cases, the compiler cannot remove all of the control flow from if-then-else statements or "?:" statements. You may attempt to optimize such situations by using if statements only around code that updates memory and around variables whose values are calculated inside the loop and used only outside the loop.
- Conditionally incremented loop control variable is not software pipelined. If a loop contains a loop control variable that is conditionally incremented, the compiler will not be able to software pipeline the loop.

```
for (i = 0; i < x; i++)
{
    . . .
    if (b > a)
        i += 2
}
```

- Too many instructions. Oversized loops typically cannot be scheduled due to the large number of registers needed. However, some large loops require an undue amount of time for compilation. A potential solution may be to break the loop into multiple smaller loops.
- Uninitialized iteration counter. The loop counter may not have been set to an initial value.
- Cannot identify iteration counter. The loop control is too complex. Try to simplify the loop.

6.3.2 Software Pipeline Failure Messages

Possible software pipeline failure messages provided by the compiler include the following:

- Address increment too large. During software pipelining, the compiler allows reordering of all loads and stores occurring from the same array or pointer. This maximizes flexibility in scheduling. Once a schedule is found, the compiler returns and adds the appropriate offsets and increments/decrements to each load and store. Sometimes, the loads and/or stores end up being offset too far from each other after reordering (the limit for standard load pointers is +/- 32). If this happens, try to restructure the loop so that the pointers are closer together or to rewrite the pointers to use precomputed register offsets.
- Cannot allocate machine registers. After software pipelining and finding a valid schedule, the compiler allocates all values in the loop to specific machine registers. In some cases, the compiler runs out of machine registers in which it can allocate values of variables and intermediate results. If this happens, either try to simplify the loop or break the loop up into multiple smaller loops. In some cases, the compiler can successfully software pipeline a loop at a higher initiation interval (`ii`).
- Cycle Count Too High. Not Profitable. In rare cases, the iteration interval of a software pipelined loop is higher than a non-pipelined loop. In this case it is more efficient to execute the non-software pipelined loop. A possible solution is to split the loop into multiple loops or reduce the complexity of the loop.
- Did not find schedule. Sometimes the compiler simply cannot find a valid software pipeline schedule at a particular initiation interval. A possible solution is to split the loop into multiple loops or reduce the complexity of the loop.
- Iterations in parallel > max. iteration count. Not all loops can be profitably pipelined. Based on the available information for the largest possible iteration count, the compiler estimates that it will always be more profitable to execute a non-software-pipelined version than to

execute the pipelined version, given the schedule found at the current initiation interval. A possible solution may be to unroll the loop completely.

- Iterations in parallel > min. iteration count. Based on the available information on the minimum iteration count, it is not always safe to execute the pipelined version of the loop. Normally, a redundant loop would be generated. However, in this case, redundant loop generation has been suppressed via the `--opt_for_speed=3` or lower option. A possible solution is to add the MUST_ITERATE pragma to give the compiler more information on the minimum iteration count of the loop.
- Register is live-too long. Sometimes the compiler finds a valid software pipeline schedule, but one or more of the values is live too long. The lifetime of a register is determined by the cycle time between when a value is written into the register and the last cycle this value is read by another instruction. By definition, a variable can never be live longer than the `ii` of the loop, because the next iteration of the loop overwrites that value before it is read. After this message, the compiler provides a detailed description of which values are live to long:

```
ii = 11 Register is live too long
|72| -> |74|
|73| -> |75|
```

  The numbers 72, 73, 74, and 75 in this example correspond to line numbers and can be mapped back to the offending instructions. The compiler aggressively attempts to both prevent and fix live-too longs. Techniques you can use to resolve live-too longs have low probabilities of success. Therefore, such techniques are not discussed in this document. In addition, the compiler can usually find a successful software pipeline schedule at a higher initiation interval (`ii`).

6.3.3 Software Pipelining Performance Issues

You can find the following issues by examining the assembly source and the Software Pipeline Information comment block. Potential solutions are given for each condition.

- Large Outer Loop Overhead in Nested Loop. If the inner loop count of a nested loop is relatively small, the time to execute the outer loop can become a large percentage of the total execution time. For cases where this seems to degrade the overall loop nest performance, two approaches can be tried. First, if there are not too many instructions in the outer loop, you may want to give a hint to the compiler that it should coalesce the loop nest. Try using the COALESCE_LOOP pragma and check the relative performance of the entire loop nest. If the COALESCE_LOOP pragma does not work, and the number of iterations of the inner loop is small and do not vary, fully unrolling the inner loop by hand may improve performance of the nested loop because the outer loop may be able to be software pipelined.

See the *C7000 Optimizing C/C++ Compiler User's Guide* (SPRUIG8) for information about pragmas.

Loop Carried Dependency Bound is Larger than the Partitioned Resource Bound

- Loop Carried Dependency Bound is Larger than the Partitioned Resource Bound. If you see a loop carried dependency bound that is higher than the partitioned resource bound, you likely have one of two problems. First, the compiler may think there is a memory dependence from a store to a subsequent load. In this document, see the section, *Use of the restrict Keyword*. The "Memory Dependencies" section of the TMS320C6000 Programmer's Guide (SPRU198) also has more information. Second, a computation in one iteration of the loop may be used in the next iteration of the loop. In this case, the only option is to try to eliminate the flow of information from one iteration to the next, thereby making the iterations more independent of each other.

Two Loops are Generated, One Not Software Pipelined / Duplicate Loop

- Two Loops are Generated, One Not Software Pipelined / Duplicate Loop Generated. If you see the message "Duplicate Loop Generated" in the Software Pipeline Information comment block, or you notice that there is a second version of the loop that isn't software pipelined, it may mean that when the iteration count (iteration count) of the loop is too low, it is illegal to execute the software pipelined version of the loop that the compiler has created. In order to generate only the software pipelined version of the loop, the compiler needs to prove that the minimum iteration count of the loop would be high enough to always safe execute the pipelined version. If the minimum number of iterations of the loop is known, using the MUST_ITERATE pragma to tell the compiler this information may help eliminate the duplicate loop.

There are Memory Bank Conflicts

- There are Memory Bank Conflicts. If the compiler generates two memory accesses in one cycle and those accesses reside within the same memory block in the cache hierarchy, a memory bank stall can occur. To avoid this degradation, memory bank conflicts can be avoided by skewing the two objects starting addresses so the accesses to them start in different memory blocks. One of the ways this can be accomplished is through the use of the DATA_MEM_BANK pragma. The DATA_MEM_BANK pragma only works for global variables. Techniques for other objects can involve using the DATA_ALIGN pragma and differential padding (empty space) at the beginning of arrays. See the *C7000 Optimizing C/C++ Compiler User's Guide* (SPRUIG8) for more information about the DATA_MEM_BANK pragma and DATA_ALIGN pragma.

6.4 If-Conversion Improvement

To mitigate the effect that control-flow statements inside a loop have on whether the loop is software pipelined, the compiler performs *if-conversion* on some if statements, which adds a predicate onto the instructions in the *then* and *else* clauses. See section *If Statements and Nested If Statements* for a primer on if-conversion. Because there are a limited number of machine predicate registers, and because of other factors, you should limit the nesting level of if-statements inside loops you hope will software pipeline.

CHAPTER

SEVEN

SUPPORT

Post compiler related questions to the TI E2E™ website at the following link: TI E2E design community forum and select the TI device being used.

CHAPTER

EIGHT

REVISION HISTORY

Changes from December 15, 2023 to present

- Extensive reorganization and additions.

Changes from January 21, 2022 to December 15, 2023 (from Revision B (January 2022) to Revision C (December 2023))

- Added C7504 and C7524 sizes to vector width and vector register descriptions. See *C7000 Split Datapath and Functional Units*.
- Changed "trip count" to "iteration count" throughout to match new software pipelined loop information. See *MUST_ITERATE and PROB_ITERATE Pragmas and Attributes*.
- Updated and extended information about vectorization and vector predication. See *Vectorization and Vector Predication*.
- Added section about automatic use of Streaming Engine and Streaming Address Generator. See *Automatic Use of the Streaming Engine and Streaming Address Generator*.

Changes from March 15, 2021 to January 21, 2022 (from Revision A (March 2021) to Revision B (January 2022))

- Updated split datapath and functional unit diagram and description. See *C7000 Split Datapath and Functional Units*.

Changes from May 1, 2020 to March 15, 2021 (from Revision -- (May 2020) to Revision A (March 2021))

- Vector predicated stores generated by the compiler may trigger page fault exceptions in certain situations. This issue can be corrected in the linker command file. See *Vectorization and Vector Predication*.

CHAPTER

NINE

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (https://www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

CHAPTER

TEN

ABOUT THIS DOCUMENT

For offline use, a PDF version of the guide is available at: TI C7000 C/C++ Optimization Guide

CHAPTER

ELEVEN

RELATED DOCUMENTS

Use the following documents from Texas Instruments to supplement this user's guide:

- SPRUIG8 C7000 Optimizing C/C++ Compiler User's Guide
- SPRUIG4 C7000 Embedded Application Binary Interface (EABI) User's Guide
- SPRU425 C6000™ Optimizing C Compiler Tutorial
- SPRA666 Hand-Tuning Loops and Control Code on the TMS320C6000™
- SPRABK5 Throughput Performance Guide for KeyStone™ II Devices
- SPRUIG5 C6000-to-C7000 Migration User's Guide
- SPRUIG6 C7000 Host Emulation User's Guide
- SPRUIG3 VCOP Kernel-C to C7000 Migration Tool User's Guide

The following documents are available only through your TI Field Application Engineer:

- SPRUIU4 C7x Instruction Guide
- SPRUIP0 C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator Technical Reference Manual
- SPRUIQ3 C71x DSP Corepac Technical Reference Manual

CHAPTER TWELVE

RELATED COLLATERAL

A series of videos summarizing the C7000 compiler and its use is available:

- C7000 Compiler Video Series

CHAPTER

THIRTEEN

LEGAL INFORMATION

C7000™, C6000™, TMS320C6000™, and KeyStone™ are trademarks of Texas Instruments.

All other trademarks are the property of their respective owners.

For more information, see *IMPORTANT NOTICE AND DISCLAIMER*.