《C7000 C/C++ 优化指南》

User's Guide

Literature Number: ZHCU881D
MAY 2020 – REVISED MAY 2024

目录

插图清单

关于本手册

备注

本文档已被基于网络的 *TI C7000 C/C++ 优化指南* 所取代，该指南提供了此文档版本中未包含的其他信息。此外，该指南的 PDF 版本也可供使用。

本文档适用于那些期望提高 C7000™ CPU 上运行的代码性能的用户。

本指南并不旨在帮助优化存储器/高速缓存层次结构、MSMC、DMA 或矩阵乘法加速器 (MMA) 的代码。

本文档的读者应具备以下能力：

- 掌握 C 语言和 C++ 知识。
- 使用编译器选项调用 C7000 编译器的经验。
- 掌握基本的汇编语言概念。
- 了解 CPU 架构特性，例如寄存器、高速缓存和功能单元。

相关文档

作为本用户指南的补充，使用德州仪器 (TI) 提供的下述文档：

SPRUIG8 *C7000 优化 C/C++ 编译器用户指南*

SPRUIG4 *C7000 嵌入式应用二进制接口 (EABI) 用户指南*

SPRUIU4 *C7x 指令指南*（可通过 TI Field 应用工程师获得）

SPRUIP0 *C71x DSP CPU、指令集和矩阵乘法加速器技术参考手册*（可通过 TI Field 应用工程师获得）

SPRUIQ3 *C71x DSP Corepac 技术参考手册*（可通过 TI Field 应用工程师获得）

SPRU425 *C6000™ 优化 C 编译器教程*

SPRA666 *手动调优 TMS320C6000™ 上的循环和控制代码*

SPRABK5 *KeyStone™ II 器件的吞吐量性能指南*

SPRUIG5 *C6000 至 C7000 迁移用户指南*

商标

引言

备注

本文档已被基于网络的 *TI C7000 C/C++ 优化指南* 所取代，该指南提供了此文档版本中未包含的其他信息。此外，该指南的 PDF 版本也可供使用。

在介绍可用来使代码更高效的编译器选项和源代码策略之前，有必要了解一些 C7000 数字信号处理器和指令集有关的一些信息。本章概要介绍了 C7000 架构、数据路径和功能单元。

1.1 C7000 数字信号处理器 CPU 架构概述

C7000 CPU DSP 架构是德州仪器 (TI) 最新的高性能数字信号处理器 (DSP)。其配备在某些德州仪器 (TI) Keystone 3 器件中。这种超长指令字 (VLIW) DSP 因其宽矢量指令和多个功能单元而拥有显著的数学处理能力。本优化指南可以帮助开发人员实现 C7000 DSP 的最高性能。

当集成到更大的 TI 器件（例如一些 Keystone 3 器件）时，C7000 往往与矩阵乘法加速器 (MMA) 搭配使用，这样，可显著提高某些机器学习网络的性能。我们建议使用 TI 深度学习库，已对其进行经优以便使用矩阵乘法加速器。TI 深度学习库是 Processor（处理器） SDK 的一部分。

C7000 DSP 拥有矢量 (SIMD) 指令，根据数据类型和 C7000 CPU 版本，该指令能够在单条指令中执行多达 64 个操作。C7000 DSP 内核上几乎所有的计算指令都是完全流水线作业，这意味着在每个时钟周期均可启动独立指令。矢量指令与流水线行为的组合允许您在每个周期中进行大量的计算。C7000 DSP 内核具备定点和浮点矢量指令。

每个 C7000 DSP 内核都有若干功能单元。在每个时钟周期中，每个功能单元可以执行一条独立指令。在本指南中，重点介绍第一代 C7000 DSP 内核（C7100 和 C7120）。由于 C7100 和 C7120 DSP 内核有 13 个功能单元，因此每个时钟周期可执行 13 条指令。在现实中，一些功能单元专门用于执行某些类型的指令，因此出于这样或那样的原因，通常在每个周期中并非所有的 13 个功能单元都在执行指令。

有关 C7000 指令集的更多信息，请参阅 *C71x DSP CPU、指令集和矩阵乘法加速器技术参考手册* (SPRUIP0)。

1.2 C7000 分离式数据路径和功能单元

以下方框图显示 C7100 DSP CPU 上的分离式数据路径。图中有 A 端数据路径和 B 端数据路径。图中显示了功能单元和多个异构寄存器文件。A 端数据路径负责标量计算，从存储器加载和在存储器中存储标量与矢量，以及控制流（分支、调用）。B 端数据路径处理矢量数学运算、数据排列和矢量谓词运算。

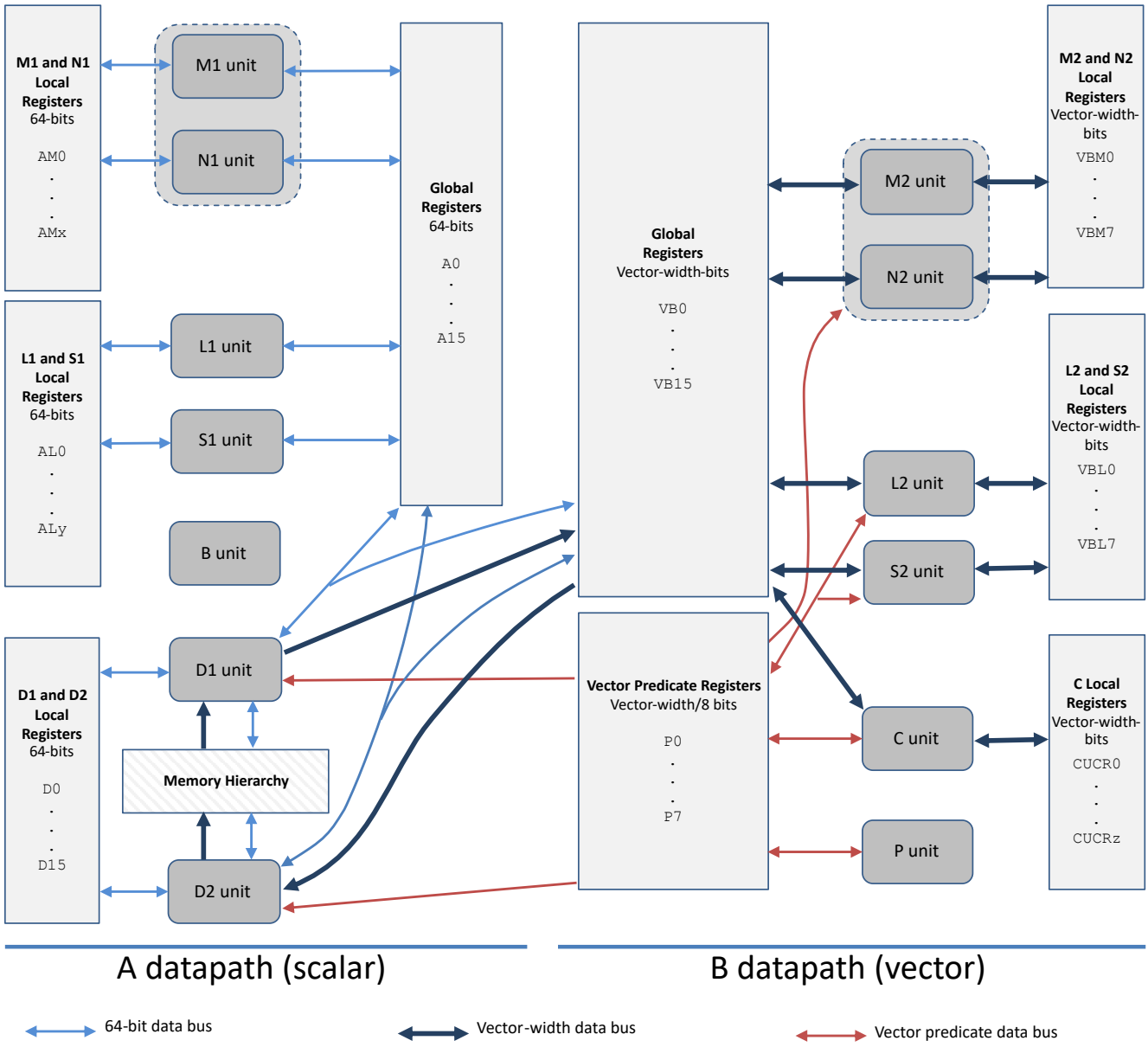

图 1-1. C7000 数据路径方框图

为了简化上图，此图中未显示某些数据移动功能和数据路径。

- 通常，功能单元可以写入同一数据路径上的任何寄存器文件。
- 大多数功能单元可以从一个或两个流引擎中获取数据。
- 每个数据路径 (A/B) 各有一个 64 位交叉路径。每个交叉路径允许每个周期从对侧全局寄存器文件读取一次。

C7100 和 C7120 器件具有 512 位向量宽度。C7504 和 C7524 器件具有 256 位向量宽度。寄存器每个寄存器有 64 位（“标量”）或每个寄存器有“向量宽度”位数。因此，C7100 和 C7120 器件具有 512 位向量寄存器，而 C7504 和 C7524 器件具有 256 位向量寄存器。

在给定的数据路径上，有若干不同类型的寄存器文件。在给定的数据路径上，每个功能单元都可以写入该数据路径上的全局寄存器文件以及该数据路径上的大部分“本地”寄存器文件。然而，只有一些功能单元可以从“本地”寄存器文件中读取。

- **D1 和 D2 单元：**这些单元位于 A 端数据路径上，可从存储器加载并存储到存储器中。两个 64 位负载可并行执行。两个 64 位存储可并行执行。64 位和向量宽度加载可以与 64 位或向量宽度存储并行执行。两个向量宽度存储不能并行执行，两个向量宽度负载也不能并行执行。
- **L1、S1、M1 和 N1 单元：**这些单元是通用功能单元，处理标量和小矢量计算的不同组合。M1 和 N1 功能单元执行各种乘法指令。
- **L2、S2、M2 和 N2 单元：**它们也是通用功能单元，并且可以运行全宽向量数据。M2 和 N2 功能单元执行各种乘法指令。
- **B 单元：**此单元处理间接的分支和调用。
- **C 单元：**此单元对数据进行排列和打乱。
- **P 单元：**此单元计算用于屏蔽矢量通道的谓词，如此，不用计算特定的通道，也无需将其存储到存储器中。

除了提供对存储器层次结构进行 CPU 访问的 D1 和 D2 单元之外，C7100 DSP 还有两个“流引擎”，便于快速从存储器获取数据。*流引擎*是一种硬件特性，允许您（或编译器）指定存储器地址模式，以便从存储器中获取数据。流引擎尽力将数据从存储器层次结构预取到靠近 CPU 的暂存存储器，以最大限度地减少由于冷缓存未命中而导致的 CPU 停顿。

章节 2

C7000 C/C++ 编译器选项

本章介绍了德州仪器 (TI) C7000 C/C++ 编译器和可用于优化性能的选项。

2.1 概述

德州仪器 (TI) C7000 编译器接受 C 或 C++ 源代码输入。编译时，编译器会经历几个阶段，如下图所示

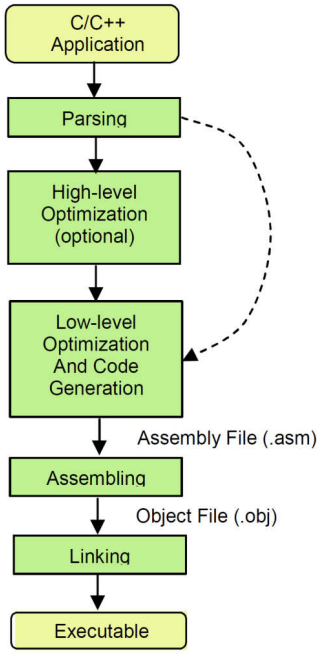

图 2-1. C7000 编译器处理阶段

首先，解析源文件，以创建高级中间表示，其与源语言非常相似，但更适合优化转换。

按某种优化级别编译的文件和函数（可选）通过高级优化器，该优化器执行函数内联、循环转换和其他代码优化。

然后，高级中间语言被翻译成与汇编非常相似的低级中间语言。低级优化器和代码生成传递执行分区、寄存器分配、软件流水线、指令调度和其他优化。

代码生成传递的输出是汇编文件，它由汇编器汇编成目标文件，然后由链接器链接到一个库或可执行文件中。

2.2 为性能选择编译器选项

在应用经过全面调试并正常工作之后，就该开始优化过程了。首先，您需要选择适当的编译器选项。以下编译器选项会影响性能。有关命令行选项的更多详细信息，请参阅 *C7000 C/C++ 编译器用户指南* (SPRUIG8)。

- **--opt_level=3 (-o3)。**编译器在 -opt_level=2 下执行函数级优化，并在 -opt_level=3 下执行文件级优化和函数内联。在 --opt_level=2 和 --opt_level=3 下，编译器执行各种循环优化，例如软件流水线、矢量化和循环合并。默认情况下，--opt_level 开关会对性能进行优化。此类优化可增加代码大小。如果代码大小构成问题，请勿降低优化级别。相反，使用 --opt_for_speed (-mf) 开关更改优化目标（性能与代码大小之间的对比关系）并使用 –oi 选项控制自动内联的数量。
- **--opt_level=4 (-o4)。**考虑使用此选项在链接时对所有文件执行优化。使用此选项可显著增加编译时间。如果在任何步骤中使用此选项，则必须在所有编译和链接步骤中使用该选项。只要源文件是使用 --opt_level=4 进行编译的，则都可以单独编译。此优化级别无法与 --program_level_compile (-pm) 一起使用。
- **--gen_func_subsections (-mo)。**如果源代码使用许多从未调用的函数，请考虑使用此选项。此选项将每个函数放在自身的输入子段中，因此，如果该函数从未被引用，则链接器可以从可执行文件中排除该函数。但是，这种优化可增加代码大小，因为编译器必须应用最小的段对齐要求。
- **--opt_for_speed=0 (-mf0) 或 --opt_for_speed=1 (-mf1)。**如果代码大小构成问题，那么在编译包含不常执行的函数或对性能微不足道的函数的文件时，请使用这些选项。这会告诉编译器优化代码大小，而非性能。请勿为了减小代码大小而降低优化级别 (--opt_level)。

如果您关心性能，*请勿使用* --disable_software_pipelining (-mu) 选项。此选项会关闭软件流水线。软件流水线对于在大多数循环上实现高性能至关重要。此选项可以作为调试工具，因为其使汇编代码更容易理解。

以下选项提供了用于调试和性能评估的附加信息：

- **--src_interlist (-s)。**此选项使编译器在编译器生成的汇编文件中发送经过高级优化的源代码副本。此输出作为汇编代码中的注释置于汇编文件中。优化器的注释输出看起来像 C 代码，并显示了已应用的高级转换，例如内联、循环合并和矢量化。此选项有助于您理解汇编代码以及编译器为优化代码性能所做的一些工作。此选项会开启 --keep_asm (-k) 选项，因此编译器生成的汇编文件 (.asm) 将不会被删除。

  --debug_software_pipeline (-mw)。此选项提供有关软件流水线循环的额外信息，包括循环的单次调度迭代。本文档后续介绍的循环调优示例中将使用此信息。此选项会开启 --keep_asm (-k) 选项，因此编译器生成的汇编文件 (.asm) 将不会被删除。

  --gen_opt_info=2 (-on2)。此选项创建 .nfo 文件，其基本名称与 .obj 文件相同。此文件包含有关已应用的高级优化的摘要信息以及提供建议。

2.3 了解编译器优化

在解释汇编代码以及其内的软件流水线信息之前，了解一些编译器在将 C/C++ 源代码编译成汇编代码是试图对其执行的一些操作会有所帮助。

2.3.1 软件流水线

像 C7000 这样的超长指令字 (VLIW) 数字信号处理器 (DSP) 依靠循环的软件流水线实现性能最大化。*软件流水线*是一种方法，其中源循环的连续迭代是重叠的，以便在整个循环中尽可能多的周期里利用 CPU 上的功能单元。

下图显示有软件流水线和无软件流水线的情况下循环迭代的执行。您可以看到，在没有软件流水线的情况下，循环被调试，因此循环迭代 *i* 完成之后迭代 *i+1* 才开始。在有软件流水线的情况下，迭代会重叠。因此，只要能够保持正确，即可在迭代 *i* 完成之前开始迭代 *i+1*。与其他方法相比，这种方法通常使机器资源的利用率更高。在软件流水线循环中，即使单个迭代可能需要 *s* 个周期才能完成，但每隔 *ii* 个周期就会启动一个新迭代。

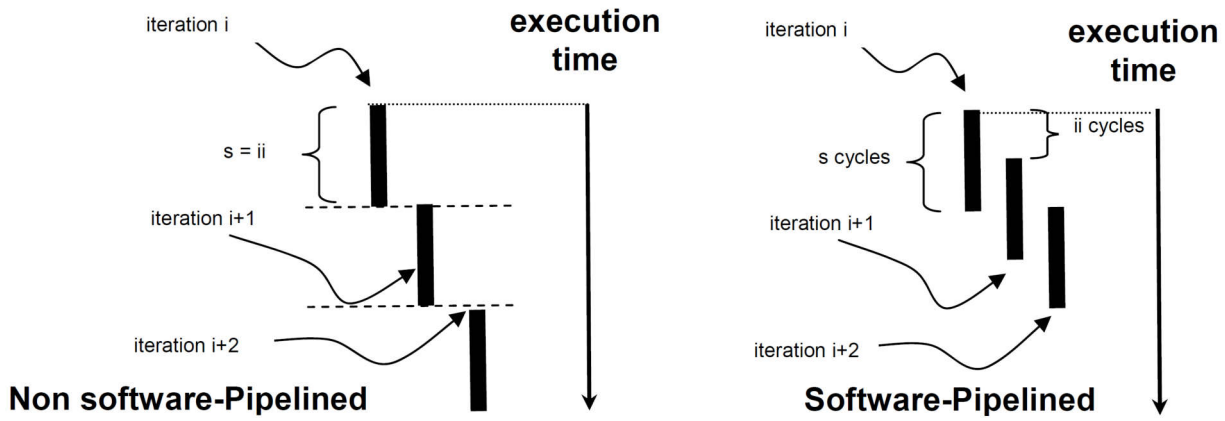

图 2-2. 软件流水线对执行的影响

在高效的软件流水线循环中，*ii* 远小于 *s*。*ii* 称为启动间隔，其是启动迭代 *i* 与启动迭代 *i+1* 之间的周期数。*s* 是完成第一个迭代所需的周期数，或者等于软件流水线循环的单次调度迭代的长度。

编译器尝试对最里面的源循环进行软件流水线处理。这些循环里面不再有任何其他循环。请注意，在编译过程中，软件流水线发生在内联之后和可将循环合并的循环转换之后，因此在某些情况下，您可能会看到编译器执行软件流水线的代码比预期的更多。

经过软件流水线处理后，循环有三个主要阶段，如下图所示：

- **pipe-up (prolog)** 阶段，在此期间启动重叠的迭代。
- **steady-state (kernel)** 阶段，在此期间继续启动迭代。
- **pipe-down (epilog)** 阶段，在此期间允许完成任何尚未完成的迭代。

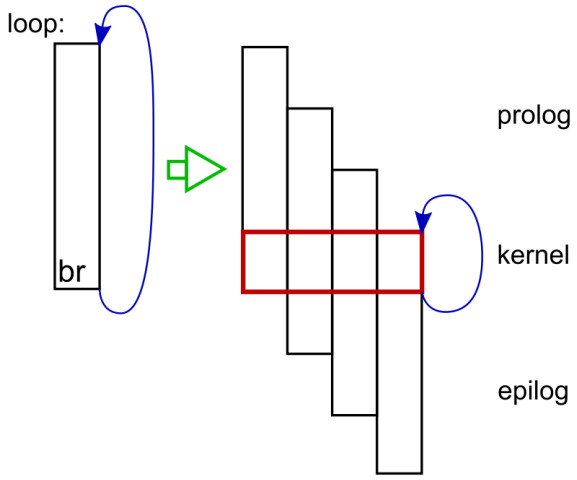

图 2-3. 有 prolog 和 epilog 的循环迭代

以下示例显示了简单加权矢量和的源代码。

```
// 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,
                  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;
    }
}
```

为了简化第一个软件流水线示例，使用了两个 pragma：

- UNROLL pragma 告知编译器勿执行矢量化，这是一种转换方法，将在下一部分中说明。
- MUST_ITERATE pragma 传递了有关循环执行次数的信息，这会在本文档后面说明。本例使用此 pragma 防止生成“重复循环”，这也会在本文档后面说明。

然后使用以下命令编译此代码：

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

`--symdebug:none` 选项阻止编译器在汇编中生成调试信息和关联的调试指令。此调试信息与本文档的论述无关，如果包含该信息，则会不必要地延长此处所示的示例。一般不用关闭调试生成，因为调试信息的生成不会降低性能。

由于使用了 --src_interlist 选项，所以编译器生成的汇编文件未被删除，并包含以下内容：

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file               : weighted_vector_sum.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(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      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>


           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
```

```
           ADDW    .L2     BL1,BL0,B0        ; [B_L2] |12| <0,11>
||         MPYWW   .N2     BM2,BM0,BL0       ; [B_N] |12| <2,7>
||         MPYWW   .M2     BM3,BM1,BL1       ; [B_M2] |12| <2,7>
||         SLDW    .D1     *D1++(4),BM0      ; [A_D1] |12| <5,1>
||         SLDW    .D2     *D2++(4),BM1      ; [A_D2] |12| <5,1>            BNL     .B1     ||
$C$L2||         ; [A_B] |10| <0,12>
||         STW     .D1X    B0,*D0++(4)       ; [A_D1] |12| <0,12>
||         TICK                               ; [A_U] <6,0>
;** --------------------------------------------------------------------------*
||$C$L3||:    ; PIPED LOOP EPILOG
;          EXCLUSIVE CPU CYCLES: 7
;**     ----------------------    return;

           ADDD    .D2     SP,0x8,SP         ; [A_D2] (O)
||         LDD     .D1     *SP(16),A9        ; [A_D1] (O)
||         ADDW    .L2     BL1,BL0,B0        ; [B_L2] |12| (E) <4,11>
||         MPYWW   .N2     BM2,BM0,BL1       ; [B_N] |12| (E) <6,7>
||         MPYWW   .M2     BM3,BM1,BL0       ; [B_M2] |12| (E) <6,7>

           STW     .D1X    B0,*D0++(4)       ; [A_D1] |12| (E) <4,12>
           ADDW    .L2     BL1,BL0,B0        ; [B_L2] |12| (E) <5,11>
           STW     .D1X    B0,*D0++(4)       ; [A_D1] |12| (E) <5,12>
           ADDW    .L2     BL0,BL1,B0        ; [B_L2] |12| (E) <6,11>
           STW     .D1X    B0,*D0++(4)       ; [A_D1] |12| (E) <6,12>

           RET     .B1     ; [A_B] (O)
||         PROT                               ; [A_U] (E)

           ; RETURN OCCURS {RP}              ; [] (O)
```

此汇编输出显示编译器生成的汇编文件中的软件流水线循环以及部分软件流水线信息注释块，其中包括关于各种循环特征的重要信息。

如果编译器成功地对循环进行软件流水线处理，那么编译器生成的汇编代码将包含软件流水线信息注释块，其中包含有关“ii = xx Schedule found with yy iterations in parallel”的消息。*启动间隔* (ii) 用于度量软件流水线循环开始执行循环新迭代的频率。启动间隔越小，执行整个循环所需的周期就越少。软件流水线循环信息还包括循环开始的源行、对循环资源和延迟要求的描述以及循环是否已展开（还有其他信息）。使用 `-mw` 编译时，该信息还包含单次调度迭代副本。

在本例中，实现的启动间隔 (ii) 是 2 个周期，并行运行的迭代次数为 7。

注释块还包括软件流水线循环的*单次调度迭代*视图。从软件流水线循环的单次调度迭代视图中，可看到编译器如何转换代码以及编译器如何调度软件流水线循环的一次迭代与软件流水线中的迭代重叠。有关如何解释此注释块中的信息，请参阅节 4.2。

2.3.2 矢量化和矢量谓词

C7000 指令集具有许多功能强大的单指令多数据 (SIMD) 指令，可在一条指令中执行多个操作。为了利用这种优势，编译器会在可能而有益的情况下尝试对源代码进行*矢量化*。矢量化通常涉及使用矢量 (SIMD) 指令一次对数据的若干循环迭代执行操作。

以下示例从上一部分的示例中删除了 UNROLL pragma 和 MUST_ITERATE pragma。UNROLL(1) pragma 阻止了 C7000 编译器中的某些循环转换优化。

```
// 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,
                  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;
    }
}
```

下面显示生成的已被矢量化的内部编译器代码。 编译器的矢量化可通过“+= 16”地址增量和优化器临时变量名称中的“32x16”（表示临时变量中有 16 个 32 位元素）来推断。

```
;***     ----------------------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;
```

生成的汇编文件中获得的软件流水线信息块如下所示：

```
;*   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
;*      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|
```

本例比较了上一部分中的示例输出，以显示矢量化的效果：

- “优化器”代码经过若干高级优化步骤，包括矢量化。（使用 -os 编译器选项时，此“优化器”代码显示在汇编文件中。）地址增量为 16，优化器临时变量的部分名称为“32x16”，表示 16 个 32 位元素。
- 汇编文件中的“SOFTWARE PIPELINE INFORMATION”（软件流水线信息）注释块显示，循环已经按 16 倍展开。这可能表明也可能不表明矢量化已发生，但往往与矢量化有关。
- 软件流水线循环现在使用 VMPYWW 和 VADDW 指令。指令助记符中的“V”通常（但不总是）表示编译器已经将代码序列矢量化（使用矢量/SIMD 指令）。
- 负载和存储指令中更大的地址增量可以是矢量化已发生的另一条线索。

在此循环中，编译器不知道循环将执行多少次。所以在我们的示例中，如果循环迭代的次数不是所选矢量宽度中元素数量的倍数，编译器就不能在最后一次循环迭代中将整个矢量存储到存储器中。例如，如果原始（未矢量化的）循环将执行 40 次迭代，而编译器将循环矢量化 16 次，则最后一次优化迭代将计算 16 个元素，但其中只有 8 个应该存储到存储器中。

C7000 ISA 具有某些矢量谓词特性，其中矢量谓词会影响哪些通道应该执行矢量运算。在这种情况下，BITXPND 指令生成矢量谓词，将在感知矢量谓词的存储指令中使用。此矢量存储指令 (VSTP16W) 使用矢量谓词来防止将上次迭代中仅作为矢量化过程的结果计算且不会原始循环中计算或存储的那些元素存储到存储器中。编译器尝试在矢量化过程中自动执行矢量谓词。矢量谓词有助于避免生成剥离式循环迭代的需求，这种迭代可抑制循环嵌套优化。

备注

如果启用了 Corepac 存储器管理单元 (CMMU)，并且存储与非法存储器页重叠，则矢量谓词存储可能会导致页面错误。任何在运行时小于非法存储器页 63 字节的存储器范围都应该在链接器命令文件中缩减长度。有关更多信息，请参阅 *C7000 C/C++ 编译器用户指南* (SPRUIG8)。

如果使用 MUST_ITERATE pragma 向编译器提供有关循环迭代次数的信息，则可以避免进行向量预测。例如，如果已知上一个示例中的循环仅以 32 的倍数执行，并且最小迭代计数为 1024，则以下示例会改进生成的汇编代码：

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

在编译后，这个修改后的示例会生成以下软件流水线信息块：

```
;*   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|
;*   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|
```

由于添加了 MUST_ITERATE pragma，编译器知道绝不需要向量预测，并且不会执行向量预测。因此，编译器删除了 CMPEQW、ANDW、VSTP16W 以及与向量预测相关的其他指令。

2.3.3 自动使用流引擎和流地址生成器

如果 C7100 和 C7120 器件上使用了 --auto_stream=no_saving 选项，或 C7504 和更高版本的器件上使用了 --auto_stream=saving 选项，编译器可以自动使用流引擎 (SE) 和/或流地址生成器 (SA)。

如果节 2.3.2 中的 weighted_vector_sum_v3.cpp 示例是使用 --auto_stream=no_saving 选项编译的，则会生成以下软件流水线信息块。（本例中生成的汇编适用于 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
;*      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|
```

在本例中，编译器使用 SE0 和 SE1 来替换之前设置了更低 ii 下限 4 的负载。通过使用 SE 执行这些负载，可以实现 2 ii。要使用上述示例中的 SE，编译器必须配置和打开它们。配置和打开操作显示在由循环之前的 --src_interlist 选项添加的注释中：

```
;***    -----------------------    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);
;***    -----------------------    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_O(*__se_mem((packed void *)b), 1, S$3);
```

默认情况下，编译器仅在使用 SE 或 SA 看起来有利可图且合法时才使用它们。

就盈利能力而言，一个关键的考虑因素是使用 SE 或 SA 会带来处理开销；编译器未必知道此开销有利可图。在示例中，MUST_ITERATE pragma 指示最小迭代计数为 1024，这使编译器相信使用 SE 或 SA 可能有利可图，因此编译器执行转换。如果编译器未使用 SE 或 SA，而您希望编译器使用它们，则使用 MUST_ITERATE 或 PROB_ITERATE pragma 指示迭代次数会有所帮助。

出于合法性的考虑，不使用 SE 或 SA 的大多数原因与是否需要寻址模式可始终映射到 SE 或 SA 相关。这些原因包括但不限于：

- 超过无符号 32 位类型范围的迭代计数器 (ICNT) 值。例如，当 `i` 和 `icnt` 是 64 位类型时，`for (i = 0; i < icnt; i++)` 中会发生这种情况。
- 超出有符号 32 位类型范围的 DIM 值。例如，当 `dim` 是 64 位类型时，`data_in[i*dim]` 中会发生这种情况。
- 寻址中超出有符号 32 位类型范围的加法或乘法。例如，当 `i` 或 `dim` 是 64 位类型时，`data_in[i*dim]` 中会发生这种情况。
- 寻址超出 INT_MIN 到 INT_MAX 个元素的范围的寻址。例如，在 `int16_ptr[i]` 中，当 `int16_ptr` 为 `int16 *` 且 `i` 为 `int` 时，最大范围是 INT_MIN*16 个元素到 INT_MAX*16 个元素。

这些都是非典型情况，在实践中不太可能发生。要允许编译器忽略它们，请使用 --assume_addresses_ok_for_stream 选项。

如果使用 SE 或 SA 在实践中没有优势，您可以使用 FUNCTION_OPTIONS pragma 覆盖单个函数的 --auto_stream 和/或 --assume_addresses_ok_for_stream 选项。

如果代码在函数中显式使用 SE 或 SA，则编译器不会选择使用 SE 或 SA 进行优化。在这种情况下，编译器会假定代码使用该函数内的 SE 和 SA 处理优化的所有方面。

有关自动使用 SE 和 SA 以及相关编译器选项的详细信息，请参见 *C7000 C/C++编译器用户指南* (SPRUIG8)。

2.3.4 循环折叠和循环合并

如果*折叠*或*合并*嵌套循环合法而且可提高性能，那么编译器会尝试这么做。*嵌套循环*是两个循环的集合，其中一个循环位于另一个封闭循环的内部。折叠和合并都涉及到将嵌套循环转换为单个循环。当外循环中没有代码时，就会发生折叠。当外循环中有代码时，就会发生合并。

两个嵌套循环组合成一个循环之后，必须转换外循环主体中的代码，从而仅在必要时有条件地执行该代码。折叠和合并都有利于性能，因为在执行循环嵌套时只执行一次 pipe-up（加速）和 pipe-down（减速），若不执行循环合并/折叠，则每次执行外循环时都要执行内循环的 pipe-up（加速）和 pipe-down（减速）。

要执行循环折叠或循环合并，合成的循环必须能够进行软件流水线作业。这意味着循环嵌套不能包含函数调用。每个循环必须有一个带符号的计数迭代器，每次迭代固定的次数。也就是说，内循环的迭代次数不能取决于外循环迭代的情况。此外，外循环不能包含太多代码，否则转换无法提高性能。如果外循环具有存储器依赖性，则可能不会执行循环合并和循环折叠。

当发生循环折叠或循环合并时，软件流水线循环会在软件信息注释块的顶部附近指示起始循环源代码行（“`Loop source line`”）。当此源代码行号引用一个外循环时，这表示内循环已经完全展开，或者编译器已经执行循环合并或折叠。在循环合并的情况下，编译器使用特殊指令，例如 NLCINIT、TICK、GETP 和 BNL。这些硬件特性的说明（包括所谓的“NLC”）不在本文档的范围内。有关 NLC 的更多详细信息，可参阅 *C71x DSP CPU、指令集和矩阵乘法加速器技术参考手册* (SPRUIP0)。

2.3.5 自动内联

编译器有时采用头文件中定义的函数，并将代码放在调用位置。这样，允许在封闭式循环中形成软件流水线作业，从而提高性能。编译器也可以这样做来消除调用函数和从函数返回的成本。

在下例中，`add_and_saturate_to_255()` 函数对两个值求和，如果总和超过 255，则将总和限制为 255。此函数是从 `inlining.cpp` 中的函数调用的，该函数通过预处理器的 #include 指令包含 `inlining.h` 文件。

```
// 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)
```

```
{
#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;
```

在此例中，编译器将内联对 add_and_saturate_to_255() 的调用，因此可执行软件流水线作业。您可以通过查看生成的汇编文件底部确定是否执行了内联。此处，编译器放置入了注释，说明 add_and_saturate_to_255() 已内联。请注意，由于 C++ 名称已改编，故函数的标识符也已被修改。

```
;; Inlined function references:
;; [0] _z23add_and_saturate_to_255ii
```

在生成的汇编代码中也可以看到内联，因为循环中没有针对函数的调用指令。事实上，由于内联 (因而消除了对函数的调用) ，循环可以进行软件流水线。 如果在循环中调用了另一个函数，则不会发生软件流水线。 请注意，由于代码大小的问题，并非每个可内联的调用都会自动内联。有关内联的更多信息，请参阅 *C7000 优化编译器用户指南*。

```
;*----------------------------------------------------------------------------*
;*        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|
;*   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|
;*----------------------------------------------------------------------------*
```

2.3.6 if 转换

为了对循环形成软件流水线 (从而提高性能) ，循环中唯一可能出现的分支是返回到循环顶部的分支。if-then 和 if-then-else 语句的分支或其他控制流构造的分支将阻止软件流水线。

为了摆脱这种限制，编译器进行了 *if-conversion*。通过预测指令以便根据“if”语句中的测试有条件地执行指令，if-conversion 尝试删除 if-then 和 if-then-else 语句相关联的分支。只要 if-then 或 if-then-else 语句中没有太多嵌套级、太多条件项或太多指令，if-conversion 通常会成功。

下述示例演示了 if-conversion。为了对此 C++ 代码中的“for”循环形成软件流水线，必须进行 if-conversion。pragma 用于防止编译器矢量化和生成对本例不重要的附加代码。

```
// 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;
    }
}
```

编译后，软件流水线信息注释块中循环的单次调度迭代如下所示：

```
;*----------------------------------------------------------------------------*
;*        SINGLE SCHEDULED ITERATION
;*
;*        ||$C$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|
;*----------------------------------------------------------------------------
```

指令 `[!A0] ADDW.D2 A1,A2,D3` 代表 if 语句的“then”部分。指令 `[A0] MVK32.S1 0,D3` 代表 if 语句的“else”部分。CMPGEW 指令计算 if-condition 并将结果放入谓词寄存器，用于有条件地执行 ADDW 和 MVKU32 指令。

章节 3

基本代码优化

此部分讨论了基本的代码优化技术，该技术可应用于将在 C7000 DSP 内核上运行的 C/C++ 代码。

3.1 迭代计数器和限制的有符号类型

为了实现自动矢量化，循环的迭代计数器和迭代限制应该是有符号类型。也就是说，使用 `int` 而非 `unsigned int`。

C 语言标准定义了无符号算术溢出的行为，但没有定义有符号算术溢出的行为。

在无符号的情况下，溢出值将“环绕”。因此，编译器必须假定（在某些情况下）循环计数器可能会循环，所以无法对循环的行为做出某些必要的结论。

在有符号类型的情况下，编译器可以假定迭代计数器不会溢出，因为根据 C 语言标准，它具有未定义的行为。因此，编译器可以对循环的行为做出某些结论，并由此可以对循环进行矢量化。

3.2 浮点除法

浮点除法运算开销很大。通常，除法运算的结果是运行时支持调用预定义函数来实现浮点除法。这样的调用阻止了软件流水线。

如果您的代码除以编译时已知的 `constant`（常量），请考虑预先计算 `1/constant`（常量）值，然后将除法运算替换为乘以 `1/constant`（常量）。仅当 `1/constant`（常量）值可用 IEEE-754 float 或 double 精确表示时，编译器才会自动执行此优化。

3.3 循环携带依赖和 restrict（限制）关键字

为了最大限度地提高生成代码的效率，C7000 编译器并行调度尽可能多的指令，特别是在软件流水线中。为了并行调度指令，编译器必须确定指令之间的关系或依赖性。依赖性意味着一条指令必须发生在另一条指令前；例如，必须先从存储器加载变量，然后才能使用该变量。因为只有独立指令才能并行执行，所以依赖性会抑制并行性。

- 考虑到完成第一条指令所需的延迟，如果编译器无法确定两条指令是独立的，则假定存在依赖性，并按顺序调度这两条指令。
- 如果编译器可以确定两条指令是彼此独立的，则能够并行调度这两条指令。

3.3.1 循环携带依赖

在进行软件流水线的某些情况下，编译器无法重叠循环的连续迭代以获得最佳性能。当编译器不能重叠循环的连续迭代时，性能则会受到影响：启动间隔（前述所述 `ii`）将大于预期间隔，并且同时使用的功能单元也很少。

几乎所有的情况下，这都是由于*循环携带依赖*造成的。循环携带依赖在一定程度上阻止了迭代 `i+1` 的执行与迭代 `i` 重叠。*循环携带依赖限制* 是软件流水线循环启动间隔的下限（因此也限制了软件流水线循环速度）。由于循环中的一组指令集存在排序约束（依赖性）周期，所以会出现循环携带依赖限制。在循环中的所有这些周期长度中，最大的循环携带依赖周期就是循环携带依赖限制。即使有大量的可用功能单元并行执行若干迭代，也会发生这种情况。

如果循环携带依赖限制大于分区资源限制，则循环携带依赖之一会减慢循环，因为启动间隔始终至少是分区资源限制和循环携带依赖限制中的最大者。

为了减少或消除有问题的循环携带依赖，必须确定周期，然后找到一种方法以缩短或打破该周期。

以下示例显示了循环携带依赖有问题的循环。

```
void weighted_sum(int * a, int * b, int * 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;
    }
}
```

本例中编译器生成的汇编代码（如下所示）显示了在汇编代码的软件流水线信息部分中，循环携带依赖限制为 7 个周期。

```
;*----------------------------------------------------------------------------*
;*   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|  ^
;*   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|
;*----------------------------------------------------------------------------*
```

软件流水线循环的最终软件流水线启动间隔至少是循环携带依赖限制和分区资源限制中的较大者。*当循环携带依赖限制值大于分区资源限制值时，表示代码中存在可能需要解决的循环携带依赖限制问题。*也就是说，当循环携带依赖限制大于分区资源限制时，如果消除了循环携带依赖限制，软件流水线循环可能会运行得更快。因此，在本例中，由于分区资源限制是 2，循环携带依赖限制是 12，所以此代码中存在应被调查的问题。

为了确定该问题，我们需要查看循环携带依赖中涉及的指令。这些指令在编译器生成的汇编文件的注释块中，以脱字符“^”标记这些指令。请注意，加载和存储指令是以插入符号标记的。由此可知，编译器认为连续迭代之间*可能*存在循环携带依赖的关系。这可能是因为编译器无法证明存储指令正在写入与加载指令所加载的位置无关的存储器区域。在缺少有关指针、数组和地址访问模式的位置信息时，编译器必须假定连续迭代可从前一次迭代的存储位置加载。有关循环携带依赖及其识别方法的更多信息，请参阅*手动调优 TMS320C6000 上的循环和控制代码* (SPRA666)。

3.3.2 restrict（限制）关键字

为了纠正前面示例中由循环携带依赖导致的问题，我们需要告知编译器，这些数组在存储器中不重叠，因此一个迭代与下一个迭代之间不存在存储器依赖关系。

许多常见的数字信号处理循环包含一个或多个加载操作、一些计算以及一个存储操作。通常，加载从数组中读取，存储则将值存储到数组。如果编译器不知道数组是独立的（或不重叠），那么编译器必须是保守的，并假定在迭代 `i+1` 或 `i+2` 等的加载中可能需要存储迭代 `i`。因此，重要的是务必告知编译器负载和存储数组是否位于完全不同的存储器区域（也就是说，指向的对象/数组不重叠）。

为此，我们可以使用 `restrict`（限制）关键字来实现这一点。此关键字告知编译器，在整个变量范围内（用于访问数组的数组名称或指针名称），只能通过该数组名称或指针名称访问该对象或数组。

备注

对 restrict（限制）指针的这种描述并不完全准确；但在大多数用途中已经足够。如果您想要详细了解 restrict（限制）关键字，请参阅 C 标准或*揭秘 restrict（限制）关键字*。

使用 restrict（限制）关键字允许有效地告知编译器，存储到存储器的数据不会写入下一次迭代的加载所读取的位置。因此，当编译器执行软件流水线时，连续的迭代可以重叠，从而使生成的代码运行得更快。

此 C 函数示例使用了 restrict（限制）关键字。生成的软件流水线信息注释块将显示，使用 restrict（限制）关键字时，循环携带依赖限制为 0，而分区资源限制为 2。这会将大大缩短两个周期的启动间隔 (ii)。

```
void weighted_sum(int * restrict a, int *restrict b, int *restrict out,
                  int weight_a, int weight_b, int n)
```

尽管 restrict（限制）关键字不是 C++14 或 C++17 标准中的一部分，但德州仪器 (TI) C7000 C/C++ 编译器允许在 C 和 C++ 模式中使用 restrict（限制）关键字。

备注

如果不正确地使用了 restrict（限制）关键字，编译器通常会生成具有未定义行为的代码，意味着编译器生成的代码将产生不正确的结果。

3.3.3 运行时别名消歧

在某些限定的情况下，编译器可生成两个循环：一个循环假定两个指针没有别名，另一个循环假定两个指针有别名。编译器生成运行时检查，以确定两个指针是否有别名。此优化称为*运行时别名消歧*。其优点在于，假定指针无别名的循环通常可以按更小的启动间隔进行软件流水线，从而提高循环性能。

由于技术性太强，无法在此进行描述，故编译器无法总是执行运行时别名消歧。此外，当编译器使用运行时别名检查生成两个不同的循环时，会抑制嵌套循环合并之类的某些进一步优化操作，因此最好在合法的前提下使用 restrict（限制）关键字。

关于识别和消除循环携带依赖的进一步讨论和详细信息，请参阅以下参考资料：

- *TMS320C6000 编程器指南* (SPRU198K)，第 2.2.2 节“存储器依赖”
- *手动调优 TMS320C6000 上的循环和控制代码* (SPRA666)，第 4.1 节“使用 restrict（限制）限定符、MUST_ITERATE pragma 和 _nasserts()”

3.4 函数调用和内联

在某些情况下，函数调用会抑制优化。例如，如果编译器无法内联被调用的函数，则包含函数调用的循环无法形成软件流水线作业。为了实现软件流水线之类的优化，可能需要通过以下方式之一定义被调用的函数：

- 在调用所在的源文件中包含“inline”（内联）关键字
- 在随 #include 预处理器指令包含的 .h 文件中，使用关键字“static inline”（静态内联）调用函数

在优化级别 --opt_level=3 和 --opt_level=4 下，编译器执行了一定量的自动内联。

3.5 MUST_ITERATE 和 PROB_ITERATE Pragma 与属性

当编译器知道循环将执行多少次时，通常可以生成更快的代码。通过 MUST_ITERATE 和 PROB_ITERATE pragma 以及 TI_must_iterate 和 TI_prob_iterate C++ 属性添加此信息可帮助编译器：

- 确定对循环进行矢量化循环是否有益
- 确定执行某些循环优化和循环嵌套优化是否有益
- 确定是否需要冗余循环（参阅下述*冗余循环*）

在对循环进行矢量化之前，编译器试图确定这种改变是否会提高性能。如果编译器掌握关于循环迭代计数的信息，那么有助于编译器更好地预测矢量化的获利。同样，编译器还试图确定某些循环优化和循环嵌套优化是否有益，因此关于循环迭代计数的信息对编译器很有帮助。

备注

请勿在 MUST_ITERATE pragma 或 TI_must_iterate C++ 属性中提供错误的迭代计数信息。如果在此 pragma/属性中指定错误信息，则编译器可能创建产生意外和错误行为的代码。

冗余循环：在某些情况下，如果编译器不知道循环将执行多少次，那么编译器会生成两个不同版本的循环。软件流水线循环通常必须执行一定的最少迭代次数才能合法执行。如果循环迭代计数小于此*最小安全迭代计数*，那么编译器会生成运行时迭代计数检查，以及软件流水线版本的循环或*重复循环*的分支。也就是说，编译器生成了“常规”版本的循环（执行速度慢很多）。

最小安全迭代计数取决于并行调度的迭代次数，以及编译器能够执行*阶段折叠*优化的效率。有关更多 信息，请参阅节 4.2.6。

汇编文件注释块中的软件流水线信息指定循环的最小安全迭代计数（迭代计数），并说明编译器是否生成了重复循环。

因为编译器有时必须生成冗余循环和在两个循环之间选择所必需的控制代码，所以在已知循环的最小迭代计数时，使用 MUST_ITERATE pragma 告知编译器是有帮助的，因为冗余循环可能不是必需的。这样可以提高性能，特别是当循环封闭在外循环中而且编译器可以对外循环和内循环执行循环折叠或其他循环优化的时候。

下述示例显示了汇编注释块的软件流水线信息部分中的冗余循环生成信息。

```
;*      Redundant loop generated
;*      Collapsed epilog stages       : 5
;*      Prolog not removed
;*      Collapsed prolog stages       : 0
```

3.6 if 语句和嵌套的 if 语句

因为除了转到内核的分支之外，软件流水线循环可能没有任何控制流，所以任何调用或控制流（if 语句）都会阻止软件流水线。为了减轻循环内部的控制流对循环是否形成软件流水线的影响，编译器对一些 if 语句执行“if-conversion”，这将在“then”和“else”子句中的指令上新增适当的谓词。由于机器谓词寄存器的数量有限，再加上其他因素，您应该限制希望形成软件流水线的循环内部的 if 语句嵌套级别。

3.7 内在函数

如果编译器没有使用您期望使用的专用 C7000 指令，或者如果运算不容易在 C 语言中表达（例如饱和加法），那么可以在 C/C++ 代码中使用内在函数。可用的内在函数显示在编译器安装目录的 `include` 目录下的 .h 文件中。

3.8 矢量类型

如果编译器未对循环进行矢量化，请考虑使用矢量类型。有关更多信息，请参阅 *C7000 优化 C/C++ 编译器用户指南* (SPRUIG8)。

请注意，编译器可能会对循环中的部分操作进行矢量化，但不会对其他操作进行矢量化，从而导致循环低效。在这种情况下，最好使用矢量类型和内在函数对循环进行手动矢量化。

3.9 待使用和避免的 C++ 特性

某些 C++ 特性会造成运行时损失。在编译时完全处理了其他特性，因此不会造成运行时损失。本文档不会全面论述哪些 C++ 特性会造成运行时损失以及哪些不会；相关论述可参见互联网上和印刷版的若干资源。

某些特性确实会造成运行时损失，但在提供所需的抽象和/或安全级别方面非常有用，因此无论如何都应该考虑使用它们。一些较常用的特性的指导原则如下：

这些特性可能产生运行时开销。请考虑其优势是否值得所投入的成本：

- 尽管其开销基本上并不比 `malloc()` 高或低，但请调用 `new()`
- 使用标准模板库 (STL)，主要是由于对 `new()` 的隐藏调用
- 异常/异常处理
- 运行时类型信息 (RTTI)
- 多重继承
- 虚函数（尽管运行时成本通常很小）

请随意使用这些特性，因为这些特性运行时开销很小甚至无开销：

- 模板
- 运算符重载
- 函数重载
- 内联
- 精心设计的继承。特别是，如果在编译时已知对象类型，则调用衍生类的成员函数不会造成损失。

以下特性可提高性能，应该尽可能使用：

- 使用 `const`
- 使用 `constexpr`
- 按引用传递对象，而非按值传递对象
- 可在编译时与运行时计算的构造和表达式

3.10 流引擎

C7100 CPU 有两个*流引擎*。流引擎是 C7000 CPU 内核的一个特性，有助于将数据从存储器加载到 CPU。流引擎可以将数据从存储器预取到 CPU 附近的位置，从而显著提高存储器层次结构的性能。预取数据可以显著减少数据载入 CPU 所需的时间。它还可以减少 L1 数据缓存容量的未命中次数，因为通过流引擎访问数据时会绕过 L1 缓存。

流引擎支持多达六维地址访问模式。当性能瓶颈涉及到存储器读取时（如果 D 单元资源限制占主导或缓存未命中占主导），如果提前知道存储器中对象的访问模式，请考虑使用一个或两个流引擎。流引擎与手动矢量化的循环结合使用时效果最好。更多有关流引擎和代码示例的信息，请参阅 *C71x DSP CPU、指令集和矩阵乘法加速器技术参考手册* (SPRUIP0)、*C7000 优化 C/C++ 编译器用户指南* (SPRUIG8) 以及编译器安装目录中的 `include` 目录下的 `c7x_strm.h` 文件。

C7000 编译器尚未自动使用流引擎特性。

3.11 流地址生成器

使用*流地址生成器*有助于限制计算加载或存储指令所用地址所需的指令数量。这反过来可减少软件流水线循环的资源限制，所以可对循环的启动间隔产生积极的影响（从而提高循环的性能）。它还允许进行循环折叠或循环合并，从而有可能提高循环的性能。

C7100 内核上有四个流地址生成器。有关更多信息和代码示例，请参阅 *C71x DSP CPU、指令集和矩阵乘法加速器技术参考手册* (SPRUIP0)、*C7000 优化 C/C++ 编译器用户指南* (SPRUIG8) 以及编译器安装目录中的 `include` 目录下的 `c7x_strm.h` 文件。

C7000 编译器尚未自动使用流地址生成器特性。

3.12 优化库

如果可以，请使用为 C7000 内核优化的库，例如 TI 的深度学习库 (TIDL)。有关可用库的更多信息，请参阅 Jacinto 7 的 Processor（处理器） SDK 文档。

3.13 存储器优化

改善数据加载和存储的优化措施通常对应用的性能至关重要。对 Keystone 3 器件上有用的存储器优化的详细检查不在本文档讨论的范围之内。但是，下述列出了用于提高存储器系统吞吐量和减少存储器层次结构延迟的最常见的优化措施。

- **分块：**输入、输出和临时数组/对象通常太大，无法放入多核共享存储器控制器 (MSMC) 或 L2 存储器中。例如，在对整个 1000x1000 像素的图像上执行算法时，图像太大了以至于无法放入到大部分或所有配置的 L2 存储器中，而且算法可能会冲击缓存，导致性能低下。使数据尽可能靠近 CPU 可以提高存储器系统的性能，但是当图像太大而无法放入 L2 缓存时，我们该如何做到这一点呢？根据算法的不同，使用一种称为“分块”的技术可能很有用，其中算法修改为在给定时间只对一部分数据进行操作。一旦处理完该数据“块”，算法会移到下一块。此技术通常与此列表中的其他技术搭配使用。
- **直接存储器访问 (DMA)：**考虑使用器件的异步 DMA 功能将新数据移动到 MSMC 存储器或 L2 存储器中，并使用 DMA 将处理过的数据移出。这样，C7000 CPU 就可以自由地执行计算，同时 DMA 为下一帧、块或层准备数据。
- **乒乓缓冲器：**考虑使用乒乓存储缓冲器，以使 C7000 CPU 在处理一个缓冲器中的数据时，正在向/从另一个缓冲器进行 DNA 传输。当 C7000 CPU 处理完第一个缓冲器时，算法切换到第二个缓冲器，如此因 DMA 传输而有了新数据。考虑将这些缓冲器放在 MSMC 或 L2 存储器中，这比 DDR 存储器快得多。

章节 4

了解汇编注释块

本章说明了通过 `--debug_software_pipeline` 编译器选项添加到汇编输出中的软件流水线信息块，以及通过 `--src_interlist` 编译器选项添加的单次调度迭代块。

4.1 软件流水线处理阶段

对循环进行软件流水线处理时，C7000 编译器会经历三个基本阶段。这三个阶段是：

1. 认证循环是否可进行软件流水线
2. 收集循环资源和依赖图信息
3. 尝试对循环进行软件流水线处理

当编译器试图对内循环进行软件流水线处理时，编译器可能已对循环中的代码应用了某些转换，也可能合并了相邻或嵌套循环。

阶段 1：认证

在编译器允许软件流水线或认为软件流水线合法之前，必须满足几个条件。导致软件流水线失败的两个最常见条件是：

- 循环中不能有太多指令。循环太大，通常需要的寄存器超过可用数量，而且需要更长的编译时间。
- 除非调用的函数已内联，否则无法在循环中调用另一个函数。控制流的任何中断都会使软件流水线无法进行，因为多个迭代正在并行执行。

如果软件流水线的任何条件*无法*满足，流水线认证将会停止，并显示不合格消息。有关疑难解答，请参阅节 4.4.1，有关此阶段所提供的信息，请参阅节 4.2.1。

如果满足软件流水线的所有条件，编译器继续进入阶段 2。

阶段 2：收集循环和依赖信息

软件流水线的第二阶段涉及收集循环资源和依赖图信息。有关此阶段输出的信息，请参阅节 4.2.2。

阶段 3：软件流水线尝试

一旦编译器认证循环可进行软件流水线，对其进行分区，并分析必要的循环携带和资源要求后，便可以尝试执行软件流水线。

编译器尝试在特定*启动间隔* (`ii`) 下开始对循环进行软件流水线处理。每当在特定启动间隔下编译器的软件流水线尝试失败时，`ii` 便会增加，并进行另一次软件流水线尝试。在软件流水线信息注释块中可看到此信息。这个过程一直持续到软件流水线尝试成功或 `ii` 等于未进行软件流水线的调度循环的长度。如果 `ii` 达到未进行软件流水线的调度循环的长度，软件流水线尝试将会停止，编译器则生成非软件流水线循环。更多有关此阶段中提供的信息，请参阅节 4.2.3。

如果软件流水线尝试不成功，编译器会提供附加反馈来帮助说明原因。有关最常见的软件流水线故障列表和缓解策略，请参阅节 4.4.2。

在特定启动间隔下找到成功的软件流水线调度和寄存器分配后，便会显示关于循环的更多信息。请参阅节 4.2.4、节 4.2.5、节 4.2.6、节 4.2.7 和 节 4.2.8。

4.2 软件流水线信息注释块

下述小节描述了在软件流水线信息注释块中找到的一些信息，当使用 `--debug_software_pipeline` 编译器选项时，该注释块添加到生成的汇编源文件中。在这种情况下，自动使用 `--keep_asm` 选项保留汇编输出。

通过了解编译器对循环进行流水线处理时产生的反馈，您可以调优 C 代码以获得更好的性能。

4.2.1 循环和迭代计数信息

如果编译器使循环符合软件流水线作业，则前几行如下例所示：

```
;*----------------------------------------------------------------------------*
;*   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
```

循环计数器叫做“迭代计数器”，因为它是一个循环中的迭代次数。此注释块部分中提供的统计信息有：

- **在文件中找到的循环、循环源代码行、循环左大括号源代码行、循环右大括号源代码行**：有关循环在原始的 C/C++ 源代码中的位置的信息。
- **已知最小迭代计数**：鉴于编译器可用的信息量，循环可能执行的最小次数。
- **已知最大迭代计数**：鉴于编译器可用的信息量，循环可能执行的最大次数。
- **已知最大迭代计数因子**：平均划分给迭代计数的最大数。即使不确定迭代计数的确切值，但可以知道该值是 2、4 等数值的倍数，可支持更积极的打包数据/SIMD 优化。

编译器尝试识别关于循环计数器的信息，例如最小值（已知最小迭代计数），以及它是否是某个数的倍数（有已知最大迭代计数因子）。

如果已知最大迭代计数因子大于 1，编译器可能会更积极地处理打包数据和优化循环展开。例如，如果循环计数器的确切值未知，但已知该值是某个数字的倍数，那么编译器也许能更好地展开循环以提高性能。

4.2.2 依赖和资源限制

软件流水线的第二阶段涉及收集循环资源和依赖图信息。阶段 2 的结果显示在软件流水线信息注释块中，如下所示：

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

此部分提供的统计信息如下：

- **循环携带依赖限制**：最大循环携带路径的距离（如果存在）。当循环的一次迭代写入在未来迭代中必须读取的值时，便会出现循环携带路径。属于循环携带限制中的一部分的指令用 ^ 符号标记。针对循环携带依赖限制显示的数字是因循环的循环携带依赖限制导致的最小迭代间隔。

  如果循环携带依赖限制大于资源限制，可能导致循环效率低下，您可以通过向编译器传递附加信息来提高性能。节 3.3 讨论了这种情况的潜在解决方案。
- **未分区资源限制**：编译器将每条指令分区到 A 端或 B 端之前，最佳情况下资源限制的最小迭代间隔 (`mii`)。
- **分区资源限制**（调度前、调度后）：指令分区到 A 端或 B 端之后的 `mii`。给出了调度前和调度后的值。调度后的值为调度发生后的分区资源限制。调度有时涉及到添加指令，这可能会影响资源限制。

4.2.3 启动间隔 (ii) 和迭代

有关软件流水线尝试的信息如下所述：

- **启动间隔 (ii)：**在本例中，编译器能够构造一个软件流水线循环，该循环每 13 个周期开始一次新的迭代。启动间隔越小，执行循环所需的周期就越少。
- **并行迭代：**在稳态（内核）下，循环示例同时执行三个迭代的不同部分。这意味着在迭代 n 完成前，迭代 n+1 和 n+2 已开始。

```
;*      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
```

4.2.4 常量扩展

每个执行数据包可容纳多达两个常量扩展。

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

如果指令的操作数常量太大而无法容纳在指令内的编码空间中，则常量扩展槽可供执行数据包中的指令使用。对于具有常量操作数的指令，常量的编码空间通常只有几位。如果这几位不能容纳常量，编译器可使用常量扩展槽。

“Constant Extension #n Used”反馈显示了每个 C0 和 C1 槽使用的常量扩展槽的数量。

4.2.5 使用的资源和寄存器表

资源分区表总结了如何将指令分配给各种机器资源以及如何在 A 端和 B 端之间对指令进行划分。下面显示了一些示例。

星号 (*) 标记决定资源限制值（也就是最大 `mii`）的条目。因为许多 C7000 指令可以在多个功能单元上执行，所以该表按照可能的资源组合将功能单元分成几类。

- 单个**功能单元（.L、.S、.D、.M、.C 单元等）**显示专门需要该单元的指令总数。可在多个功能单元上操作的指令不包括在这些计数中。

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

- **分组功能单元（.M/.N、.L/.S、.L/.S/.C 等）**显示可在所有列出的功能单元上执行的指令总数。例如，如果 .L/.S 行显示 A 端值为 14，B 端值为 12，则意味着在 .L1 或 .S1 上将执行 14 条指令，在 .L2 或 .S2 上将执行 12 条指令。

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

- **.X 交叉路径**显示了将数据从一个数据路径移动到另一个数据路径（A 到 B 或 B 到 A）所需的交叉路径总线的数量。

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

- **限制：**当仅考虑可在该行上列出的一组功能单元上操作的指令时，显示循环可进行软件流水线处理的最小 `ii`。例如，如果 .L .S .LS 行显示 A 端值为 3，B 端值为 2，则意味着有足够的指令需要在 .L 和 .S 上运行，在软件流水线调度中需要为 .L1 和 .S1 安排三个周期，并为 .L2 和 .S2 安排两个周期。请注意，.L .S .LS 符号意味着我们考虑只能在 .L 单位上运行的指令、只能在 .S 单位上运行的指令或可以在 .L 或 .S 上运行的指令。

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

- **寄存器使用表**编译器显示了在软件流水线内核的每个周期中使用哪些 CPU 寄存器。您很难使用此信息来提高循环性能，但是可以由此了解整个循环中有多少寄存器处于活动中。

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

4.2.6 阶段折叠

在某些情况下，编译器可以通过一种称为*阶段折叠*的转换，减少软件流水线循环的最小安全迭代计数。有关阶段折叠的信息显示在软件流水线信息注释块中。示例如下所示。

阶段折叠始终有助于缩减代码大小。阶段折叠通常有利于提高性能，因为它可以减少软件流水线循环的最小安全迭代计数，这样当循环执行的次数很少时，软件流水线循环很可能执行（速度更快），而且不必将执行转移到重复循环（速度更慢且不是软件流水线）。

```
;*      Epilog not entirely removed
;*      Collapsed epilog stages       : 2
;*
;*      Prolog not removed
;*      Collapsed prolog stages       : 0
;*
;*      Max amt of load speculation   : 128 bytes
;*
;*      Minimum safe iteration count  : 3 (after unrolling)
```

上例中的反馈显示了两个结尾阶段已折叠。但是，编译器无法折叠任何开场阶段，所以无法将软件流水线循环的最小安全迭代计数减小到 1（这是最好的情况）。 软件流水线循环开场或结尾无法删除的技术原因很复杂，编程人员也很难影响这种结果。

执行阶段折叠时，编译器可能生成*推测式*执行加载指令的代码，意味着加载结果可能得不到使用。在编译器需要推测式执行加载指令的情况下，如果访问的地址超出了合法存储器的范围，则编译器只对不会导致异常的加载指令执行此操作。有关“Max amt of load speculation”的反馈告知您加载推测将访问的地址超出正常地址访问范围的程度。

4.2.7 存储器组冲突

编译器对缓存层次结构的存储器组结构和通过存储器访问的对象的对齐方式了解有限。尽管如此，编译器还是尝试评估由于存储器组冲突停顿给不良存储器对齐的性能造成的影响。编译器在软件流水线信息注释块中显示此信息：

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

4.2.8 循环持续时间公式

编译器还会给出公式，计算执行相关软件流水线循环所需的周期数。由于编译器经常并行调度 prolog 和/或 epilog 与循环周围的其他一些代码，因此该公式并不能精确计算整个函数的预计周期数。

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

4.3 单个调度迭代注释块

因为软件流水线循环的迭代有重叠，所以很难理解与循环相对应的汇编代码。如果源代码是使用 --debug_software_pipeline 选项编译的，则将单个调度迭代注释块添加到生成的汇编源文件中。检查此代码可以更容易地理解编译器所做的工作，反过来也使循环优化更容易。

```
;*----------------------------------------------------------------------------*
;*         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|
;*----------------------------------------------------------------------------*
```

4.4 识别流水线故障和性能问题

下述小节解释了可能阻止循环优化的情景。

4.4.1 阻止循环进行软件流水线作业的问题

以下情景可能会妨碍循环形成软件流水线。通过检查注释块中的汇编输出和软件流水线信息可发现这些问题。

- **循环包含函数调用：**虽然软件流水线循环可包含内在函数，但不能包含函数调用。这包括将导致调用不可内联的运行时支持例程的代码，例如浮点除法。您可以尝试内联用户定义的小函数；参阅节 2.3.5。
- **循环包含控制代码：**在某些情况下，编译器不能从 if-then-else 语句或“?:”语句中删除所有的控制流。您可以尝试通过仅在更新存储器的代码周围以及在循环内部计算且仅在循环外部使用的变量周围使用 if 语句来优化这种情况。
- **有条件递增的循环控制变量未经软件流水线处理。**如果循环包含有条件递增的循环控制变量，则编译器无法对循环进行软件流水线处理。

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

- **指令太多。**由于超大循环需要大量的寄存器，所以通常无法调度。然而，有些大循环需要过长的时间进行编译。潜在的解决方案是将大循环分解成多个更小的循环。
- **未初始化的迭代计数器。**循环计数器可能未设置为初始值。
- **无法识别迭代计数器。**循环控制过于复杂。尝试简化循环。

4.4.2 软件流水线故障消息

编译器可能提供的软件流水线故障消息包括下述内容：

- **地址增量过大。**在软件流水线期间，编译器允许对来在同一数组或指针的所有负载和存储进行重新排序。这样最大限度地提高了调度灵活性。一旦发现调度，编译器就会返回并将适当的偏移量和增量/减量添加到每个负载和存储中。有时在重新排序后，加载和/或存储最终会相互偏移太远（标准加载指针的限制是 +/−32）。如果发生这种情况，尝试重新构造循环，使指针靠得更近，或者重写指针以使用预先计算的寄存器偏移量。
- **无法分配机器寄存器。**在进行软件流水线作业并找到有效的调度之后，编译器将循环中的所有值分配给特定的机器寄存器。在某些情况下，编译器会耗尽机器寄存器，它可以在这些寄存器中分配变量和中间结果的值。如果发生这种情况，请尝试简化循环，或者将循环分解成多个更小的循环。在某些情况下，编译器可以在更高的启动间隔 (`ii`) 下成功对循环进行软件流水线处理。
- **循环计数太高。无益。**在极少数情况下，软件流水线循环的迭代间隔比非流水线循环的迭代间隔更高。在这种情况下，执行非软件流水线循环更有效率。一种可能的解决方案是将循环分成多个循环或降低循环的复杂性。
- **未找到调度。**有时，编译器在特定启动间隔下无法找到有效的软件流水线调度。一种可能的解决方案是将循环分成多个循环或降低循环的复杂性。
- **并行迭代 > 最大迭代计数。**并非所有循环都可以进行有益的流水线处理。根据可用的最大迭代计数信息，编译器估计，鉴于在当前启动间隔下找到的调度，执行非软件流水线版本总是比执行流水线版本更有利可图。一种可能的解决方案可能是完全展开循环。
- **并行迭代 > 最小迭代计数。**根据可用的最小迭代计数信息，执行流水线版本的循环并非总是安全的。正常情况下会生成冗余循环。然而，在这种情况下，冗余循环生成已通过 `--opt_for_speed=3` 或更低的选项抑制。一种可能的解决方案是添加 MUST_ITERATE pragma，向编译器提供有关循环的最小迭代计数的更多信息。
- **寄存器存活时间过长。**有时，编译器发现有效的软件流水线调度，但其中一个或多个值存活时间过长。寄存器的生命周期由值写入寄存器的时间与另一条指令读取该值的最后一个周期之间的周期时间决定。根据定义，变量的存活时间永远不会超过循环的 `ii`，因为循环的下一次迭代会在读取该值之前将其覆盖。在这条消息之后，编译器会详细描述哪些值存活时间过长：

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

在本例中，数字 72、73、74 和 75 对应行号，可映射到违规指令上。编译器积极尝试阻止并修复存活过长的问题。可用来解决存活过长问题的方法成功概率很低。因此，本文档不讨论这些方法。此外，编译器通常可以在更高的启动间隔 (`ii`) 下找到成功的软件流水线调度。

4.4.3 性能问题

通过检查汇编源代码和软件流水线信息注释块，可发现以下问题。对于每种情况，都给出了潜在的解决方案。

- **生成了两个循环，一个未经软件流水线处理/生成了重复的循环**。如果您在软件流水线信息注释块中看到“Duplicate Loop Generated”消息，或者您发现循环的第二个版本未经软件流水线处理，这可能意味着当循环的迭代计数（迭代计数）过低时，执行编译器创建的软件流水线版本的循环是非法的。为了只生成软件流水线版本的循环，编译器需要证明循环的最小迭代计数将足够高以始终安全执行流水线版本。如果已知循环的最小迭代次数，使用 MUST_ITERATE pragma 告诉编译器此信息可能有助于消除重复的循环。
- **循环携带依赖限制大于分区资源限制**。如果您看到循环携带依赖限制高于分区资源限制，那么您可能会遇到下述两个问题之一。第一，编译器可能认为从存储到后续的加载存在存储器依赖关系。更多信息，请参阅 *TMS320C6000 编程器指南* (SPRU198) 的“存储器依赖”部分。第二，一次循环迭代中的计算可能用于下一次循环迭代。在这种情况下，唯一的选择是尽量消除从一个迭代到下一个迭代的信息流，从而使迭代更加独立。
- **嵌套循环中外循环开销大**。如果嵌套循环的内循环计数相对较小，则执行外循环的时间可能会在总执行时间中占很大比例。对于似乎会降低整个循环嵌套的性能的情况，可以尝试两种方法。第一，如果外循环没有太多的指令，可能需要提示编译器，使其应该合并循环嵌套。尝试使用 COALESCE_LOOP pragma 并检查整个循环嵌套的相对性能。如果 COALESCE_LOOP pragma 不起作用，并且内循环的迭代次数很小且没有变化，那么手工完全展开内循环可能会提高嵌套循环的性能，因为外循环可能能够进行软件流水线作业。
- **存在存储器组冲突**。如果编译器在一个周期内生成两个存储器访问，并且这些访问驻留在缓存层次结构中的同一个存储块中，则可能发生存储器组停顿。为了避免这种退化，可以通过使用 DATA_ALIGN pragma 将两个访问放在不同的存储块中，以避免存储器组冲突。

有关 pragma 的信息，请参阅 *C7000 优化 C/C++ 编译器用户指南* (SPRUIG8)。

注：以前版本的页码可能与当前版本的页码不同

Changes from DECEMBER 15, 2023 to MAY 20, 2024 (from Revision C (December 2023) to Revision D (May 2024)) **Page**

- 此文档版本已弃用。可通过提供的链接获取本文档的修订和扩展版本。.....6

Changes from JANUARY 21, 2022 to DECEMBER 15, 2023 (from Revision B (January 2022) to Revision C (December 2023)) **Page**

- 在向量宽度和向量寄存器描述中添加了 C7504 和 C7524 大小。.....8
- 通篇将“循环计数”更改为“迭代计数”，以匹配新软件流水线循环信息。.....13
- 更新和扩展了有关矢量化和向量谓词的信息。.....16
- 添加了有关自动使用流引擎和流地址生成器的部分.....20

Changes from MARCH 15, 2021 to JANUARY 21, 2022 (from Revision A (March 2021) to Revision B (January 2022)) **Page**

- 更新了分离式数据路径和功能单元的图表与说明.....8

Changes from MAY 1, 2020 to MARCH 15, 2021 (from Revision -- (May 2020) to Revision A (March 2021)) **Page**

- 由编译器生成的矢量谓词存储在某些情况下可能会触发页面错误异常。可在链接器命令文件中更正此问题。.16

This page intentionally left blank.

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265