“神威•太湖之光”计算机

编译系统用户手册

Version 0.9

国家并行计算机工程技术研究中心

2016 年 6 月

目 录

第一章 概述 1

第二章 神威编译器快速入门 2

2.1 安装说明 2
2.2 使用说明 2
2.3 其他编译器 4
2.3.1 基于 GCC 的编译器 4
2.3.2 老版编译器 4
2.4 常用工具 4
2.4.1 查看编译选项工具 4
2.4.2 编译器版本号 5
2.5 输入文件类型 5
2.6 其他输入文件 6
2.7 一般编译选项 6
2.8 共享库 7
2.9 内存模式支持 7
2.10 调试 8
2.11 反馈 8

第三章 FORTRAN 编译器 9

3.1 使用 FORTRAN 编译器 9
3.2 固定源和自由源格式文件 10
3.3 模块 10
3.3.1 出现的顺序 10
3.3.2 与其它目标程序进行链接 11
3.3.3 模块相关的报错信息 11
3.4 扩展 12
3.4.1 对不同空间属性的支持 12
3.4.2 SIMD 扩展 14
3.4.3 普通 instrinsic 15
3.4.4 REAL 和 INTEGER 类型的升级 15
3.4.5 Cray 指针 15
3.4.6 编译指示 15
3.5 编译器和运行时库的特点 17
3.5.1 使用-cpp 进行预处理 17
3.5.2 使用-ftpp 进行预处理 17
3.5.3 支持可变长的字符串 17
3.5.4 使用-fcoco 进行预处理 17
3.5.5 预定义的宏 18
3.5.6 Fortran 90 内部向量 19
3.5.7 边界检查 19
3.5.8 伪随机数 19

3.6 混合编程 19
3.6.1 C 和 Fortran 之间互相调用 20
3.6.2 调用举例 20
3.6.3 C 访问公用块举例 23
3.7 I/O 大小端的转换 24
3.7.1 assign 命令 25
3.7.2 使用通配符选项 25
3.7.3 转换数据和记录头 25
3.7.4 ASSIGN 子程序 25
3.7.5 I/O 编译选项 26
3.8 FORTRAN 的调试和查错 26
3.8.1 写常数区域错误 26
3.8.2 Fortran 哑元别名引起运行时错 27
3.8.3 Fortran malloc 的调试 27
3.8.4 参数拷贝到临时变量 27
3.8.2 保留的文件单元 29
3.9 源代码的兼容性 29
3.10 库的兼容性 30
3.10.1 换名 30
3.10.2 ABI 兼容性 31
3.11 移植 FORTRAN 代码 31
3.12 FORTRAN 程序栈大小 31
第四章 C/C++编译器 33
4.1 使用 C/C++编译器 33
4.1.1 兼容 GCC2.96 和 GCC4.2 的 C 和 C++前端 34
4.1.2 对不同空间属性的支持 34
4.1.3 DMA instrinsic 35
4.1.4 其他扩展 40
4.2 编译和运行时特性 40
4.2.1 预处理 40
4.2.2 编译指示 41
4.2.3 混合编程 42
4.2.4 链接 42
4.3 C/C++程序的调试 42
4.4 不支持的 GCC 扩展 42
第五章 加速线程库 44
5.1 运算控制核心加速线程库 44
5.1.1 初始化线程库 44
5.1.2 创建单线程 44
5.1.3 等待单线程终止 45
5.1.4 关闭单线程流水线 45
5.1.5 创建线程组 46
5.1.6 等待线程组终止 47

5.1.7 关闭线程组流水线 47
5.1.8 创建带屏蔽码线程组 48
5.1.9 等待带屏蔽码线程组终止 48
5.1.10 创建抢占动态调度运算控制核心线程 49
5.1.11 创建抢占动态调度运算核心线程组 49
5.1.12 获取核组最大线程总数 50
5.1.13 设置并行区线程总数 50
5.1.14 获取并行区线程总数 50
5.1.15 强制线程结束 51
5.1.16 中断管理 51
5.1.17 异常管理 52
5.1.18 运算控制核心取运算核心局存地址 54
5.1.19 运算控制核心访问运算核心局存 55
5.1.20 线程空闲状态查询 56
5.1.21 核组状态初始化 56
5.1.22 核组间同步 56
5.2 运算核心加速线程库 57
5.2.1 获得线程逻辑标识号 57
5.2.2 获得线程物理运算核心号 57
5.2.3 发送异步中断给运算控制核心 58
5.2.4 发送同步中断给运算控制核心 59
5.2.5 数据接收 GET 60
5.2.6 数据发送 PUT 61
5.2.7 物理地址数据接收 62
5.2.8 物理地址数据发送 62
5.2.9 DMA 栏栅 62
5.2.10 核组内同步 62
5.3 运算核心局存分配 63
5.3.1 get_allocatable_size 63
5.3.2 ldm_malloc 63
5.3.3 ldm_free 64
第六章 优化快速入门 65
6.1 基本优化 65
6.2 过程间分析优化（IPA） 65
6.3 反馈式编译优化（FDO） 65
6.4 更激进的优化 65
6.5 编译选项的推荐使用方法 66
6.6 性能分析 67
第七章 优化选项 68
7.1 常用优化选项：-O 选项 68
7.2 组合优化分析(-CG, -IPA, -LNO -OPT, -WOPT) 69
7.3 过程间分析 69
7.3.1 IPA 编译模式 69

7.3.2 过程间分析和优化 70
7.3.3 优化措施 70
7.3.4 IPA 控制 71
7.3.5 克隆 73
7.3.6 其他 IPA 调节选项 73
7.3.7 SPEC CPU2000 个案研究 74
7.3.8 调用 IPA 74
7.3.9 IPA 的大小与正确性的缺陷 75
7.4 循环嵌套优化（LNO） 75
7.4.1 循环分裂和融合 76
7.4.2 Cache 大小说明 76
7.4.3 Cache 分块，循环展开，循环交换，循环变换 76
7.4.4 数据预取 77
7.4.5 向量化 77
7.5 代码生成（CG） 77
7.6 激进的优化 78
7.6.1 别名分析 78
7.6.2 影响精度的优化 79
7.6.3 IEEE 754 的兼容性 79
7.6.4 其它不安全的优化 80
7.6.5 算术精度的假定 80
7.7 编译优化指示 81
7.7.1 分支频率指示 81
7.7.2 对界指示 82
7.7.3 循环展开指示 83
7.7.4 指针别名指示 83
第八章 使用 SIMD 扩展指令 85
8.1 SIMD 概述 85
8.1.1 有关 SIMD 的几个基本概念 85
8.1.2 编译器对 C 语言的 SIMD 扩展 86
8.1.3 SIMD 扩展的适用范围 86
8.1.4 用户如何让自己的代码使用 SIMD 86
8.1.5 使用 SIMD 可以预期获得多大的性能提升 87
8.2 SIMD 编程快速入门 87
8.2.1 一个简单的例子 87
8.2.2 变量声明 88
8.2.3 将标准类型的数据赋给扩展类型 88
8.2.4 扩展类型变量的运算 88
8.2.5 扩展类型的类型转换 88
8.2.6 扩展类型的打印 89
8.3 SW 26010 的运算控制核心和运算核心对 C 语言的扩展 89
8.3.1 数据类型的扩展 89
8.3.2 对界（Alignment） 89
8.3.3 SIMD 对 C 操作的扩展 90

8.3.4 SIMD 对 C 内部函数的扩展 92
8.3.5 扩展类型的类型转换 94
8.3.6 扩展类型与标准类型之间的数据交换 96
8.3.7 应用程序二进制接口（ABI） 97
8.4 串行程序的向量化 97
8.4.1 程序形式对向量化的限制 98
8.4.2 循环向量化步骤 99
8.4.3 如何更好地向量化 100
8.5 运算控制核心扩充内部函数、宏定义库 105
8.5.1 使用的符号说明 105
8.5.2 装入/存储操作宏定义 105
8.5.3 定点向量运算操作函数 109
8.5.4 浮点向量运算操作函数 122
8.5.5 SIMD 整理指令 133
8.5.6 赋值运算操作宏定义 136
8.5.7 扩展类型的打印函数 140
8.6 运算核心扩展内部函数、宏定义库 145
8.6.1 使用的符号说明 145
8.6.2 装入/存储操作宏定义 145
8.6.3 定点向量运算函数接口 146
8.6.4 浮点向量运算函数接口 153
8.6.5 数据整理函数接口 166
8.6.6 赋值函数接口 170
8.6.7 访问存储器并广播函数接口 170
8.6.8 寄存器通信函数接口 173
8.6.9 向量查表函数接口 177
8.6.10 扩展类型的打印函数 178
第九章 调试与典型问题的解决办法 179
9.1 通过 GPROF 工具调试性能 179
9.2 DEBUG 调试 181
9.3 对未初始化变量的处理 181
9.4 对大的静态数据的处理 182
9.5 嵌汇编 182
9.6 使用-IPA 和-OFAST 选项 182
9.7 识别循环归纳变量 183

第一章 概述

该用户手册的主要内容包括：神威编译器的使用指南、配置方法、优化选项说明等。该手册还介绍了相应程序语言的扩展以及和其他通用编译器的区别。

本文余后章节如下安排：

- 第二章 神威编译器快速入门
- 第三章 Fortran 编译器
- 第四章 C/C++ 编译器
- 第五章 加速线程库
- 第六章 优化快速入门
- 第七章 优化选项
- 第八章 使用 SIMD 扩展指令
- 第九章调试与典型问题的解决办法

如下约定贯穿全文：

约定	意义
命令	表示命令、文件、函数、以及路径名
变量	斜体字表示变量名或概念
用户输入	粗体字表示用户输入项
$	表示命令行提示符
[ ]	表示命令和指示行的可选项
…	表示重复前述信息
备注	表示重要信息

第二章 神威编译器快速入门

本章主要介绍使用神威编译器的快速入门方法。本编译器遵守 Unix 和 Linux 编译器标准协议，所生成目标码遵守 Linux ABI 协议，并可以在 SW 26010 芯片家族中运行。

2.1 安装说明

神威编译器是针对 C，C++，和 Fortran 的优化编译器套装，并提供运行时支持。神威编译器包含以下内容：

- 支持 SW 26010 体系架构 C 编译器
- 支持 SW 26010 体系架构 C++编译器
- 支持 SW 26010 体系架构 Fortran 编译器
- 说明文档集
- 库
- 调试器
- 二进制工具集

2.2 使用说明

神威编译器拥有三个不同的前端可以分别处理 C、C++、和 Fortran 程序，然后并遍历共同的优化和代码生成模块。不同的语言写的程序代码使用不同的编译驱动，不同目标核心代码生产通过不同选项区分，详细如下：

语言/目标核心	运算控制核心	运算核心	混合链接
C	sw5cc -host	sw5cc -slave	sw5cc -hybrid
C++	sw5CC -host	不支持	sw5CC –hybrid
Fortran 77 Fortran 90 Fortran 95	sw5f90 -host	sw5f90 -slave	sw5f90 -hybrid

神威编译器使用方法如下：

1、 编译命令安装路径：/usr/sw-mpp/bin

2、 ***sw5cc/sw5CC/sw5f90*** 编译程序需要增加指定目标核心的选项，-host 选项编译运算控制核心程序，-slave 选项编译运算核心程序。

3、 生成混合链接程序需要分别生成运算控制核心的.o 文件和运算核心.o 文件，最后使用链接器 ***sw5ld*** 进行链接（也通过直接调用 ***sw5cc/sw5CC/sw5f90*** 来间接调用 ***sw5ld***），链接的时候加-hybrid 选项。

4、 编程示例：

a) 独立的运算控制核心程序 master.c

sw5cc -host master.c -o master.out

b) 独立的运算核心程序 slave.c

sw5cc –slave slave.c -o slave.out

c) 运算控制核心程序 master.c、运算核心程序 slave.c

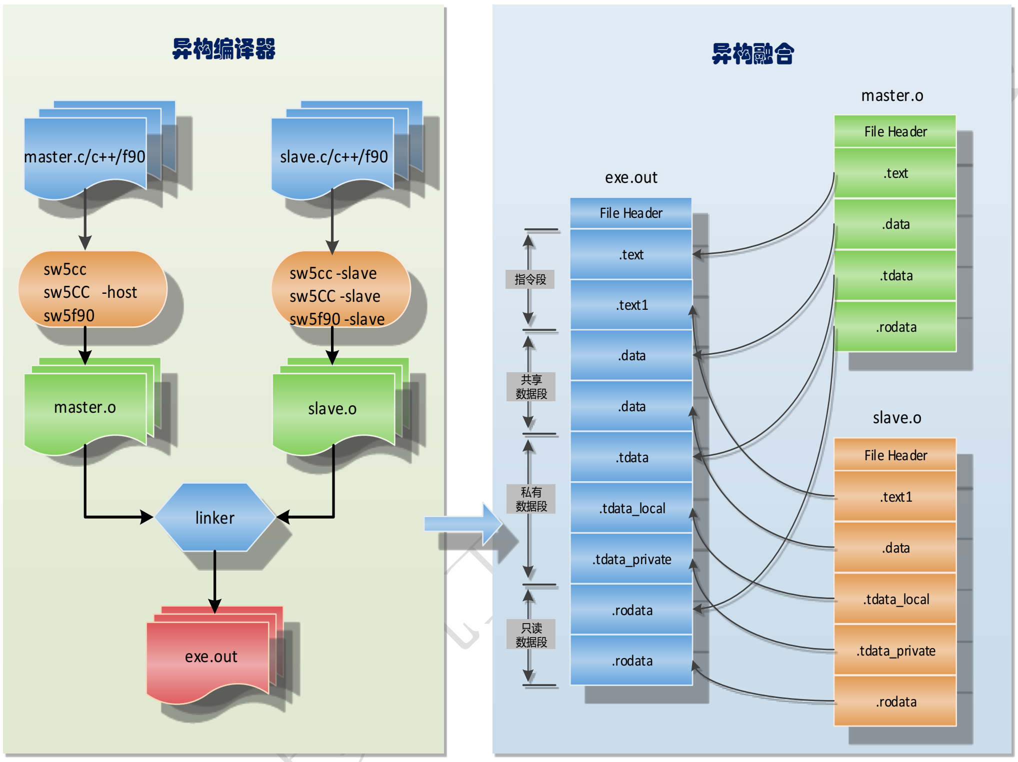
编译命令：

sw5cc -host -c master.c

sw5cc –slave -c slave.c

sw5cc -hybrid master.o slave.o -o exe.out

5、 编译过程如下图所示



6、 所有的编译器都可以用-o <文件名>来指定所生成的程序可执行文件名。

如果编译时用选项-v(或 -version)，编译器则会给出自身的版本信息，例如：

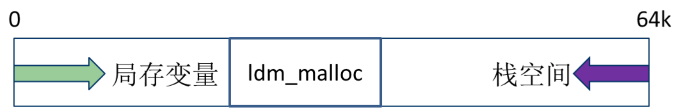
$ sw5cc -v

SWCC Compilers: Version 5.421 by xiaoq1 at hpserver0.localdomain on 2014-08-12 17:58:57 +0800

在 man 说明中给出了大量的可用选项说明，可以命令行中输入 man sw5cc 就可以看到所有跟编译器相关的 man 信息。

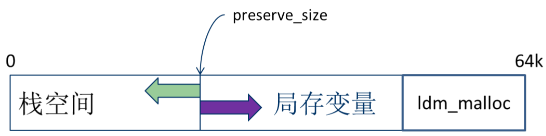
7、 使用编译选项：-preserve preserve_size，支持用户指定大小的栈空间预留：

不使用编译选项-preserve 的情况下，LDM 使用情况如下：



由于无法预测实际使用的栈空间大小，可能会发生栈空间与局存变量或是 ldm_malloc 空间相互冲突的情况，造成不可预料的错误。

使用-preserve preserve_size 编译，LDM 使用情况如下图所示：



这样可以避免空间冲突，栈地址小于 0 时，会有报错提示。这个预留空间的大小由用户指定，比较符合具体程序的具体特点。

2.3 其他编译器

2.3.1 基于 GCC 的编译器

基于 GCC 的编译器只是基础的功能性编译器，不提供 SIMD 扩展接口支持，不提供一些平台相关的优化技术。

编译器命令如下表：

语言/目标核心	运算控制核心	运算核心
C	swgcc	sw5gcc
C++	swg++	不支持
Fortran	swgfortran	sw5gfotran

2.3.2 老版编译器

编译器命令如下表：

语言/目标核心	运算控制核心	运算核心	混合链接
C	sw5cc.old -host	sw5cc.old -slave	sw5cc.old -hybrid
C++	sw5CC.old -host	不支持	sw5CC.old -hybrid
Fortran	sw5f90.old -host	sw5f90.old -slave	sw5f90.old -hybrid

2.4 常用工具

2.4.1 查看编译选项工具

神威编译器套集中的 swhow-compiled 工具能够查看编译选项和编译器版本，该工具在编译器安装完毕后可以在路径/usr/sw-mpp/bin 中找到（如果安装环境是非局部缺省的则可以在路径/<*install_directory*>/bin 中找到）。

如果.o 文件、.a 文件、或者可执行文件传给工具 swhow-compiled，将会打印出每个.o 文件的编译选项。同样适用于可链接库文件。

如下例所示，用 sw5cc 编译文件 my.c，然后用工具 swhow-compiled：

$ sw5cc –host my.c -o my

$ swhow-compiled my

输出：SWCC Compiler Version 5.421-sw-446 compiled my.c with options: -O2

2.4.2 编译器版本号

列出当前编译器版本号：

$ sw5cc/sw5CC/sw5f90 -v

列出所有编译器版本号：

$ sw5cc/sw5CC/sw5f90 -ver list

指定编译器版本编译：

$ sw5cc/sw5CC/sw5f90 -ver 版本号(5.421-sw-xxx)

2.5 输入文件类型

源文件名的格式通常为 filename.ext，其中 ext 是 1 到 3 字节的扩展名，不同的扩展名包含不同的意义。

扩展名	调用编译器驱动
.c	处理 C 语言程序
.C .cc .cpp .cxx	处理 C++语言程序
 .f .f90 .f95	Fortran 源文件 .f 是固定格式，没有调用预处理程序 .f90 是自由格式，没有调用预处理程序 .f95 是自由格式，没有调用预处理程序
 .F .F90 .F95	Fortran 源文件 .F 是固定格式，调用预处理程序 .F90 是自由格式，调用预处理程序 .F95 是自由格式，调用预处理程序

在用 sw5f90 编译扩展名为.f.、f90、或者.f95 的 Fortran 源文件时，可以在命令行中用选项-ftpp 来调用 Fortran 预处理程序或者-cpp 来调用 C 处理程序。在选项缺省的情况下，扩展名为.F、.F90、或者.F95 的源文件将缺省使用-cpp 的预处理程序。第 3.4.1 节将给出预处理的详细信息。

编译驱动器将根据文件扩展名来决定所调用的前端。比如，有些混合语言写的程序可以以如下方式编译：

$ sw5f90 –host stream_d.f second_wall.c -o stream

sw5f90 编译驱动器将识别到.c 扩展名并自动调用 C 前端来处理 second_wall.c 文件，在最后链接目标码生成 stream 可执行文件。

备注：GNU 的 make 并没有包含生成.f90 目标文件的规则，但可以在 Makefile 文件里面添加下面的语句来实现：

```
$.o:    %.f90
    $(FC)   $(FFLAGS)   -c   $<
$.o:    %.F90
    $(FC)   $(FFLAGS)   -c   $<
```

在第三章可以看到更多的有关兼容性和可移植性的信息。有关 GCC 的兼容性可以参考第 3.7 节的内容。

2.6 其他输入文件

其他的可能的输入文件，类似于 C/C++和 Fortran，比如汇编代码文件，目标文件和库文件等。如下文件可以在命令行当作输入文件。

扩展名	相关驱动器
.i	C 文件的预处理文件
.ii	C++文件的预处理文件
.s	汇编语言文件
.o	目标文件
.a	静态库
.so	动态库

2.7 一般编译选项

SWCC 编译和其他许多的 Linux 和 Unix 编译器一样，拥有许多命令行选项：

选项	作用
-c	为每个源文件生成一个中间目标文件，但是不进行链接
-g	为之后的调试生成调试符号信息
-I<dir>	为预处理头文件添加查找<路径>
-l<library>	在链接阶段指定需要链的库文件
-L<dir>	为链接阶段添加查找<路径>
-lm	在链接阶段使用 libm 数学库，在 C 程序中若使用了 exp(),log(),sin(),cos()函数则需要调用该库
-o<filename>	指定生成的可执行(库)文件的名字
-O3	生成高级优化的可执行代码
-O or -O2	生成优化的可执行代码
-pg	为 pathprof 分析程序生成反馈信息

更多的可用选项以及说明请参考 man 帮助文档。

2.8 共享库

神威编译器包括各种共享运行时库，所有的库都打包在/usr/sw-mpp/swcc/lib/gcc-lib/shenwei-swcc-linux/文件夹中。在链接可执行文件和共享库的时候，缺省情况下编译器会自动去链这些库文件，所以必须先将这些文件安装在程序运行所在的系统中。

如果想达到最大的可移植性或达到最佳性能也可以使用静态运行时库，使用静态库不需要动态装入程序段减少运行开销，不需要动态链接时候需要额外的执行环境支持。

在链接的时候想用静态库来代替动态库，只需要用-static 选项，例如，下面的代码就是用动态链接库的：

```
$ sw5cc –slave -o hello hello.c
$ ldd hello
        libc.so.6.1 => /lib/libc.so.6.1 (0x000002000003e000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x0000020000000000)
```

如果用-static 选项，就不需要用动态库了：

```
$ sw5cc -slave -o hello hello.c -static
$ ldd hello
        not a dynamic executable
$
```

在缺省情况下，神威编译器为了适应跨平台编译都是以-static 编译，运用静态库的。

2.9 内存模式支持

目前神威编译器支持两种内存模式：小和中间模式。在 SW 26010 系统中的缺省情况情况下，编译器支持中间内存模式。表示包含库的代码的偏移地址用符号 32 位表示，而包含库的数据偏移地址用符号 64 位表示。在这种模式下，所有可执行文件中的代码段必须小于 2GB，而包括静态数据和 BSS 的数据段允许超过 2GB。

编译器支持小模式(作为一种优化选项-m32)，代码和数据的偏移值用符号 32 位来表示。在这种模式下，所有的可执行文件代码段以及数据段的大小以及栈空间和动态分配数据空间总和必须少于 2GB。其中数据段是由静态数据段和 BSS 段组成。

备注: 神威编译器不支持和-m32 选项和共享模式一起用，即小模式并不支持 PIC 模式。

参看第 11.3 节中有关大目标文件的更多信息，GCC4.2.1 的文档中也有关与该主题的详细信息。

神威编译器不支持大内存模式，意味着代码的偏移值必须适应符号 32 位地址空间。可以用 Linux 的 size 命令来查看代码是否已经接近这些限制。

```
$ size bench
text      data      bss      dec      hex      filename
```

910219　　1448　　3192　　914859　df5ab　　bench

如果 text 段的值接近 2GB，那么就需要考虑内存模式的大小了。我们相信这种规模的代码是极其大了，最好确认是否真需要这种应用。bss 段和 data 段大小可以用中间内存模式来解决。

2.10 调试

选项-g 将使得神威编译器产生以 DWARF2.0 格式的数据，这些数据将应用在调试器 GDB 中。这些数据格式将包含在目标文件中。

-g 选项将自动和-O0 优化绑定在一起，除非在命令行中直接指定更高级优化，但是这些优化的代码变换将会使得生成调试信息更加困难。

查看第四章和第九章将会有调试的更详细信息。

2.11 反馈

通常每个程序都会有“热点”函数，也称为核心函数，即一小段代码和循环却占用大部分的执行时间。利用反馈是一种比较普遍的方法来定位程序中的核心函数。为了能够找到优化代码的位置和方式，可以先用 time 工具粗略的统计时间，然后判断是系统 load，应用程序的 load 还是系统资源影响了性能。然后使用 gprof 工具来找到程序的核心函数。一旦找到核心段，就可以改善该段代码或者利用这些信息来调整编译选项使之更好的性能达到。

time 工具可以给出 运行程序所用到的 real 时间、user 时间、和 system 时间。书写格式为：time ./<程序名 参数>。real 时间主要适用于并行程序时间的确定，但是如果系统中还有其它的访存操作，user 时间将更能准确的表示程序性能。如果程序占用的 system 时间比较多，但是你却并不想让系统中的非计算资源时间包括在程序性能内，你也可以用内核的反馈工具来定位原因。

其中其中编译器中的反馈工具为 gprof 和 gcov，相关的详细信息和例子说明参考第 9 章，下面的几个步骤是反馈相关所必须的：

1. 在编译和链接阶段均要添加-pg 选项，则生成二进制文件。
2. 执行时候输入相关数据，产生 gmon.out 文件，记录了一些反馈数据。
3. 运行 gprof　<程序名> gmon.out 执行生成反馈文件，标准的 gprof 输出含有两个表格：

a） 其中一个表格记录了每个函数消耗的时间和被调用的次数

b） 另外一个表格描述了整个程序的函数调用图，表示函数之间的调用关系。同时也给出了每个函数以及所有它调用的子函数的时间。

备注：gprof 工具和 OpenMP 合用的时候会产生段违例，OpenMP 应用程序运行时候产生多个线程。

详细的信息参考第 9 章的内容。

第三章 Fortran 编译器

神威编译器支持 Fortran 77、Fortran 90 和 Fortran 95。具体包括：

- 遵从标准 ISO/IEC 1539:1991 程序设计语言–Fortran (Fortran 90)
- 遵从更新的标准 ISO/IEC 1539-1:1997 程序设计语言–Fortran (Fortran 95)
- 遵从标准 ISO/IEC TR 15580: Fortran: 浮点异常处理。更完整的描述也可参看 ISO/IEC 1539-1:2004, Fortran 2003 标准。
- 遵从标准 ISO/IEC TR 15581: Fortran: 增强的数据类型功能
- 遵从标准 ISO/IEC 1539-2: 可变长字符串（见 3.4.3)
- 遵从标准 ISO/IEC 1539-3: 条件编译（见 3.4.4)
- 支持 FORTRAN 77 (ANSI X3.9-1978) 传统程序
- 提供对上述语言定义的通用扩展的支持
- 可与 GNU Fortran 77 编译器生成的目标码进行链接
- 生成的代码遵从 SW 26010 ABI 的约定

3.1 使用 Fortran 编译器

使用如下命令启动编译器：

$ sw5f90

一般地，当输入文件的后缀为.F 或.f，编译器把它当成固定源格式文件处理；当输入文件的后缀为.F90、.f90、.F95 或.f95 时，编译器把它当成自由源格式文件处理。这个规则可以使用选项-fixedform 和-freeform 进行改变。更多的固定源和自由源文件的信息参见 3.2。

缺省情况下，所有后缀为.F，.F90 或.F95 的文件都会先使用 C 预处理程序（-cpp)。如果指定-ftpp 选项，所有文件都会使用 Fortran 预处理程序进行预处理，不管是什么后缀。有关预处理的内容详见 3.5.1 和 3.5.2。

不加任何选项使用编译器，编译器***缺省使用-O2 级优化***。下面两个命令是等价的：

$ sw5f90 -host test.f90

$ sw5f90 -host –O2 test.f90

缺省时编译器采用 O2 级优化，这时编译器会进行广泛的优化，通常会缩短程序运行时间，但可能会增加编译的时间。O0 级不会做任何优化。O1 级优化只进行局部的优化。O3 级优化进行更大胆的优化，这些优化也有可能不会提高程序执行的效率。有关-O 选项的更多信息参见 7.1。

使用-ipa 选项打开过程间分析模块：

$ sw5f90 -c -ipa matrix.f90

$ sw5f90 -c -ipa prog.f90

$ sw5f90 -ipa matrix.o prog.o -o prog

注意在链接命令行上也需要使用-ipa 选项。这是启动 IPA 链接所必需的。有关 IPA 的更多信息参见 7.2。

备注：编译器为了更好的性能，典型的方法是给 Fortran 程序在栈中分配数据空间。大多数 Linux 系统为程序预留的栈空间非常有限。如果你在这种系统中试图运行一个栈空间很大的 Fortran 程序，将会打印出错信息并退出。你可以通过执行 shell 命令“ulimit”(bash)或“limit”（tcsh）增加栈空间的限制。有关 Fortran 编译器栈大小的更多信息参见 3.12。

3.2 固定源和自由源格式文件

固定源格式文件遵循比较古老的 Fortran 标准，源程序每行的前 6 个字符有特殊的含义。如果一行的第一个字符是’C’，’！’或’*’，则表示该行的作为注释行处理。如果’!’出现在一行的任何位置（第 6 个字符位置除外），则表示该行随后的内容作为注释处理。只包含空字符的行和空行均作为注释。

若一行的第 6 个字符为非空字符，则表示该行是上一行的续行。根据 Fortran 标准，最多可以允许 19 个续行，但神威编译器支持最多 499 个续行。

每行的第 7 个字符到第 72 个字符是正式的源程序。可以使用分号区隔同一行上的多条语句。但在一行的第 7 个字符到第 72 个字符中分号不能是第一个非空字符。

第 1 到 5 个字符为语句标号。既然语句标号不能出现在续行上，所以续行上的前五个字符位置必须是空的。

自由源格式文件在代码编排上限制很少。一行可以任意长，一行的末尾为&，则下一行为上一行的续行。语句标号可以放在一行的任何位置，只要它前面有一个空字符。一行中的任何位置出现‘！’则表示注释的开始。

针对两个原格式文件，使用选项-fixedform 和-freeform 进行改变。

3.3 模块

当编译一个 Fortran 模块时，模块信息放到一个名字为 MODULENAME.mod 的文件中。该文件缺省放在命令执行的当前目录下。可以使用-module 选项改变缺省的目录位置。MODULENAME.mod 文件允许其它 Fortran 文件使用模块中定义的过程、函数、变量及其它实体。模块文件可以理解成与 C 语言头文件类似。

根 C 语言头文件一样，你可以使用-I 选项指定模块文件放置的目录：

$ sw5f90 -host -I/work/project/include -c foo.f90

它指示编译器在/work/project/include 目录下去寻找.mod 文件。如果 foo.f90 包含’use arith’语句。将会按如下顺序查找模块文件：

/work/project/include/ARITH.mod

./ARITH.mod

3.3.1 出现的顺序

如果模块和使用该模块的’use’语句出现在同一源程序文件，则该模块必须出现在前面。

如果模块跟使用该模块的’use’语句出现在不同的源程序文件中，则包含模块的文件必须在前面编译。

如果在一个命令行上编译所有文件，则包含模块的文件在命令上必须出现在包含使用该模块的’use’语句的文件前面：

$ sw5f90 -host mymodule.f95   myprogram.f95

3.3.2 与其它目标程序进行链接

包含模块的源程序编译后生成模块信息文件（.mod）的同时生成目标文件(.o)，尽管可能源程序文件中除了该模块外什么也没有了。该目标文件必须在后面与其它目标程序一起进行链接。如果是在同一个命令行上进行编译和链接，则模块对应的目标文件会自动被链接进来，但如果链接命令是分开的，你必须留意别丢掉只含有模块的源程序对应的目标文件。不需要关注目标文件在命令行上的顺序。例如：

$ sw5f90 -host -c   mymodule.f95

$ sw5f90 -host -c   myprogram.f95

$ sw5f90 -hybrid myprogram.o   mymodule.o

值得注意的是包含多个模块的源程序文件会产生一个目标文件(.o)，文件名对应于源程序文件名，以及多个模块信息文件(.mod)，文件名对应于模块名本身。例如，产生MYMODULE1.mod、MYMODULE2.mod、MYMODULE3.mod 和 my3modules.o：

$ sw5f90 -host -c   my3modules.f95

下面再生成使用这些模块的主程序：

$ sw5f90 -host -c   myprogram.f95

$ sw5f90 -hybrid myprogram.o   my3modules.o

3.3.3 模块相关的报错信息

只要模块中出现错误，就会在模块的第一行报一个出错信息，也许真正的出错点离得很远。真正的出错信息在后面也会报出。下面给出一个例子。

这里有个程序，hellow.f95，包含如下一个模块：

```
MODULE HELLOW
CONTAINS
SUBROUTINE HELLO( )
SPRINTZ *,"Hello, World!"
END SUBROUTINE HELLO
END MODULE HELLOW
```

然后编译包含此模块的文件，将会产生一些错误信息：

```
$ sw5f90 hellow.f95
MODULE HELLOW
^
sw5f90-855 sw5f90: ERROR HELLOW, File = hellow.f95, Line = 1, Column = 8
The compiler has detected errors in module "HELLOW". No module information file will be
```

created for this module.

```
SPRINTZ *,"Hello, World!"
^
sw5f90-724 sw5f90: ERROR HELLO, File = hellow.f95, Line = 5, Column = 11
Unknown statement. Expected assignment statement but found "*" instead of "=" or "=>".
sw5f90: SWCC Fortran Version 2.1.99 (f14) Tue Nov 21,
2006 14:22:16
sw5f90: 9 source lines
sw5f90: 2 Error(s), 0 Warning(s), 0 Other message(s), 0 ANSI(s)
sw5f90: "explain sw5f90-message number" gives more information
about each message
```

注意到真正的错误现场在第一个报错的后面也打印了出来。

3.4 扩展

Fortran 编译器支持许多 Fortran 标准以外的扩展，这一节将进行具体地介绍。

3.4.1 对不同空间属性的支持

3.4.1.1 语言扩展

SW 26010 运算核心的用户虚空间分为 LDM 空间和主存空间，主存空间进一步可以分为系统区、用户连续区和用户交叉区。

SW 26010 的物理空间编排如下表所示：

区域		PA[39]	PA[38]	PA[37]	PA[36:35]	PA[10:9]
连续区	系统连续	0	0	0	核组号	
	用户连续	0	0	1	核组号	
用户交叉区		0	1			核组号
核组IO空间		1	0	0	核组号	
全芯片IO空间		1	0	1		

其中用户连续区根据软件使用方式又可以进一步分为私有连续区和共享连续区。

SW 26010 所有核看到的虚空间试图都是一样，操作系统会根据不同空间属性统一进行物理空间映射；默认所有的连续空间都是映射到本核组空间，而交叉段空间是全芯片空间统一映射。

运算控制核心虚空间编排如下：

空间类型	虚空间
	起始地址
系统连续 （本地可 cache）	0x00,0000,0000

用户私有连续 （本地可 cache）	0x20,0000,0000 【OS 空间：0x20,0000,0000-0x20,0000,FFFF】 【用户空间起始地址：0x20,0001,0000】
用户共享连续(只读) （本地可 cache）	0x4F,F000,0000 【OS 空间：0x4F,F000,0400-0x4F,F000,FFFF】 【用户空间起始地址：0x4F,F001,0000】
用户共享连续(读写) （本地可 cache）	0x50,0000,0000
用户共享交叉 （不可 cache）	0x60,0000,0000
核组 IO 空间 （不可 cache）	0x80,0000,0000
芯片 IO 空间 （不可 cache）	0xa0,0000,0000

从核虚空间编排如下：

空间类型	虚空间
	起始地址
LDM 空间	0x00,0000,0000
用户私有连续	0x20,0000,0000 【OS 空间：0x20,0000,0000-0x20,0000,FFFF】 【用户空间起始地址：0x20,0001,0000】
用户共享连续(只读)	0x4F,F000,0000 【OS 空间：0x4F,F000,0400-0x4F,F000,FFFF】 【用户空间起始地址：0x4F,F001,0000】
用户共享连续(读写)	0x50,0000,0000
用户共享交叉	0x60,0000,0000

为了支持 SW 26010 多种地址空间，SW 26010 编译器提供了 Fortran 语言扩展，支持多种空间属性。

Fortran 程序除了函数内临时变量外，分下列变量类型：

共享变量：common 变量，运算控制核心和运算核心共享变量。

运算核心局存私有变量：common 变量，放在用/local_*/关键字标识的区间中，并在程序执行代码前用!$omp threadprivate 编译指示指定。

运算核心私有变量：common 变量，放在用/private_*/关键字标识的区间中，并在程序执行代码前用!$omp threadprivate 编译指示指定。

例子：ex1.f90

```
        subroutine slave_FUNC
```

```
        implicit none
        integer :: x(10), z(30),a(10)
        common /local_g1/ x(10)              /*运算核心局存私有变量*/
        common /private_g3/ z(30)            /*运算核心私有变量*/
        common /g4/ a(10)                    /*运算控制核心和运算核心共享变量*/
!$omp threadprivate(/local_g1/ ,/private_g3/)
        程序执行代码
        end
```

程序编译：sw5f90 -slave -mp ex1.f90

注意：

1、对运算核心以及核组数据(/local_**/,/private_**/)的初始化操作必须放在运算核心代码区，否则无法对数据进行加载分配；如果没有初始化，则不需要进行 block data 操作，直接用 common 属性来申明数据。

2、通过关键字对变量进行分类申明后，运算控制运算控制核心和运算核心运算控制核心和运算核心代码都必须用!$omp threadprivate 进行指示说明，将变量属性申明为私有属性。

3.4.1.2 主从代码空间

SW 26010 的可执行代码段位于用户共享连续只读空间，具体的说用户空间起始地址是 0x4F,F001,0000，编译器通过代码段名字的不同来区分具体的代码是属于运算控制核心还是属于从核。运算控制核心代码以 text 为段名，从核运算核心代码以 text1 为段名。对位于运算核心 text1 段中的运算核心程序，编译器和底层工具链会自动对函数增加”slave_”前缀，如函数 func( )会自动换名为 slave_func( )。用户已经手工添加 slave_前缀的则不再重复添加。

对有特殊需求要手工书写汇编的程序员，必须遵守 SW 26010 约定的汇编规则，因为神威编译器和底层工具会依赖于特定的汇编指示，包括机器模式指示和段名指示进行相应的处理和优化。

一个典型的运算核心汇编文件头部具有下面的汇编指示：

```
.set      noat
.set      noreorder
.arch     sw5
.section .text1, "ax", "progbits"
.align    4
……
```

如果用户不遵守上述指示，可能汇编和链接过程不会报错，但程序运行期的行为可能出错。建议使用编译器（sw5cc –slave -S）处理高级语言生成一个汇编框架，此框架满足所有约定。

3.4.2 SIMD 扩展

详细接口见第八章“SIMD 接口”。

3.4.3 普通 instrinsic

普通 intrinsic 一般包括一条 intrinsic 语句对应一条或多条汇编：

get_row()

get_col()

get_cg()

……

3.4.4 REAL 和 INTEGER 类型的升级

-r8 -i8 对应于把 REAL 和 INTEGER 类型的数据从缺省 4 字节类型长度升级成 8 字节类型长度。这对移植整型和浮点数据缺省为 8 字节长的 Cray 代码很有用。不过要小心可能与外部库的类型不一致。

备注：-r8 -i8 的选项只会影响缺省的实型和整型数据，不影响包含显式指定类型长度 KIND 的变量。如果一个 4 字节缺省 real 或 integer 类型参数传递到子程序中声明为 KIND=4，这可能会导致出错。使用显式 KIND 值的代码不易移植因此不推荐。正确的 KIND(例如 KIND=KIND(1) 或 KIND=KIND(0.0d0))用法不会有任何问题。

3.4.5 Cray 指针

Cray 指针是 Fortran 的一种数据类型扩展，用于声明动态对象，跟 Fortran 指针不同。Cray 和 Fortran 指针都使用 POINTER 关键字，但编译器可以通过如下方式来区分这两种指针。

Cray 指针的声明方法为：

POINTER (<pointer>, <pointee>)

Fortran 指针的声明方法为：

POINTER :: [<object_name>]

SWCC 实现 Cray 指针的方法采用的是 Cray 实现，相比其它编译器，这是一种较严格的实现。特别的，SWCC Fortran 编译器对 pointer 与 integer 分开。如果你通过 p = ((p+7) / 8) * 8 去对 pointer 进行对界，编译器会报错。

3.4.6 编译指示

在程序单元中的编译指示只对该单元起作用，在程序单元结束后自动恢复缺省值。如果编译指示出现在程序单元以外，则会改写缺省值，因此作用于随后的所有余下的代码，除非又被后续编译指示改写掉。

文件中的编译指示缺省优先于命令行的选项。为了让命令行选项优先于编译指示，可以使用如下选项：

-LNO:ignore_pragmas

SWCC 编译器支持下面的预取编译指示。

3.4.6.1 F77 或 F90 预取指示

1) C*$* PREFETCH(N [,N]) 指示 cache 预取的级别。作用范围为包含该编译指示的整个函数。N 可以为如下值：

0 关闭预取（缺省）

1 打开预取，但比较保守

2 打开预取，比较大胆（当预取打开后这是缺省方式）

2) C*$* PREFETCH_MANUAL(N) 指出是否进行手工预取（通过编译指示）。作用范围为包含该编译指示的整个函数。N 可以为如下值：

0 忽略手工预取

1 进行手工预取

3) C*$* PREFETCH_REF_DISABLE=A [, size=num] 该指示关闭当前函数中所有对数组 A 的预取。自动预取器（若打开）会忽略数组 A。大小的参数可以用于容量分析。作用范围：包含指示的整个函数。

Size=num 代表循环中数组访问的大小，单位为 Kb，这是一个可选参数，但必须是常数。

4) C*$* PREFETCH_REF=array-ref, [stride=[str] [,str]], [level=[lev] [,lev]], [kind=[rd/wr]], [size=[sz]] 该指示产生一个针对特定内存区域的预取指令。编译器根据提供的访问在当前嵌套循环中搜索相应的数组访问。如果找到一个这样的访问，则该访问就跟指示的预取及参数联系起来。如果没找到，则该预取节点自由浮动，将会在很大范围内进行调度。

所有在该循环嵌套中的数组访问都将被自动预取器（若打开）忽略。若有 size 参数，自动预取器（若打开）将会根据计算出来的大小减小有效 cache 容量。

编译器尝试每 stride 个迭代发射一个预取，但对此不保证。采用冗余预取的方式可能会比变换后（比如插入一些条件语句，会带来其它额外开销）更好。

作用范围：没有范围。只产生一条预取指令。

该指示包含下面的一些参数：

array-ref 必须的。访问本身，例如 A(i, j)。

str 可选。在本循环中每 str 个迭代预取一次。缺省值为 1。

lev 可选。指定预取的存储器层次。缺省为 2。如果 lev=1，从 L2 预取到 L1。如果 lev=2，则从内存预取到 L1。

rd/wr 可选。缺省为 read/write。

sz 可选。循环数组访问的容量(单位：Kb)。必须为常数。

3.4.6.2 通过编译指示改变优化

优化选项可以通过在用户程序中插入编译指示而改变。在 Fortran 中，编译指示的格式如下：

C*$* options <”list-of-options”>

在函数作用范围内可以插入任意数量的编译指示。每一个编译指示只影响当前的函数单元。每一个指示中也可以包含任意多个不同的选项，选项之间通过空格分开，而且都必须在同一组引号之内。当编译到下一个函数时，编译器的选项设置自动恢复到与命令行一致的情况。

当前版本中选项指示只处理了有限的一些选项，最终对编译优化起作用。

- 没有处理的选项就不会报警或报错。
- 这些指示只有在后端时处理。因此只会影响处理后的优化。
- 另外，它不会影响后端模块启动的优化。例如指定-O0 不会关闭全局优化器，因为后端会忠于特定的优化级别。
- 除了优化级别选项外，只有下面的选项组被处理：-LNO, -OPT 和-WOPT。

3.5 编译器和运行时库的特点

编译器提供三个不同的预处理选项：-cpp，-ftpp 和-fcoco。

3.5.1 使用-cpp 进行预处理

在进入编译器前端之前，源程序可以先经过预处理程序处理。预处理器在文件中搜寻特定的指示，并基于这些指示保留或去除部分程序代码，嵌入其它文件或定义和替换宏。在缺省情况下，Fortran .F、.F90 和.F95 文件会自动使用 C 预处理程序-cpp 处理。

3.5.2 使用-ftpp 进行预处理

Fortran 预处理程序-ftpp 接受许多 C 预处理程序中的“#”指示，但也有显著的区别（例如，它不接受 C 风格的注释——以”/*”起始的多行注释）。如果你希望使用 C 预处理程序处理后缀为.f、.f90 和.f95 的 Fortran 程序，你可以使用-cpp 选项。如果你再命令行上没有使用-ftpp（选择 Fortran 预处理程序）或-cpp（选择 C 预处理程序）选项，这些文件就不会被预处理。

3.5.3 支持可变长的字符串

在神威编译器中支持 ISO/IEC 标准 1539-2，提供对可变长字符串的支持。这是 Fortran 标准的可选项。你可以下载编译此模块。网址如下：

http://www.fortran.com/fortran/iso_varying_string.f95

3.5.4 使用-fcoco 进行预处理

在神威编译器中，Fortran 编译器支持 ISO/IEC 1539-3 条件编译预处理程序。若你指定-fcoco 选项，编译器对每一个源程序文件在编译之前进行-fcoco 预处理，而不是缺省的根据文件后缀为.F、.F90 和 F95 进行 cpp 预处理而对后缀为.f、.f90 和.f95 的文件不进行预处理。ISO/IEC 标准没有为预处理程序规定任何命令行选项，作为一种扩展，我们增加了-I 和-D 选

项，这跟-cpp 和-ftpp 预处理程序是一样的。

与其它预处理程序一样，选项-Isubdir（最后的"/"不是必需的）告诉预处理程序把 subdir 加到搜寻头文件的目录列表中去。需要区分的是-fcoco 与-cpp 和-ftpp 预处理程序不一样，它需要所有标识符具有数据类型，所以类似-DIVAR=5 的选项相当于声明了一个常数（不是变量）IVAR，类型为 integer，值为 5。同样，-DLVAR 选项相当于声明了一个逻辑类型常数，其值为" .true."。只允许整型和逻辑类型。你可以使用-D 选项改变源程序中声明的常数标识符的值。

标准的处理方式是规定预处理程序通过读一个"setfile"文件来定义常数、变量和操作模式，但没有指出怎样找 setfile。在编译器中如果使用-fcoco，预处理程序在当前目录下寻找 coco.set 文件。如果没有该文件，则预处理程序在没有 setfile 的情况下进行处理。你也可以使用选项如-fcoco=somedir/mysettings，则预处理程序使用文件 somedir/mysettings。你不能使用-D 选项覆盖 setfile 中声明的常数。

本特性基于的开放源码包中有更多的扩展和命令行选项，具体可以参看 http://users.erols.com/dnagle/coco.html。为了把这些选项通过编译器总控传递给预处理程序，你可以使用选项-Wp, <*options*>。例如使用-Wp, -m 可以-m 选项传给预处理程序关闭宏的预处理。值得说明的是，你使用 SWCC 编译器时，该网页上有关传递文件名给预处理器和识别 setfile 不需要再考虑，因为编译器已为你自动把源程序名传给预处理器，然后得到预处理程序的输出接着往下编译，根据前面介绍的方法识别 setfile。

有关-fcoco 的更多信息可以参看联机帮助文件。

3.5.5 预定义的宏

神威编译器为预处理程序提供了一些特定的宏。当你使用 C 预处理程序 cpp 处理 Fortran 程序，或依赖.F、.F90 和.F95 后缀缺省使用 cpp 预处理程序时，神威编译器使用 C 预处理程序缺省用到的宏以外，额外增加了如下的宏：

LANGUAGE_FORTRAN

_LANGUAGE_FORTRAN 1

_LANGUAGE_FORTRAN90 1

LANGUAGE_FORTRAN90 1

__unix 1

unix 1

__unix__ 1

注意：当使用的优化级别为-O1 或更高时，编译器会为 cpp 设置宏__OPTIMIZE__。有关 cpp 的宏请参看 4.2.1.1。

如果你使用 Fortran 预处理器-ftpp，只有五个预先定义的宏：

LANGUAGE_FORTRAN 1

__LANGUAGE_FORTRAN90 1

LANGUAGE_FORTRAN90 1

__unix 1

unix 1

备注：Fortran 缺省使用 cpp。只有你在命令行上指定了-ftpp 选项才会使用 Fortran 预处理程序。

下面的命令可以打印出使用-cpp 处理 Fortran 文件时的所有“#define”宏：

$ echo > junk.F90; sw5f90 -host -cpp -Wp,-dD -E junk.F90

对于 Fortran 预处理程序（-ftpp）没有找到相应的方法列出所有的预定义宏。有关怎样查到在 C 和 C++中预定义宏的方法参见 4.2.1.1。

对于-fcoco 预处理程序没有任何预定义宏。

3.5.6 Fortran 90 内部向量

现代的 Fortran 提供机制允许程序获得动态分配对象的一些信息，例如数组和字符串的长度等。获取这些信息的语言结构实例有 ubound 和 size 内部函数。

为了实现这种结构，编译器通过“dope vector”的数据结构维护这些信息。如果需要理解这个数据结构的信息内容，可以查看源码文件 clibinc/cray/dopevec.h。

3.5.7 边界检查

Fortran 编译器可以对数组进行边界检查。使用-C 选项打开该功能：

$ sw5f90 -C gasdyn.f90 -o gasdyn

产生的代码会检查所有的数组访问，保证它们不会出现越界访问。如果存在数组访问越界，程序运行时你会看到在错误输出上打印出报警信息。

$ ./gasdyn

lib-4961 : WARNING

Subscript 20 is out of range for dimension 1 for array

’X’ at line 11 in file ’t.f90’ with bounds 1:10.

如果你设置环境变量 F90_BOUNDS_CHECK_ABORT 为 YES。结果程序会在发现越界访问后直接退出。

可以确定，数组边界检查会影响程序性能，所以它只有在调试时打开，生成性能代码时关闭。

3.5.8 伪随机数

Fortran 标准库中实现了一个伪随机数生成器（PRNG），它是一个非线性、累积、反馈的 PRNG，具有 32 个入口的长种子表。PRNG 的周期接近 16*((2**32)-1)。

3.6 混合编程

如果你有一个很大的应用，同时包含 Fortran 代码和其它语言的代码，其中应用的 main 入口是 C 或 C++的，你可以使用 sw5cc 或 sw5CC 取代 sw5f90 来进行链接。这样做的时候，

还必须把 Fortran 的运行时库放到链接命令行上。例如，你可以这样执行：

$ sw5CC –hybrid -o my_big_app file1.o file2.o -lfortran

3.6.1 C 和 Fortran 之间互相调用

在 C 和 Fortran 之间互相调用，要注意两点：

■ 把 Fortran 函数名映射成 C 函数名

■ 保证参数类型匹配

一般地，sw5f90 函数名"x"若不包含下划线则创建一个链接符号"x_"，若函数名中包含下划线如"x_y"，则创建链接符号"x_y__"（注意有两个下划线）。相对而言 sw5cc 函数名不会在链接符号中加下划线。你可以在 C 代码中遵循这个规则：使用"x_"，这样就会与 Fortran 的"x"匹配。或者使用选项-fdecorate，它提供把 Fortran 名字映射特定的链接符号（两个名字可能很不一样），详情见 man sw5f90，或者你可以使用选项-fno-underscoring，但这样创建的符号经常与 Fortran 和 C 的运行时库出现冲突，故不建议使用。

通常 sw5f90 的参数传递方式是传地址，所以在 C 中要使用指针去表示 Fortran 的参数。大多数情况你也可以在 Fortran 中使用内部函数%val()实现参数传值。

程序设计者要特别留心参数数据类型是否一致。例如，sw5f90 integer*4 匹配 C int，integer*8 匹配 C long long，real 匹配 C float（提供的 C 函数有显式原型）和 double precision 匹配 C double。Fortran character 可能存在问题，它除了传第一个字符对应的地址参数外，在参数列表末尾中增加了一个整型参数说明字符串的长度。Fortran Cray 指针由 pointer 语句声明，对应与 C 指针，但 Fortran 90 指针，定义了 pointer 属性，这是 Fortran 所专有的。sequence 关键字使得 Fortran 90 的结构大多数情况会跟 C 结构的布局一致，但最好还是要通过实验确认。对于数组，最好仅限于使用 Fortran 77 提供的数组形式，Fortran 90 对于数组增加了数组结构存储数组信息，这些 C 是无法处理的。

因此，例如，参数"a (5, 6) "或"a (n) "或"a (1:*) "可以简单地传相应地址给 C 数组，而"a (:,:)"或可分配数组或 Fortran 90 指针数组在 C 中没有对应的类型。

> ***备注***：Fortran数组在内存中以列优先顺序放置，而C数组是行优先顺序。另外还一个东西需要调整，C数组的下标从0开始，而Fortran数组下标缺省从1开始，而且还可以指定从其它值开始。
>
> 在C++和Fortran之间互相调用更加困难，这跟C和C++之间互相调用存在困难是一样的：C++编译器为了实现重载必须把符号名“变乱”，C++编译器揉进了数据结构的信息（比如虚拟表指针），其它编译器无法弄懂这些。最简单的办法是在C++源代码中使用extern "C"，产生C兼容的接口，从而消除了与C和Fortran交互的问题。

3.6.2 调用举例

这里有三个可以编译运行的文件用来说明 C 和 Fortran 之间怎样互相调用。

C 代码如下（c_part.c）：

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

```
#include <alloca.h>
#include <string.h>
extern void f1_(char *c, int *i, long long *ll, float *f,
   double*d, int *l, int c_len);
/* Demonstrate how to call Fortran from C */
void call_fortran() {
   char *c = "hello from call_fortran";
   int i = 123;
   long long ll = 456ll;
   float f = 7.8;
   double d = 9.1;
   int nonzero = 10; /* Any nonzero integer is .true. in Fortran */
   f1_(c, &i, &ll, &f, &d, &nonzero, strlen(c));
}
/* C function designed to be called from Fortran, passing arguments by * reference */
void c_reference__(double *d1, float *f1, int *i1, long long *i2,
char * c1,int *l1, int *l2, char *c2, char *c3, int c1_len, int c2_len,
int c3_len) {
/* A fortran string has no null terminator, so make a local copy
and add a terminator. Depending on the situation, it might be preferable
to put the terminator in place of the first trailing blank. */
char *null_terminated_c1 = memcpy(alloca(c1_len + 1), c1, c1_len);
char *null_terminated_c2 = memcpy(alloca(c2_len + 1), c2, c2_len);
char *null_terminated_c3 = memcpy(alloca(c3_len + 1), c3, c3_len);
null_terminated_c1[ c1_len] = null_terminated_c2[ c2_len] =
   null_terminated_c3[ c3_len] = '\ 0';
printf("d1=%.1f, f1=%.1f, i1=%d, i2=%lld, l1=%d, l2=%d, "
   "c1_len=%d, c2_len=%d, c3_len=%d\n",
   *d1, *f1, *i1, *i2, *l1, *l2, c1_len, c2_len, c3_len);
printf ("c1='%s', c2='%s', c3='%s'\n",
   null_terminated_c1, null_terminated_c2, null_terminated_c3);
fflush(stdout); /* Flush output before switching languages */
call_fortran ();
}
/* C function designed to be called from Fortran, passing arguments by * value */
int c_value__(double d, float f, int i, long long i8) {
   printf("d=%.1f, f=%.1f, i=%d, i8=%lld\n", d, f, i, i8);
   fflush(stdout); /* Flush output before switching languages */
```

```
  return 4; /* Nonzero will be treated as ".true." by Fortran */
}
```

下面是 Fortran 源程序(f_part.f90):

```
program f_part
  implicit none
  ! Explicit interface is not required, but adds some error-checking interface
  subroutine c_reference(d1, f1, i1, i2, c1, l1, l2, c2, c3)
    doubleprecision d1
    real f1
    integer i1
    integer*8 i2
    character* (*) c1, c3
    character*4 c2
    logical l1, l2
  end subroutine c_reference
  logical function c_value(d, f, i, i8)
    doubleprecision d
    real f
    integer i
    integer*8 i8
  end function c_value
  logical l
  pointer (p_user, user)
  character*32 user
  integer*8 getlogin_nounderscore ! File decorate.txt maps this to
  external getlogin_nounderscore ! "getlogin" without underscore
  intrinsic char
  ! Demonstrate calling from Fortran a C function taking arguments by reference
  call c_reference(9.8d0, 7.6, 5, 4_8, ’hello’, .false., .true., &’from’, ’f_part’)
  ! Demonstrate calling from Fortran a C function taking arguments by value.
  l = c_value(%val(9.8d0), %val(7.6), %val(5),%val(4_8))
  write(6 , "(a,l8)") "l=", l
  ! "getlogin" is a standard C library function which returns "char *".
  ! When a C function returns a pointer, you must use a Cray pointer
  ! to receive the address and examine the data at that address,
  ! instead of assigning to an ordinary variable
  p_user = getlogin_nounderscore()
  write(6, "(3a)") "’", user(1:index(user, char(0)) - 1), "’"
```

```
end program f_part
! Subroutine to be called from C
subroutine f1(c, i, i8, f, d, l)
   implicit none
   intrinsic flush
   character* (*) c
   integer i
   integer*8 i8
   real f
   doubleprecision d
   logical l
   write(6, "(3a,2i5,2f5.1,l8)") "’", c, "’", i, i8, f, d, l
   call flush(6); ! Flush output before switching languages
end subroutine f1
```

第三个文件(decorate.txt)：

```
getlogin_nounderscore getlogin
```

编译运行这三个文件（c_part.c，f_part.f90 和 decorate.txt）的方式如下：

$ sw5f90 -host -Wall -intrinsic=flush -fdecorate decorate.txt f_part.f90 c_part.c

$ ./a.out

```
d1=9.8, f1=7.6, i1=5, i2=4, l1=0, l2=1, c1_len=5, c2_len=4,
c3_len=6 c1=’hello’, c2=’from’, c3=’f_part’
’hello from call_fortran’ 123 456 7.8 9.1 T
d=9.8, f=7.6, i=5, i8=4
l= T
’johndoe’
```

3.6.3 C 访问公用块举例

在 Fortran 90 模块中变量合并到两个公用块中，一个用于初始化数据，一个用于未初始化数据。可以通过使用-fdecorate 从 C 程序访问这两个公用块，举例如下：

$ cat mymodule.f90

```
module mymodule
   public
      integer :: modulevar1
      doubleprecision :: modulevar2
      integer :: modulevar3 = 44
      doubleprecision :: modulevar4 = 55.5
end module mymodule
program myprogram
```

```
   use mymodule
   modulevar1 = 22
   modulevar2 = 33.3
   call mycfunction ()
end program myprogram
```

$ cat mycprogram.c

```
#include <stdio.h>
extern struct {
   int modulevar1;
   double modulevar2;
   } mymodule_data;
extern struct {
   int modulevar3;
   double modulevar4;
   } mymodule_data_init;
void mycfunction ()
{
   printf ("%d %g\n", mymodule_data.modulevar1,
   mymodule_data.modulevar2); printf ("%d %g\n",
   mymodule_data_init.modulevar3,
   mymodule_data_init . modulevar4);
}
```

$ cat dfile

```
.data_init.in.mymodule mymodule_data_init
.data.in.mymodule.in.mymodule mymodule_data
mycfunction mycfunction
$ sw5f90 –host -fdecorate dfile mymodule.f90 mycprogram.c
mymodule. f90:
mycprogram. c:
$ ./a.out
22 33.3
44 55.5
```

3.7 I/O 大小端的转换

在其它系统中通过 Fortran I/O 库产生的数据文件可能与神威编译器编译出代码所希望或生成的文件格式不一致，本节将介绍 Fortran 编译器如何与这些系统进行文件交换。

使用 assign 命令或 ASSIGN()子程序，可以在进行 I/O 时实现大小端的转换。

3.7.1 assign 命令

assign 命令用于改变或显示某个 Fortran 文件或 unit 的 I/O 处理指示。assign 命令允许多个处理指示关联与一个 unit 或文件名。可以用于实现 I/O 的数组转换。assign 命令使用环境变量 FILENV 所指定的文件来存储处理指示。该文件也用于 Fortran I/O 库运行时加载指示。

例如：

$ FILENV=.assign

$ export FILENV

$ assign -N mips u:15

该命令指示 Fortran I/O 库把所有写到 15 unit 的数字数据采用 MIPS 格式数据。其效果是在读文件时文件中的内容从大端格式（MIPS）转换成小端格式。写到文件中的数据从小端格式转换成大端格式。

相关的详细信息参看 assign(1)联机帮助。

3.7.2 使用通配符选项

assign 命令的通配符选项为：

assign -N mips p:%

在执行程序之前，运行下面的命令：

$ FILENV=.assign

$ export FILENV

$ assign -N mips p:%

例子中的设置适用于所有文件。

3.7.3 转换数据和记录头

如果需要把所有无格式 unit 的数字数据转成小端，且把记录头也转成小端，用法如下：

$ assign -F f77.mips -N mips g:su

$ assign -I -F f77.mips -N mips g:du

su 限定符匹配所有打开的顺序无格式文件。du 限定符匹配所有打开的直接无格式文件。-F 选项设置记录头格式为大端(F77.mips)。

3.7.4 ASSIGN 子程序

ASSIGN()子程序提供与 assign 命令对应的编程接口。其中子程序的第一个参数是与 assign 命令一致的字符串，第二个参数用于存储返回的错误值。例如：

integer :: err

call ASSIGN("assign -N mips u:15", err)

该例子的效果与 3.6.1.1 中的例子效果一样。

3.7.5 I/O 编译选项

在 I/O 兼容性方面增加了两个编译选项：-byteswapio 和-convert conversion。-byteswapio 选项指示在小端机器上进行无格式文件 I/O 时交换字节顺序，以大端格式读写（反之亦然）。-convert conversion 选项控制在小端机器上进行无格式文件 I/O 时交换字节顺序，以大端格式读写（反之亦然）。为了保证起作用，这两个选项必须在编译 Fortran 主程序时使用。

在运行程序时若设置了环境变量 FILENV，则 assign 命令的设置优先于编译选项。

-convert conversion 选项有三种参数：

- native – 不做转换，缺省
- big_endian – 大端文件
- little_endian – 小端文件

具体的详细信息参看 sw5f90 联机帮助。在运行程序时若设置了环境变量 FILENV，则 assign 命令的设置优先于编译选项。

3.8 Fortran 的调试和查错

使用-g 选项告诉神威编译器生成调试器（例如 GDB，Etnus’ TotalView®等）能使用的符号信息。符号信息格式采用 DWARF2.0，直接放在目标程序中。使用-g 编译的目标码可以使用调试器 GDB 或其它调试器进行调试。-g 选项一般自动把优化级别设为-O0，除非在命令行上显式指示优化级别。调试高优化级别的代码是允许的，但代码经过许多优化变换后会使得调试更加困难。

边界检查对调试非常有用。这也能用于调试动态分配内存。

如果你关心数字精度问题，7.7 节有更多的有关数字精度的内容。

有关调试和查错的更多信息参看第九章。有关调试器的更多内容请参看 GDB 的有关文档。

3.8.1 写常数区域错误

一些 Fortran 编译器为常数值分配可读-写的内存区域。Fortran 编译器为常数值分配只读内存区域。这两种策略都没有问题，但神威编译器的方法允许更大胆的进行常数传播。如果 Fortran 程序中出现改变常数变量值的情况， Fortran 的处理方式会导致运行时出错。典型的例子是子程序或函数的实参是常数值 0 或.FALSE.，但在子程序或函数中却试图给该参数赋新的值。

我们推荐尽可能改掉写常数区域的代码以避免此类情况。这种做法既可以继续使用其它 Fortran 编译器，也能在 Fortran 编译器上不会出错，而且可能性能更好。

如果你不能修改代码，我们提供一个-LANG:rw_const=on 选项改变编译器的策略，为常数值分配可读-写的内存区域。我们不它设成缺省的选项是为了不影响编译器的常量传播优化，避免使程序变慢。

你也可以尝试用-LANG:formal_deref_unsafe 选项。该选项告诉编译器在 Fortran 中推测一个废弃的形参是否安全。缺省是关闭的，这有利于性能。有关这两个选项的详细信息参见联机帮助。

3.8.2 Fortran 哑元别名引起运行时错

Fortran 的标准规定函数或子程序的参数之间不存在别名。例如下面的情况是不合法的：

```
program bar
...
call foo(c,c)
...
subroutine foo (a,b)
integer i
real a(100), b(100)
do i = 2, 100
a(i) = b(i) - b(i-1)
enddo
```

因为 a 和 b 是两个哑元，编译器在对该程序进行优化时依靠 a 和 b 不会存在内存重叠的假设，导致优化后的程序执行结果出错。

编程人员有时候会不遵守这个别名规则，结果程序在高优化级别时会结果出错。这类错误经常会被认为是编译器的问题，所以我们在编译器中增加了选项来检测此类错误。如果出错程序在-OPT:alias=no_parm 或-WOPT:fold=off 选项下结果正确，很可能你的程序破坏了 Fortran 的别名规则。

3.8.3 Fortran malloc 的调试

神威编译器包含调试 Fortran 中动态分配内存的特性。通过设置环境变量 SWC_FDEBUG_ALLOC，动态分配的内存在运行时初始化成以下值：

SWC_FDEBUG_ALLOC	值
ZERO	0
NaN	0xffa5a5a5 (4 字节 NaN)
NaN8	0xffa5a5a5fff5a5a5ll (8 字节 NaN)

例如，如果想把所有分配的内存初始化成 0，则在程序执行前设置 SWC_FDEBUG_ALLOC=ZERO。四字节和八字节 NaN 根据数组对界情况选择相应的值进行初始化（32 和 64 位）。

3.8.4 参数拷贝到临时变量

有时候，Fortran 标准需要把调用函数的实参拷贝到临时变量。通常出现在程序中使用了 Fortran 90 标准的数组特性，然后通过传统的 Fortran 77 风格的隐式接口调用子程序。特

别地，Fotran 77 风格的子程序认为所有数组都是内存连续的，但 Fortran 90 允许数组元素是分散或有跨步的。

拷贝需要花费时间，但连续的数组可能有利于提高 Cache 的局部性。程序是否会快或慢取决于哪个因素占上风以及程序的具体细节。因为隐含的拷贝会影响程序性能，编译器提供了报警选项。下面的例子举了众多情况中的两种出现拷贝的情况：一个是根据数组的有条件拷贝，另一个是无条件拷贝。

$ cat cico.f90

```
subroutine possible(a, n)
   implicit none
   integer :: n
   integer, dimension(n) :: a
   print '(a,25i5)', "possible:", a
end subroutine possible

program copier
   implicit none
   logical :: l
   integer :: i
   integer, target :: a(5,5) = reshape((/ (i, i=1,25) /), (/ 5, 5/))
   integer, pointer, dimension(:,:) :: p
   read *, l
   if (l) then
     p => a
   else
     p => a(1:5:2, 1:5:2)
   endif
! Because "possible" does not have an explicit interface, it
! expects a contiguous array. Therefore, the compiler generates a
! runtime test to check a "contiguous" bit belonging to the
! pointer "p", and if the target is not contiguous, the values are
! copied to a temporary array before the call and copied back
! after the call
   call possible(p, size(p))
! The compiler must always copy this sequence array to a
! temporary variable to make it contiguous
   call possible(a((/1,2,5/),(/2,3,5/)),size(a((/1,2,5/),(/2,3,5/))))
end program copier
```

```
$ sw5f90 -host -fullwarn -c cico.f90
call possible(p, size(p))
^
sw5f90-1438 sw5f90: CAUTION COPIER, File = cico.f90, Line = 26,
Column = 17
This argument produces a possible copy in and out to a temporary variable.
call possible(a((/1,2,5/),(/2,3,5/)), size(a((/1,2,5/),(/2,3,5/))))
^
sw5f90-1438 sw5f90: CAUTION COPIER, File = cico.f90, Line = 30,
Column = 18
This argument produces a copy in to a temporary variable.
sw5f90: SWCC Fortran Version 2.9.99 (f14) Thu Dec 7,
2006 06:03:17
sw5f90: 32 source lines
sw5f90: 0 Error(s), 0 Warning(s), 2 Other message(s), 0 ANSI(s)
sw5f90: "explain sw5f90-message number" gives more information
about each message
```

为了减少拷贝数量，同时受益于 Fortran 90 特性，可以使用 Fotran 90 风格的假定型和延迟型数组（数组的边界是"(:,:)"而不是"(2,3)"或"(n,m)"）声明数组哑元。这需要程序在所有子程序中使用显式的接口（interface 块）、模块 use 语句或 contains 嵌套函数等方式。根据 Fortran 标准的观点，上述每一种方式都为编译器提供了显式的接口。

备注：冗余的接口是不允许的：不能在子程序中提供 interface 块的同时又通过 use 语句引入相同的接口。编译器有时候也会把不连续的数组拷贝到临时变量，虽然这不是标准必须的，但根据启发式信息认为有利于改善 Cache 的性能。可以使用命令行选项"-LANG:copyinout=off"关闭这类拷贝。

3.8.2 保留的文件单元

Fortran 编译器保留 Fortran 文件单元 5，6 和 0。

3.9 源代码的兼容性

本节介绍针对在其它编译器上开发的源代码在上的兼容性。不同的编译器处理各种类型是有区别的，这样会引起一些问题。

Fortran KIND 属性是指定精度和类型长度的一种方法。现代 Fortran 使用 KIND 来声明类型。这个机制非常灵活，但有一个缺点。下面介绍一种值得推荐和方便移植的 KIND 使用方法：

```
integer :: dp_kind = kind(0.0d0)
```

实际上，一些用户喜欢在他们的程序中把具体的值写死：

integer :: dp_kind = 8

这是一种不利于移植的习惯，因为一些编译器对双精度浮点值使用不同的 KIND 值。

大多数编译器使用字节数作为 KIND 的值。例如浮点数，32 位浮点数意味着 KIND=4，64 位浮点数意味着 KIND=8。神威编译器也采用这个约定。

对我们和用户来说，这种程序放到 GNU Fortran，g77 上是不兼容和不易移植的。g77 使用 KIND=1 表示单精度浮点（32 位）和 KIND=2 表示双精度浮点（64 位）。而整型方面，g77 使用 KIND=3 表示 1 字节，KIND=5 表示 2 字节，KIND=1 表示 4 字节，KIND=2 表示 8 字节。我们正在研究怎样提供一个选项兼容这些难以移植的 g77 程序。如果你觉得这是一个问题，最好的解决办法是改变你的程序，查询实际的 KIND 值而不是把值写死。

如果你使用-i8 或-r8，更多内容请参看 3.3.1。

3.10 库的兼容性

本节介绍 Fortran 与 C 或其它 Fortran 编译器库的兼容性。

与其它 Fortran 编译器生成的目标码进行链接是很复杂的问题。Fortran 90 或 95 编译器实现模块和数组的方法相差很大，因此很难把两个或更多编译器 生成的目标码链接在一起。对于 Fortran 77，运行时库中的 I/O 和内部函数也不一样，但还是有可能把两者的运行时库都链接进来生成一个可执行程序。

我们跟 g77 编译的目标码进行了链接实验。该代码不保证任何时候都正确。有可能我们库中的函数名与 g77 库中的函数名相同但调用约定不一样。除了 g77 外，我们没有测试过与其它编译器生成的代码进行链接。

3.10.1 换名

在把 Fortran 源程序编译成目标码的过程中，对于函数、子程序和公用块的名字换成内部表示。例如，Fortran 子程序名为 foo，在目标码文件中被转换成"foo_"。这样做的目的是避免与其它库中相似函数冲突，使得跟 C、C++和 Fortran 混合编程更加容易。

换名可以保证 Fortran 程序或库的函数、子程序和公用块名字与其它编程语言对应的库中名字不冲突。例如，Fortran 库中包含一个函数"access"，它的功能跟标准 C 库中的 access 函数一样，但它有四个参数，而标准 C 库中只有两个参数。如果你与标准 C 库一起链接就会引起符号名冲突。对 Fortran 的符号换名避免这类情况。

缺省的，我们在进行换名的时候采用 GNU g77 和 libf2c 库一样的换名规则。对于没有下划线的名字在后面追加一个下划线，若名字中有下划线，则追加两个下划线。下面举些例子进行说明：

molecule -> molecule_

run_check -> run_check__

energy_ -> energy___

在 sw5f90 编译器中该换名规则可以使用-fno-second-underscore 和-fno-underscoring 选项进行改变。

PGI Fortran 和 Intel Fortran 缺省的策略对应我们的-fno-second-underscore 选项。

公用块名也会换名。我们的空公用块名跟 g77 一样(_BLNK__)。PGI 编译器也使用同样的名字，Intel 的编译器使用_BLANK__。

3.10.2 ABI 兼容性

神威编译器支持 SW 26010 应用二进制接口(ABI)。其它编译器不一定完全遵循这个接口。特别指出，g77 没有遵循 SW 26010 ABI 有关在返回值类型为 COMPLEX 的函数中直接通过结果寄存器返回结果。有关 g77 更详细的信息可以通过“info g77”查看-ff2c 选项。

所以在链接二进制库时例如 BLAS 库或 CXML 库（Compaq 扩展数学库）时要考虑类似的问题。一些库如 FFTW 和 MPICH 没有任何函数返回 COMPLEX，所以没有这方面的问题。

怎样链接 g77 编译的返回值为 COMPLEX 的函数请参看 3.8.2。

跟大多数 Fortran 编译器一样，我们在把字符串传递给子程序时使用字符串地址，并在参数列表的最后增加一个整型的长度参数。

3.11 移植 Fortran 代码

移植 Fortran 代码时首先可以使用下述选项来修正一些问题。

-r8 -i8 可以把缺省的 REAL 和 INTEGER 类型长度从 4 字节升级为 8 字节。这对移植整型和实型缺省为 8 字节长的 Cray 代码很有用。但要小心外部库函数的类型是否一致。

前面的一些章节包含了对移植 Fortran 代码有帮助的内容：

- 3.7 节有怎样移植包含 KIND 的代码的内容，有时候这会造成 Fortran 代码移植的障碍
- 3.9 节包含源代码兼容性的内容
- 3.10 节包含库兼容性的内容

3.12 Fortran 程序栈大小

Fortran 编译器经常在栈中分配数据空间。一些环境中对处理器栈空间的大小有限制，从而导致使用了大量栈空间的 Fortran 程序运行后很快出错。Fortran 编译器运行时环境检测栈大小限制，并会在 Fortran 程序开始运行之前为 Fortran 进程增加栈空间。

缺省情况下，它会自动根据系统的物理内存数量调整栈空间限制，至少为每个 CPU 保留 128M 空间。例如，如果系统有 4 个 CPU，1G 主存，则 Fortran 运行时库把栈空间最多增加到 1G-(128M*4)=512M。

如果在执行 Fortran 程序前设置环境变量 SWC_STACK_VERBOSE，Fortran 运行库会告诉你它设置的栈大小限制。

你也可以使用环境变量控制 Fortran 运行时库设置的栈空间大小限制。如果该环境变量设的是空字符串，Fortran 运行库就不会修改栈大小的限制

另外，该变量必须包含一个数字。如果除了数字之外没有任何文字，则处理成字节数。

如果跟着字母"k"或"K"，则数字的单位是 Kb（1024 字节）。如果跟着字母"m"或"M"，则数字的单位是 Mb（1024K 字节）。如果跟着字母"g"或"G"，则数字的单位是 Gb（1024M 字节）。如果跟着符号"%"，则处理成系统物理内存的百分比。如果数字是负值，则代表为系统预留的空间量，就是，总的系统物理内存减去它作为栈空间大小的限制值。如果文字后面跟着"/cpu"，意思是每个 cpu 的数值，最终是把它乘以系统中的 CPU 数。这对在多机系统中并发执行多个进程时有用。指定（包括隐式地或显式地）的值是每个进程的内存量。

下面列举一些栈空间大小限制的实例（在一个 4CPU 和 1G 内存的系统上）：

环境变量值	栈空间大小限制
100000	100000 字节
820K	820K (839680 字节)
-0.25g	保留 0.25G 即 0.75G
128M/cpu	每个 CPU 128M 即总共 512M
-10M/cpu	每个 CPU 保留 10M(总共保留 40Ml)即 0.96G

如果 Fortran 运行时库尝试修改栈大小限制失败，将会输出一些报警信息，但不会退出。

第四章 C/C++编译器

C 和 C++编译器遵守以下标准以及扩展：

C 编译器：

- 遵守 ISO/IEC 9899:1990，C 语言编程标准
- 支持 C 语言扩展，该扩展在“Using GCC: The GNU Compiler Collection Reference Manual”中描述
- 参照本文档第 4.4 节，该节列出了目前还不支持的扩展
- 在编译器运行的平台上，神威编译器与 GNU C 编译器（gcc）的 C 应用程序二进制接口是一致的
- 支持大部分 gcc 命令行选项
- 生成的代码符合 SW 26010 ABI

C++ 编译器:

- 遵守 ISO/IEC 14882：1998（E），C++语言编程标准
- 支持 C++语言扩展，该扩展在“Using GCC: The GNU Compiler Collection Reference Manual”中描述
- 参照本文档第 4.4 节，该节列出了目前还不支持的扩展
- 在编译器运行的平台上，神威编译器与 GNU C++（g++）编译器的应用程序二进制接口是一致的
- 支持大部分 g++命令行选项
- 生成的代码符合 SW 26010 ABI

下列命令用来启动 SWCC C 和 C++编译器

- sw5cc — 启动 C 编译器
- sw5CC — 启动 C++编译器

编译使用的命令行选项与 GCC 族是一致的，详细的选项请参考 4.1 节。

4.1 使用 C/C++编译器

如果你正使用 GCC 编译器，那么将很快熟悉神威编译器的命令行。在现有的使用 GCC 的 Makefile 中只要改变启动编译器为神威编译器并重新编译就可以了。

启动编译器的方法虽然与 GCC 一致，但是控制编译过程的选项却不尽相同。我们在基础编译选项的设置上尽量与 GCC 兼容，尽管如此，神威编译器仍然提供了 GCC 所不具备的优化特性，比如 IPA 和 LNO。

一般说来，GCC 的优化是做为一个整体，而神威编译器把优化分成各个不同的阶段来进行，每个不同的阶段做为单独的一个优化组；这样，优化选项将会冠以一个组名，如-IPA，-LNO，-OPT，-CG 等，这方面我们的编译器与 GCC 差别是很大的，更多信息请参考联机帮助。

编译器默认的优化级别是 2 级，也就相当于加-O2 优化选项。下面两个命令的作用是一

样的：

$ sw5cc -host hello.c

$ sw5cc –O2 hello.c

关于优化级别的更详细的讨论请参考第 7.1 节。

如果使用-Ofast 或-ipa，不但编译，而且链接时也必须有此选项，如

$ sw5cc -host -c -Ofast warpengine.cc

$ sw5cc –host -c -Ofast wormhole.cc

$ sw5cc –hybrid -o ftl -Ofast warpengine.o wormhole.o

-ipa 和-Ofast 选项请参考 7.2 节

4.1.1 兼容 GCC2.96 和 GCC4.2 的 C 和 C++前端

sw5cc/sw5CC 支持 GCC4.2.1 前端

sw5cc.old/sw5CC.old 支持 GCC2.96 前端

4.1.2 对不同空间属性的支持

C 程序除了函数内临时变量外，与 Fortran 程序类似，也分下列变量类型：

__thread_local：该属性表示它所修饰的数据对象存放在 LDM 空间；

__thread：该属性表示它所修饰的数据对象存放在用户私有连续区；

__thread_group：该属性表示它所修饰的数据对象存放在用户共享交叉区；

不加修饰的普通数据对象存放在用户共享连续区；

__thread_local_fix：该属性表示它所修饰的数据对象从程序开始到结束一直常驻在 LDM 空间中。

__thread_local_kernel（kernel_name）：该属性表示它所修饰的数据对象存放在 LDM 的重用空间内，该 kernel 空间名称为 kernel_name，由程序员任意指定。

不同的 kernel 空间内的数据共享相同的局存地址空间，举例如下：

__thread_local_kernel(“xx”) int a;

__thread_local_kernel(“xx”) long b[100];

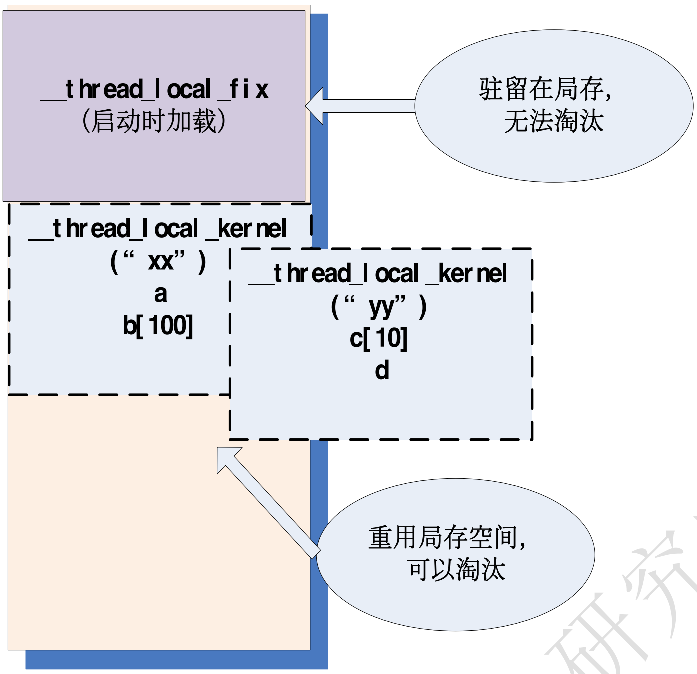
……

__thread_local_kernel(“yy”) int c[10];

__thread_local_kernel(“yy”) float d;

……

这里局存变量 a、b[100]与 c[10]、d 共享同一段局存地址空间。如下图：



需要注意的是，__thread_local_kernel 类型的变量不允许进行初始化赋值，如下所示的初始化编译器会报错退出：

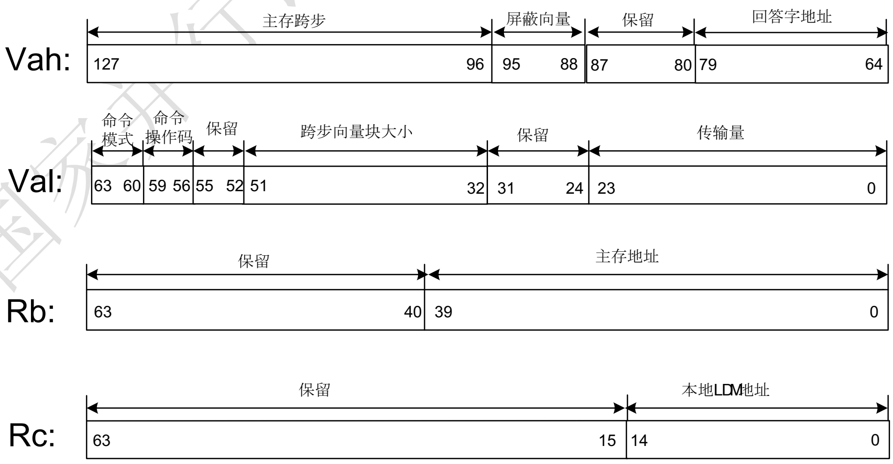
__thread_local_kernel(“example”) int a = 123；//错误用法。

可执行代码段的扩展同 Fortran。

4.1.3 DMA instrinsic

DMA intrinsic 的主要功能是在运算核心的 LDM 和主存之间进行数据的传输，DMA 只能由运算核心发起。

DMA 的峰值性能在主存地址为 128B 对界，且传输量为 128B 倍数的时候。



dma va, rb, rc	发起 DMA 请求和 DMA 栏栅操作命令

DMA 为异步操作，流水线 0 发送 DMA 命令到运算核心中的通道缓冲后即结束，可继续运行，通道缓冲满的情况下流水线 0 暂时停止运行。

数据 DMA 中使用的 DMA 的命令模式包括：

- 单运算核心模式
- 广播模式
- 行模式
- 广播行模式
- 行集合模式

用户在程序中使用编译器提供的 DMA 接口时，必须包括头文件：

#include "dma.h"

扩充的关键字：

- dma 描述符——**dma_desc**：dma_desc 使用 128 位的向量数据类型；
- dma 模式——**DMA_MODE**

```
enum DMA_MODE {
    PE_MODE，
    BCAST_MODE，
    ROW_MODE，
    BROW_MODE，
    RANK_MODE，
};
```

- dma 操作码——**DMA_OP**

```
enum DMA_OP {
    DMA_PUT，
    DMA_GET，
    DMA_PUT_P，
    DMA_GET_P，
    DMA_BARRIER=5，
};
```

扩充的 intrinsic：

- dma_set_size
- dma_get_size
- dma_set_reply
- dma_get_reply
- dma_set_op
- dma_get_op
- dma_set_mode
- dma_get_mode
- dma_set_mask
- dma_get_mask
- dma_set_bsize
- dma_get_bsize

- dma_set_sleng
- dma_get_sleng
- dma_desc_init
- dma
- dma_wait
- dma_barrier

4.1.3.1 dma_set_size(dma_desc *dma_d, int size)

函数名

dma_set_size —— 设置 dma 描述符的数据传输量属性

功能描述

设置 dma 描述符的数据传输量属性。

例如：dma_set_size(dma_d，256);

返回值

dma_desc

备注

数据传输量

4.1.3.2 dma_set_reply(dma_desc *dma_d, int *reply)

函数名

dma_set_reply —— 设置 dma 描述符的回答字属性

返回值

dma_desc

功能说明

设置 dma 描述符的回答字属性。

例如：

__thread_local reply；

dma_set_reply(dma_d, &reply);

备注

4.1.3.3 dma_set_op(dma_desc *dma_d, dma_op op)

函数名

dma_set_op —— 设置 dma 描述符的 dma 操作属性

对应指令

无

返回值

dma_desc

功能说明

设置 dma 描述符的 dma 操作属性。

例如：

dma_set_op(dma_d, dma_put);

备注

4.1.3.4 dma_set_mode(dma_desc*dma_d, dma_mode mode)

函数名

dma_set_mode —— 设置 dma 描述符的 dma 模式属性

对应指令

无

返回值

dma_desc

功能说明

设置 dma 描述符的 dma 模式属性。

例如：

dma_set_op(dma_d, pe_mode);

备注

4.1.3.5 dma_set_mask(dma_desc *dma_d, int mask)

函数名

dma_set_mask ——设置 dma 描述符的屏蔽码属性

对应指令

无

返回值

dma_desc

功能说明

设置 dma 描述符的屏蔽码属性。

例如：

dma_set_mask(dma_d, 1);

备注

只用于广播和广播行模式

4.1.3.6 dma_set_bsize(dma_desc *dma_d, int bsize)

函数名

dma_set_bsize ——设置 dma 描述符的跨步向量块大小属性

对应指令

无

返回值

dma_desc

功能说明

设置 dma 描述符的跨步向量块大小属性。

例如：

dma_set_bsize(dma_d, 0x32);

备注

只用于跨步模式以及行集合模式；

广播模式下的对界（TODO）

行集合模式下 bsize 是传输到每个运算核心上的块大小。

4.1.3.7 dma_set_stepsize(dma_desc *dma_d, int length)

函数名

dma_set_stepsize ——设置 dma 描述符的主存跨步长度属性

对应指令

无

返回值

功能说明

设置 dma 描述符的主存跨步长度属性。

例如：

dma_set_stepsize(dma_d, 0x128);

备注

只用于跨步模式以及行集合模式；

广播模式下的对界（TODO）

4.1.3.8 dma (dma_desc dma_d, long mem, long ldm)

函数名

dma

对应指令

dma

功能说明

根据参数指定的 dma 描述符、主存起始地址和局存起始地址，发起 dma；

返回值：

TODO

备注

4.1.3.9 dma_wait(int *reply, int count)

函数名

dma_wait

功能说明

根据参数指定的回答字地址等待，直到回答字的值等于 count

返回值：

TODO

备注

4.1.3.10 dma_barrier

函数名

dma_barrier

对应指令

dma

功能说明

发起本运算核心的 dma 栏栅

返回值：

TODO

备注

4.1.4 其他扩展

对 SIMD、intrinsic 的扩展同 Fortran 编译器。

4.2 编译和运行时特性

4.2.1 预处理

在编译器前端处理之前，源程序需要经过预处理器的处理。预处理器将在源程序中寻找特定的指示字，从而排除源码中不需要编译的部分，包括进特定的头文件并扩展宏。

若指定-noccp 选项，则 C 和 C++文件不经过预处理器的处理。

4.2.1.1 预定义宏

神威编译器为预处理定义了一些宏，如下所示：

__linux

__linux__

linux

__unix

__unix__

unix

__gnu_linux__

__GNUC__ 4

__GNUC_MINOR__ 1

__GNUC_PATCHLEVEL__ 2

__SWCC__ "2.0"

__SWCC__ 2

__SWCC_MINOR__ 0

__SWCC_PATCHLEVEL__ 0

__LP64__

_LP64

备注: __ GNU*和__SWCC*的值取决于相应的编译器版本号，各发行版本可能会不同。

如果源程序是 FORTRAN，但仍使用 cpp 进行预处理，那么下列 Fortran 宏被使用：

_LANGUAGE_FORTRAN 1

LANGUAGE_FORTRAN 1

_LANGUAGE_FORTRAN90 1

LANGUAGE_FORTRAN90 1

备注：当使用的优化级别为-O1 或更高的时候，编译器将使用_ _OPTIMIZE_ _

在编译时可以使用-dD -keep 选项，在结果.i 文件中可以看到预定义 cpp 宏。

4.2.2 编译指示

4.2.2.1 编译指示 pack

该编译指示的语法是：#pragma pack（n）

该编译指示指定结构体内的成员必须 n 字节对界，如果其自然对界大于 n 的话。

4.2.2.2 使用编译指示改变优化

通过用户程序中的指示可以改变编译优化选项。

在 C 和 C++中，指示的形式如下：

#pragma options <list-of-options>

在一个函数范围内，该指示可以任意使用，每一条的作用范围从函数入口到它所在的位置。<list-of-options>可包含用空格隔开的任意选项。编译流程转到下一个函数时，优化选项转为正常。

该指示的限制：

- 对未处理的选项不会有报警或错误信息
- 该指示只对后端的优化选项起作用，只有影响后端优化的才会被处理

- 此外，该指示并不影响后端各个优化过程的触发，如指示-O0 并不会使全局优化失效，但进入特定的优化阶段后，必须遵守指示字所规定的优化
- 除了-O 优化级别选项，只有属于-LNO，-OPT 和-WOPT 这些指令组的选项才会起作用

4.2.2.3 使用编译指示进行代码布局优化

该编译指示应用于 C 和 C++，用户可以为编译器提供一个指示，表明程序中某一个 if 语句的执行频率更高，从而指导编译器进行分支优化。该指示的形式是：

#pragma frequency_hint <hint>

<hint>可以取下列值：

- never：该分支不会或极少被执行
- init：该分支只在初始化时执行
- frequent：该分支执行频率很高

包含上述指示的 if 语句在编译时将按照所指示的内容进行优化

4.2.3 混合编程

如果一个应用程序既有 Fortran 代码，又有 C 代码，并且主函数是 C 和 C++的，那么使用 sw5cc 或 sw5CC 来进行链接，而不是使用 sw5f90；并且在链接时必须手动添加 Fortran 运行时的库。

更多细节请参考 3.5。如果用 sw5cc 或 sw5f90 进行链接，但如果目标文件是 sw5CC 生成的，则应该增加选项-lstdc++。

4.2.4 链接

需要注意，使用 sw5cc 进行链接时，如果程序调用 libm 函数，那么必须加-lm 选项，第二遍反馈编译必须显式的增加-lm 选项。

4.3 C/C++程序的调试

加-g 选项可以使编译器产生调试信息，其格式为 DWARF2.0，用-g 编译的目标码可以用 GDB 或其他调试器进行调试。

-g 选项将自动使优化级别设置为-O0，除非用户显示指定其它的优化级别。调试更高级别的优化代码也是可以的，但是因为优化带来的代码变换将使调试变得比较困难。

4.4 不支持的 GCC 扩展

C/C++编译器支持大部分 GCC4.2.1 扩展，但是也有例外，如下所示：

对 C：

- 不支持嵌套函数

- 不支持整数复数类型，尽管编译器支持浮点复数，但是它不支持整数复数类型，如_Complexint
- 许多__buildin 函数
- 跳到基本块外的 goto。编译器支持取得当前函数内标号的地址，并通过间接跳转到该标号
- java 异常
- java_interface 属性
- init_priority 属性

第五章 加速线程库

SW 26010 加速线程库(athread 库)是针对主从加速编程模型所设计的程序加速库，其目的是为了用户能够方便、快捷地对核组内的线程进行控制和调度，从而更好地发挥核组内多运算核心执行加速性能。

一般来说并发线程的应用程序无需考虑可用处理器的数量，但是在 SW 26010 架构下，每个线程绑定一个运算核心资源，所以在使用 athread 库的时候，需要考虑运算核心资源的分配状况。在调用使用线程库前，必须显式检查核组内运算核心的可用资源状态。

5.1 运算控制核心加速线程库

运算控制核心加速线程库主要是提供运算控制核心程序使用的 athread 接口，主要用于控制线程的创建回收、线程调度控制、中断异常管理、异步掩码支持等一系列操作。用户可以通过调用以下接口实现相关功能。

5.1.1 初始化线程库

函数名：

athread_init

函数说明：

完成加速线程库的初始化，在使用任何线程库接口前必须调用该接口。

参数说明：

无

返回值：

成功：返回 0，表示加速线程初始化成功；

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_init()将失败，并返回相应的值：

- -EINVAL：初始化失败。

附加说明：

无

5.1.2 创建单线程

函数名：

athread_create

函数说明：

在当前进程中添加新的受控线程。线程执行的起始任务由函数 fpc 指定；函数 fpc 的参数由 arg 提供。

参数说明：

int id：线程绑定 ID 号；

start_routine fpc：函数指针；

void * arg：函数 f 的参数起始地址。

返回值：

成功：返回 0。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_create()将失败，并返回相应的值。

- -EAGIAN：超出了系统限制，创建的线程太多。
- -EINVAL：ID 号非法(负数，或者超过最大线程数)；或者 ID 号已经被占用，被占资源没有释放。

附加说明：

线程绑定 ID 号后，线程位置跟 ID 号无关，如果需要知道线程在核组中所占用的运算核心资源信息，可以调用相关接口得到该类信息。

5.1.3 等待单线程终止

函数名：

athread_wait

函数说明：

显式阻塞调用该线程，直到指定的线程终止。指定的线程必须位于当前的进程中。

参数说明：

int id：等待线程 ID 号

返回值：

成功：返回 0；

失败：其它任何返回值都表示出现了错

- -ESRCH：没有找到指定线程 ID 对应的线程。
- -EINVAL：ID 号非法。

附加说明：

无

5.1.4 关闭单线程流水线

函数名：

athread_end

函数说明：

在确定线程所占运算核心无相关作业后，停滞运算核心流水线，关闭运算核心，该运算核心在本进程中将无法再次使用。

参数说明：

int id：指定线程 ID 号

返回值：

成功：返回 0，线程所占用的运算核心资源被成功关闭。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_end()将失败，并返回相应的值。

- -ESRCH：没有找到给定线程 ID 对应的线程。
- -EINVAL：线程运算核心流水线终止失败，ID 对应线程仍然在作业运行中。

附加说明：

必须保证运算核心无任何用户作业才能使用该函数。

举例说明：

```
typedef void * (*start_routine)(void *);
int fun(void * arg){
	retrun (int)arg[0]+(int)arg[1];
}
main(){
	int tid1,tid2;
	int arg1[2],arg2[2];
	//线程库初始化
	athread_init();
	//创建绑定线程，线程 ID 号为 0
	arg1[0]=1;
	arg1[1]=2;
	athread_create(0, fun,arg1);
	//创建绑定线程，线程 ID 号为 1
	arg2[0]=3;
	arg2[1]=4;
	athread_create(1, fun,arg2);
	//等待线程结束
	athread_wait(0);
	athread_wait(1);
	//关闭线程所占运算核心流水线，该运算核心无法在本进程中继续使用
	athread_end(0);
	athread_end(1);
}
```

5.1.5 创建线程组

函数名：

athread_spawn

函数说明：

在当前进程中添加新的受控线程组。线程执行的任务由函数 fpc 指定；函数 fpc 的参数由 arg 提供。

参数说明：

start_routine fpc：函数指针；

void * arg：函数 f 的参数起始地址。

返回值：

成功：返回 0。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_spawn() 将失败，并返回相应的值。

- -EINVAL：线程组创建失败。

附加说明：

调用时 athread_spawn 接口时，启动核组中的所有可用运算核心资源。如果需要知道线程组在核组中所占用的运算核心资源信息，可以调用相关接口得到该类信息。

5.1.6 等待线程组终止

函数名：

athread_join

函数说明：

显式阻塞等待该线程组，直到指定的线程组终止。指定的线程组必须位于当前的进程中。

参数说明：

无

返回值：

成功：返回 0；

失败：其它任何返回值都表示出现了错

- -EINVAL：核组内无线程组运行。

附加说明：

无

5.1.7 关闭线程组流水线

函数名：

athread_halt

函数说明：

在确定线程组所占所有运算核心无相关作业后，停滞运算核心组流水线，关闭运算核心组，该运算核心组在本进程无法再次使用。

参数说明：

无

返回值：

成功：返回 0，线程所占用的运算核心资源被成功关闭。

失败：其它任何返回值都表示出现错误，如果监测到以下任一情况，athread_halt()将失败，并返回相应的值。

- -EINVAL：无法正常关闭运算核心资源。

附加说明：

必须保证运算核心组无任何用户作业才能使用该函数。

5.1.8 创建带屏蔽码线程组

函数名：

athread_spawn_group

函数说明：

在当前进程中添加新的带屏蔽码的受控线程组。线程执行的任务由函数 fpc 指定；函数 fpc 的参数由 arg 提供。

参数说明：

unsigned long gmask:运算核心屏蔽码；

start_routine fpc：函数指针；

void * arg：函数 f 的参数起始地址。

返回值：

成功：返回 0。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_spawn_group()将失败，并返回相应的值。

-EINVAL：线程组创建失败。

附加说明：

调用时 athread_spawn_group 接口时，启动屏蔽码中所有运算核心资源。如果需要知道线程组在核组中所占用的运算核心资源信息，可以调用相关接口得到该类信息。

5.1.9 等待带屏蔽码线程组终止

函数名：

athread_join_group

函数说明：

显式阻塞等待带屏蔽码线程组任务结束，直到指定的线程组终止后退出该接口。指定的线程组必须位于当前的进程中。

参数说明：

unsigned long gmask:运算核心屏蔽码

返回值：

成功：返回 0；

失败：其它任何返回值都表示出错

-EINVAL：核组内无线程组运行。

附加说明：

无

5.1.10 创建抢占动态调度运算控制核心线程

函数名：

athread_task

函数说明：

创建抢占动态调度运算控制核心线程。执行的线程任务由函数 fpc 指定；arg 由用户指定线程任务总数，该值表示任务结束的判断标志。

参数说明：

start_routine fpc：函数指针；

void * arg：任务总数

返回值：

成功：返回 0；

失败：其它任何返回值都表示出错

-EINVAL：线程组创建失败。

附加说明：

5.1.11 创建抢占动态调度运算核心线程组

函数名：

athread_spawn_task

函数说明：

创建抢占动态调度运算核心线程组。执行的线程任务由函数 fpc 指定；arg 由用户指定线程任务总数，该值表示任务结束的判断标志。

参数说明：

start_routine fpc：函数指针；

void * arg：任务总数

返回值：

成功：返回 0。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_spawn_task()将失败，并返回相应的值。

-EINVAL：线程组创建失败。

附加说明：

由于运算控制核心和运算核心运算控制核心和运算核心代码对应使用不同的编译器，运算核心编译的代码不能被运算控制核心调用，反之亦然，用户必须保证运算控制核心、

运算核心参与的抢占任务有各自对应的代码和编译支撑。

5.1.12 获取核组最大线程总数

函数名：

athread_get_max_threads

函数说明：

使用该接口得到当前进程的可用的所有核组线程资源总数。

参数说明：

无

返回值：

成功：返回核组内可用最大线程个数(1<=ret<=64)；

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_get_max_threads 将失败，并返回相应的值。

- -EINVAL：返回失败，出现异常。

附加说明：

5.1.13 设置并行区线程总数

函数名：

athread_set_num_threads

函数说明：

使用该接口设置当前进程下一个并行区启动的所有核组线程总数。

参数说明：

int num：线程总数

返回值：

成功：0；

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_set_num_threads 将失败，并返回相应的值。

- -EAGAIN：设置线程数目超过了可用线程资源，将返回该值。
- -EINVAL：并行区线程数设置失败，比如上个并行区任务未完成。

附加说明：

无

5.1.14 获取并行区线程总数

函数名：

athread_get_num_threads

函数说明：

使用该接口得到当前进程启动的所有核组线程总数。

参数说明：

无

返回值：

成功：返回当前核组内线程个数(1<=ret<=64)；

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_get_num_threads 将失败，并返回相应的值。

- -EINVAL：返回失败，出现异常。

附加说明：

无

5.1.15 强制线程结束

函数名：

athread_cancel

函数说明：

该函数用来终止指定线程，线程 ID 以及资源可以立即收回。

参数说明：

int id：指定退出线程号

返回值：

成功：返回 0；

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_cancel 将失败，并返回相应的值。

- -ESRCH：没有找到指定线程 ID 对应的线程。
- -EINVAL：ID 号非法。
- -EFAULT：ID 线程对应运算核心故障。

附加说明：

线程结束可以通过以下方法来终止执行：

- 从线程的第一个过程返回，即线程的启动例程。参考 athread_create。
- 调用 athread_cancel，提前退出。

5.1.16 中断管理

函数名：

athread_signal

函数说明：

接收中断，并指定中断处理函数。

参数说明：

int signo：中断信号

start_routine fpc：中断处理函数

返回值：

无

附加说明：

参考 athread_sigqueue()接口。

5.1.17 异常管理

函数名：

athread_expt_signal

函数说明：

挂载特定异常的处理函数。

参数说明：

int signo：异常信号

start_routine fpc：异常处理函数

返回值：

无

附加说明：

a：异常类型：

根据 SW 26010 运算核心异常类型，用户可以挂载以下 32 种异常处理信号：

```
enum Exception_Kind {
        UNALIGN = 0,     // 不对界异常
        ILLEGAL,         // 非法指令
        OVI,             // 整数溢出
        INE,             // 非精确结果
        UNF,             // 下溢
        OVF,             // 上溢
        DBZ,             // 除数为 0
        INV,             // 无效操作
        DNO,             // 非规格化数
        CLASSE1,         // 分类计数器溢出
        CLASSE2,         // 分类参数设置异常
        SELLDWE,         // 向量查表异常
        LDME,            // LDM 相关异常
        SDLBE,           //  SDLB 代换时发生越权或越界
        MABC,            // 访问主存的地址超出主存实际配置的容量
        MATNO,           // 主存访问目标核组不在位
        RAMAR,           // 运算核心收到访存异常错误响应
        IFLOWE,          // 指令流异常
        SBMDE1,          // SBMD 匹配异常和 SBMD 查询异常
```

```
        SBMDE2,         // 其他 SBMD 异常
        SYNE1,          // 运算核心发出的同步向量中不包含本运算核心
        SYNE2,          // 可降级同步不使能
        RCE,            // 通信异常
        DMAE1,          // DMA 产生 DMA 描述符静态检查异常
        DMAE2,          // DMA 产生 DMA 描述符静态检查警告
        DMAE3,          // 保留
        DMAE4,          // 行集合模式使用过程中有可能出现访问地址超出 LDM 容量
        DMAE5,          // 保留
        IOE1,           // 访问保留的 IO 空间地址
        IOE2,           //  IO 访问越权
        IOE3,           //  Io 不可访问
        OTHERE,         // 用户不关心的各类异常
        _SYS_NSIG,
};
```

b：挂载方式

athread_expt_signal（int exceptnum， void *handler）

c：异常处理函数

用户自定义异常处理函数原型为：

handler_t * handler(int signum, siginfo_t *sinfo, struct sigcontext *sigcontext)

第一个参数：signum 为异常信号

第二个参数：sinfo 的信息如下：

```
sinfo->si_signo =signum;    // 异常信号
sinfo->si_pid = peid;       // 出现异常的运算核心号
sinfo->si_uid = cgid;       // 出现异常的核组号
```

第三个参数：sigcontext 的前三个参数用来保留特定信息，如下：

```
sigcontext-> sc_onstack        // 运算核心异常 PC 是否精确标志
sigcontext->sc_pc              // 运算核心异常 PC
sigcontext-> sc_mask               // 运算核心数据流异常访存信息
```

sc_pc 如果为 0 表示没有异常 PC，不为 0 的情况下根据 sc_onstack 判断是否为精确 PC。

sc_onstack 为 0 表示没有精确异常 pc，此时 sc_pc 记录的是发生异常时的非精确 PC；sc_onstack 为 1 表示有精确异常 pc，此时 sc_pc 记录的是发生异常时的精确 PC；

sc_mask 项只对数据流异常 DFLOWE 有效，为 4b`0001 表示 LD 主存，为 4b`0010 表示 ST 主存，为 4b`0100 表示 DMA_GET，为 4b`1000 表示 DMA_PUT。

sigcontext 其它各个域的内容由 os 确定并原样传给用户处理程序。

发生异常时，线程库会报出第一个捕获的异常，这个异常可以是核组异常也可以是运算核心异常。如果用户没有挂载相应该异常的处理函数，那么线程库会打印异常相关的所有信息，halt 发生异常的运算核心，并退出程序；如果用户挂载了该异常的处理函数，则进入用户定义的异常处理函数。

（For 并行 C 库：

athread_expt_signal_for_ccc(void *handler)

并行 C 库使用该函数对所有异常事件进行挂载，发生异常时，将会进入并行 C 库定义的异常函数进行处理，handler 函数的参数如下：

void handle(int sig, siginfo_t *sinfo, struct sigcontext *sigcontext)

上述三个参数的内容由操作系统确定并原样传给并行 C 库异常处理函数

）

5.1.18 运算控制核心取运算核心局存地址

宏名：

IO_addr(elment, penum, cgnum)

宏说明：

运算控制核心可以通过 IO 访问局存，该宏返回运算核心 LDM 变量 element 的 IO 地址。

参数说明：

element：运算核心程序内的局存私有变量__thread_local elment，运算控制核心使用 IO_addr 接口前必须加 extern __thread 声明成外部变量

penum：核号

cgnum：核组号

返回值：

返回 IO 地址

附加说明：

IO_addr 宏不会判断 penum 和 cgnum 的合法性，由用户保证。

举例说明：

运算核心程序 slave.c:

__thread_local char ch='b';

__thread_local long para[10] __attribute__((__aligned__(128)))= {0x123, 0x234, 0x345, 0x456};

运算控制核心程序 main.c：

extern long __thread para[];

extern char __thread ch;

unsigned long LDM_addr;

LDM_addr = IO_addr(para[0], 23， 0);

这样，LDM_addr 等于第 0 号核组第 23 号运算核心上 para[0]的 IO 地址

5.1.19 运算控制核心访问运算核心局存

宏名：

h2ldm(elment, penum, cgnum)

宏说明：

运算控制核心可以通过 IO 访问局存，该宏直接在运算控制核心对运算核心 LDM 变量 element 进行 IO 存取操作

参数说明：

element：运算核心程序内的局存私有变量__thread_local elment，运算控制核心使用 IO_addr 接口前必须加 extern __thread 声明成外部变量

penum：核号

cgnum：核组号

返回值：

typeof(element)

附加说明：

h2ldm 宏不会判断 penum 和 cgnum 的合法性，由用户保证。

举例说明：

运算核心程序 slave.c:

```
__thread_local char ch='b';
__thread_local long para[10] __attribute__((__aligned__(128)))= {0x123, 0x234, 0x345, 0x456};
```

运算控制核心程序 main.c：

```
extern long __thread para[];
extern char __thread ch;
unsigned long   LDM_addr;
h2ldm(para[0], 23, 0) = 0xaaa;
h2ldm(ch, 5， 1) = 'z';
```

运算控制核心对 0 号核组第 23 号运算核心上局存变量 para[0]赋值；运算控制核心对 1 号核组第 5 号运算核心上的局存变量 ch 赋值。

备注：

线程 create 和 spawn 接口只带一个参数接口。而运算控制核心访问运算核心局存接口可以用作运算控制核心对核组加速区进行批量的参数赋值。比如：运算核心程序专门申请一块局存空间用作存放参数，运算控制核心在启动不同的加速区之前可以对这块参数区进行赋值；如果每个运算核心上分配相同的参数，通常最优的做法是只对某一个运算核心进行参数赋值，然后运算核心通过寄存器通信将参数广播到核组上所有的运算核心。

5.1.20 线程空闲状态查询

函数名：

unsigned long athread_idle（）

函数说明：

使用该函数可获得当前运算核心线程组是否处于空闲状态。

参数说明：

无

返回值：

如果运算核心组处于空闲状态则返回当前可用的运算核心位图；否则返回 0。

附加说明：

这里的空闲状态是指等待任务的状态，也即：线程组未执行用户 spawn 的运算核心加速线程或线程组已经完成用户 spawn 的运算核心加速线程。

举例说明：

```
unsigned long idleslave;
if(idleslave=athread_idle())
{
        用运算核心优化(spawn)；
        ……
}
```

5.1.21 核组状态初始化

函数名：

void athread_task_info()

函数说明：

查询当任务所在 CPU 中，参与任务执行的核组个数以及当前所在核组的核组号。分别通过查看内部库变量__cgid/__cgnum 来得到相应的值。

参数说明：

无

返回值：

无

附加说明：

5.1.22 核组间同步

函数名：

void athread_master_sync ()

函数说明：

核组间同步。

参数说明：

无

返回值：

无

附加说明：

调用该接口前，必须先调用 athread_task_info 来初始化__cgid 和__cgnum，通过内部变量来通知参与同步的核组的个数。

该接口会占有交叉段前 64B（0x6000000000~0x6000000000+0x40）的数据作为锁空间，用户在使用该接口时需要注意避开该空间，防止不可预测的错误出现。

5.2 运算核心加速线程库

运算核心加速线程库主要是提供运算核心程序的 athread 接口，主要用于运算核心线程的线程识别、中断发送等一系列操作。用户可以通过调用以下接口实现相关功能。

5.2.1 获得线程逻辑标识号

函数名：

athread_get_id

函数说明：

使用该函数获得本地单线程的逻辑标识号(即 ID 号)。该接口运算控制核心和运算核心运算控制核心和运算核心通用。

参数说明：

int core：指定查询物理核号，-1 默认本地运算核心(运算核心接口有效)。

返回值：

成功：线程逻辑 ID 号(0<=ret<=63)。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_get_id 将失败，并返回相应的值。

- -EINVAL：返回失败，出现异常。

附加说明：

只要当线程是被用 athread_create 创建单线程接口启动的线程才会有绑定的标识符。

举例说明：

```
int myid;
myid = athread_get_id(-1);
```

5.2.2 获得线程物理运算核心号

函数名：

athread_get_core

函数说明：

使用该接口获得对应线程的物理运算核心号。该接口运算控制核心和运算核心运算控制核心和运算核心通用。

参数说明：

int id：指定查询线程号，-1 默认本地线程(运算核心接口有效)。

返回值：

成功：线程所占运算核心物理号(0<=ret<=63)。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_get_core 将失败，并返回相应的值。

- -EINVAL：返回失败，出现异常。

附加说明：

返回值低 6 位有效，低 3 位为列号，高 3 位为行号，分别表示核组 8*8 阵列。

举例说明：

int mycore；

mycore = athread_get_core(-1);

5.2.3 发送异步中断给运算控制核心

函数名：

athread_sigqueue

函数说明：

运算核心给运算控制核心发送携带消息异步中断，中断发出后，运算核心并不等待运算控制核心是否接收到中断并完成中断处理函数，而是继续执行后续的运算核心代码。

参数说明：

int pid ：主进程，用 0 表示

int signo：中断号

const union sigval value：中断消息

返回值：

成功：返回 0；

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_sigqueue ()将失败，并返回相应的值。

- -EINVAL：signo 无效，或者发送消息失败。

附加说明：

```
enum {
        SW3_SIGSYSC = 1,          /*1: host side syscall agent */
        SW3_SIGHALT,              /*2: slave core group halt */
        SW3_SIGEND,               /*3: slave core end */
        SW3_SIGERR,               /*4: slave error report */
        ……
        SW3_SIGMAX = 63,          /*63: core signal max number */
```

};

举例说明：

如何正确挂载一个中断处理函数：

假设运算控制核心和运算核心运算控制核心和运算核心约定当运算核心发出一个SW3_SIGMAX-1 的信号的时候，则运算控制核心调用中断处理函数 x_handler。

运算核心程序：

```
a=1234567890;
fun()
{
//将中断信号 SW3_SIGMAX-1
//以及中断携带信息&a 发给主进程。
athread_sigqueue(0，SW3_SIGMAX-1，&a)
}
```

运算控制核心程序：

```
x_handler(int sig, siginfo_t *sinfo, struct sigcontext *sigcontext)
{
        union sigval sv;
        printf("x_handler begin !\n");
        //sv==运算核心中断携带信息&a
        sv = sinfo->si_value;
}
main()
{
        athread_signal(SW3_SIGMAX-1，x_handler); //和 signal 函数用法一致
}
```

5.2.4 发送同步中断给运算控制核心

函数名：

athread_sigsend

函数说明：

运算核心给运算控制核心发送携带消息同步中断，中断发出后，运算核心等待运算控制核心接收到中断直到完成中断处理函数后才能继续执行后序运算核心代码。

参数说明：

int pid ：主进程，用 0 表示

int signo：中断号

const union sigval value：中断消息

返回值：

成功：返回 0；

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_sigsend ()将失败，并返回相应的值。

- -EINVAL：signo 无效，或者发送消息失败。

附加说明：

```
enum {
        SW3_SIGSYSC = 1,        /*1: host side syscall agent */
        SW3_SIGHALT,            /*2: slave core group halt */
        SW3_SIGEND,             /*3: slave core end */
        SW3_SIGERR,             /*4: slave error report */
        ……
        SW3_SIGMAX = 63,        /*63: core signal max number */
};
```

举例说明：

参考 athread_sigqueue 说明。

5.2.5 数据接收 GET

函数名：

athread_get

函数说明：

运算核心局存 LDM 接收主存 MEM 数据，进行主存 MEM 到运算核心 LDM 的数据 get 操作，将 MEM 的数据 get 到 LDM 指定位置。传输模式由 mode 指定，如果是在广播模式和广播行模式下，要进行屏蔽寄存器配置；在其它模式下，mask 值无效。

参数说明：

extern int athread_get(dma_mode mode, void *src, void *dest, int len, void *reply, char mask,int stride,int bsize);

dma_mode mode：DMA 传输命令模式；

void *src：DMA 传输主存源地址；

void *dest：DMA 传输本地局存目标地址；

int len：DMA 传输数据量，以字节为单位；

void *reply：DMA 传输回答字地址，必须为局存地址，地址 4B 对界；

char mask：DMA 传输广播有效向量，有效粒度为核组中一行，某位为 1 表示对应的行传输有效，作用于广播模式和广播行模式；

int stride：主存跨步，以字节为单位；

int bsize：行集合模式下，必须配置，用于指示在每个运算核心上的数据粒度大小；其它模式下，在 DMA 跨步传输时有效，表示 DMA 传输的跨步向量块大小，以字节为单位。

返回值：

成功：返回 0。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_get 将失败，并返回相应的值。

- -EINVAL：返回失败，出现异常。

附加说明：

```
typedef enum {
        PE_MODE,
        BCAST_MODE,
        ROW_MODE,
        BROW_MODE,
        RANK_MODE
} dma_mode;
```

举例说明：

```
__thread_local long   ldm_a[64];
__thread long   mem_b[64];
__thread_local   int reply=0;
athread_get(PE_MODE,mem_a,ldm_a,64*8,&reply,0,0,0);
```

//以单运算核心模式，运算核心私有连续段以 mem_a 为基地址取 64*8B 数据放入运算核心局存以 ldm_a 为基地址的连续 64*8B 的空间中，其中广播向量、主存跨步、向量块大小均为 0。

5.2.6 数据发送 PUT

函数名：

athread_put

函数说明：

运算核心局存 LDM 往主存 MEM 发送数据，进行运算核心 LDM 到主存 MEM 的数据 put 操作，将 LDM 的数据 put 到 MEM 指定的位置。传输模式由 mode 指定，不支持广播模式和广播行模式。

参数说明：

int athread_put(dma_mode mode, void *src, void *dest, int len, void *reply,int stride,int bsize)

dma_mode mode：DMA 传输命令模式；

void *src：DMA 传输局存源地址；

void *dest：DMA 传输主存目的地址；

int len：DMA 传输数据量，以字节为单位；

void *reply：DMA 传输回答字地址，必须为局存地址，地址 4B 对界；

int stride：主存跨步，以字节为单位；

int bsize：行集合模式下，必须配置，用于指示在每个运算核心上的数据粒度大小；其它模式下，在 DMA 跨步传输时有效，表示 DMA 传输的跨步向量块大小，以字节为

单位。

返回值：

成功：返回 0。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_put 将失败，并返回相应的值。

- -EINVAL：返回失败，出现异常。

附加说明：

DMA 传输命令模式参考 athread_get 接口附加说明。

举例说明：

无

5.2.7 物理地址数据接收

函数名：

athread_get_p

函数说明：

物理地址数据接收，除源主存地址为物理地址外，其它使用方式与 athread_get 一致

5.2.8 物理地址数据发送

函数名：

athread_put_p

函数说明：

物理地址数据发送，除目的主存地址为物理地址外，其它使用方式与 athread_put 一致

5.2.9 DMA 栏栅

函数名：

athread_dma_barrier

函数说明：

发起 dma 栏栅，对应 dma 命令操作码为 barrier_group

参数说明：

无参数

5.2.10 核组内同步

函数名：

athread_syn

函数说明：

核组内运算核心同步控制。

参数说明：

scope scp：同步范围控制

int mask：同步屏蔽码

返回值：

成功：返回 0。

失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_syn 将失败，并返回相应的值。

- -EINVAL：返回失败，出现异常。

附加说明：

```
typedef enum {
        ROW_SCOPE,    //行同步，低 8 位有效
        COL_SCOPE,    //列同步，低 8 位有效
        ARRAY_SCOPE,  //全核组同步，低 16 位有效，其中 16 位中低 8 位为列同步屏蔽码、高 8 位为行同步屏蔽码
}scope;
```

5.3 运算核心局存分配

运算核心 LDM 支持精简快速的动态分配，具体接口如下：

5.3.1 get_allocatable_size

函数原型：

get_allocatable_size(void)

函数说明：

得到当前可动态分配的 LDM 空间大小，也即局存动态分配运行时库管理的 LDM 空间大小

参数说明：

无

返回值：

5.3.2 ldm_malloc

函数原型：

ldm_malloc(size_t size)

函数说明：

局存空间申请。

参数说明：

size：以字节为单位的空间大小

返回值：

成功返回分配 LDM 空间的起始地址；

失败返回 NULL；

5.3.3 ldm_free

函数原型：

ldm_free(void *addr, size_t size)

函数说明：

局存空间释放，释放空间起始地址为 addr。

参数说明：

void * addr ：必须是前面某一次 ldm_malloc 返回的分配成功的局存地址；

size_t size：显示指定释放空间的大小，一般必须与前面某一次 ldm_malloc 的大小相匹配

返回值：

无

第六章 优化快速入门

本章将给出使用神威编译器进行代码优化的一些基本方法。

下面讲述的方法是一些性能优化的基本技巧，它们简单易于实践，并且通常会有较好的优化效果，使用这些基本的性能选项，用户将不需要修改 Makefile 文件，也不用担心优化后结果不对。这些选项相关的更详细的讨论可以参考下一章或联机帮助。

6.1 基本优化

基本的-O 选项等同于-O2，这是优化代码最先考虑用到的选项，

-O2

然后

-O3

接着

-O3 -OPT:Ofast

有关-O 选项和-OPT:Ofast 选项的更详细信息，参见 7.1 节

6.2 过程间分析优化（IPA）

IPA 是过程间分析优化的缩写，编译器采用过程间分析与优化将编译优化的作用范围扩展到了整个程序，对所有过程进行全局的分析和优化，消除了因过程调用而产生的编译优化屏障。用户通过-ipa 选项调用该优化模块。

IPA 可以和别的优化选项结合使用，比如-O3 -ipa 或者-O2 -ipa，这种组合选项达到的性能比单独使用-O3 或-O2 要好，-ipa 在编译和链接的时候需要同时用，更详细的使用方法参见 7.3 节

6.3 反馈式编译优化（FDO）

反馈式编译优化技术是一种特殊的优化，先执行一遍甚至多遍程序，在运行时收集反馈信息，然后这些信息被用到新的执行过程中。详细见 7.6 节

6.4 更激进的优化

SWCC 编译器还针对各种特殊情况提供了一系列选项，前面提到的那些选项，主要是针对提高程序的性能，这一节将主要介绍除了-O2 或-O3 之外首先可选的选项。有些选项是需要知道算法的行为和程序编程风格的，否则会影响到程序的正确性。

备注：就象所有的高性能优化编译器一样，SWCC 编译器有一系列的优化，一些优化可以保证程序的输出与优化前一致，但一些优化会稍微改变程序的行为。第一类的优化我们称之为安全的，相对应第二类是非安全的。这些优化的更多信息参见 7.7 节。

-OPT:Olimit=0 是一个常规的安全选项，但用该选项编译时可能要花更多的时间或占用

更多的内存。这个选项告诉编译器，不管要编译的文件有多大，都要用指定的级别优化。

-fno-math-errno：这个选项旁路了数学函数里对 ERRNO 的设置，如果程序运行时检查浮点错时，不需要遵循 IEEE 异常处理的话，则可以使用该选项来提高性能。

-OPT:roundoff=2：如果允许优化前后浮点溢出和舍入的结果不同的话，用该选项，编译器可以对代码进行更深度的优化。详细参见 7.7.4 节

-OPT:roundoff=3：一般针对 fortran 课题，对运算顺序进行调整以寻求更多的优化机会。对部分课题优化效果明显，但可能造成精度损失。

-OPT:div_split=ON：该选项允许把 x/y 转换为 x*(recip(y))，这可能要在 损失一点浮点计算的精度，详细参见 7.7.4 节

-OPT:alias：该选项的设置允许编译器对程序进行更激进的优化。-OPT:alias=typed 假设用户程序是遵循 ANSI/ISO C 标准进行编写的，该标准规定不同类型的两个指针不能指向同一个内存地址。-OPT:alias=restrict：该选项允许编译器假设指针所指的都是不同的、不重叠的对象。如果用户使用了这些选项，但程序违背了前面的这些假设，也许运行结果就会不正确。详细参见 6.7.4 节。

-OPT:fast_math_library：针对数学库的编译器优化。

有几个简写选项可以代替以上所述。比如 -OPT:Ofast 等价于 -OPT:roundoff =3:Olimit=0:div_split=ON:alias = typed:.fast_math_library=ON 而 -Ofast 则等价于 -O3 -ipa -OPT:Ofast 。当用这些简写选项的时候，用户要知道与之对应的等价的组合选项分别的含义，这样才能确保该选项的效果。

还有更多的选项可以帮助优化程序的性能，这些选项可以通过本手册的其他章节或查看联机帮助。

6.5 编译选项的推荐使用方法

作为一个通常的方法，我们推荐您从-O2 开始调整性能，然后-O3, 再用-O3 -OPT:Ofast ，最后使用-Ofast。

用-O3 -OPT:Ofast 和-Ofast，您需要关注下结果是否精确。

-OPT:Ofast 选项用到都是最大化性能的的优化，虽然这些优化通常都是安全的，但由于计算的重排可能影响到浮点的精度，这主要是由于以下选项的打开导致的：

-OPT:ro=3:Olimit=0:div_split=ON:alias=typed

如果采用-O3 -OPT:Ofast 在计算精度上出现了问题，那么你试着使用下面的组合：

-O3 -OPT:Ofast:ro=1

-O3 -OPT:Ofast:div_split=OFF

上面选项中的’ro’是 roundoff 的缩写。

-Ofast 等价于-O3 -ipa -OPT:Ofast -ffast-math 快速数学库选项跟-O3 -OPT:Ofast 一样要注意，可能会引起精度上的误差。

要想使用过程间分析又不想用“Ofast-类”优化的时候，可以使用

-O3 -ipa

-O2 -ipa

6.6 性能分析

除了上面的这些建议可以优化你的代码外，这里还提供了其他一些思路。在 2.11 节探讨了如何定位程序段的关键调优部位。主要是得到程序调用的时间图，通过 gprof 发现程序的热点。

第七章 优化选项

本章将更深入地介绍 SWCC 编译器中可用的主要选项组。

7.1 常用优化选项：-O 选项

-O2 选项是默认优化选项。

-O0 表示没有优化，这样做的目的是为了 debug 使用。Debug 中的-g 选项与该选项完全兼容。

备注：如果只使用-g 选项而没有使用-O0 选项，在没有指定其他优化选项时，编译器默认使用-O0 选项。

-O1 是最小程度的优化，在编译时间上与-O0 没有很大区别。该优化只局限于线性代码（基本块），例如窥孔优化和指令调度。-O1 级别的优化编译时间最短。

-O2 选项需要较长的编译时间，但是也带来了更加突出的优化效果：

- 针对内层循环：
  - 循环展开
  - 简单 if 转化
  - 循环相关优化
- 二遍指令调度
- 基于第一遍指令调度的全局寄存器分配
- 函数级的全局优化
  - 局部冗余代码删除
  - 无效 store 删除
  - 控制流优化
  - 强度减弱优化，循环结束判断替换优化。
  - 跨基本块的指令调度
- -O2 选项意味着-OPT:goto=on,该选项将 goto 结构转化为更高级的结构，例如 for 结构。
- -O2 选项设定-OPT:Olimit=6000.

-O3 选项打开更多的优化开关，使用户的程序得到最快的运行速度，很少减低程序效率。该选项除了包括所有-O1,-O2 的优化之外，还包含了如下选项：

- -LNO:opt=1,打开循环嵌套优化开关.
- -OPT 选项和下列选项的组合(可以参考-OPT 选项的节获取更多信息)
  - OPT:roundoff=1
  - OPT:IEEE_arith=2
  - OPT:Olimit=9000
  - OPT:reorg_common=1

7.2 组合优化分析(-CG, -IPA, -LNO -OPT, -WOPT)

这一组优化选项涵盖了大部分的优化动作,本节将介绍它们.

这一组选项以两种方式支持子选项:

- 将子选项分列在这些优化选项的后面.
- 在命令行将各个选项组合一一列出.

例如,下面两个命令行是等价的:

sw5cc -OPT:roundoff=2:alias=restrict wh.c

sw5cc -OPT:roundoff=2 -OPT:alias=restrict wh.c

有些子选项有打开和关闭两种状态,想采取打开状态,可以只指定该子选项名或将该子项名赋值为=1,=ON,或者=TRUE,关闭则赋值为=0,=OFF,=FAUSE,下列命令行是等价的：

sw5f90 -OPT:div_split:fast_complex=FALSE:IEEE_NaN_inf=OFF wh.F

sw5f90 -OPT:div_split=1:fast_complex=0:IEEE_NaN_inf=false wh.F

7.3 过程间分析

许多应用软件,它们的源代码通常由多个源文件组成,由 Makefile 指定的编译器编译每个源文件的过程,叫做编译单元,这种各个源文件单独编译的做法是比较传统的做法.在所有的源文件都生成.o 文件之后,由链接器生成最终的可执行代码.

这种单独编译文件的做法,有一个弊端,就是无法提供给编译器一个完成的源码信息.使得编译器遇到程序处理外部数据或者访问外部函数时,采取最稳妥的措施.如果在整个程序的优化过程中,编译器能搜集到全部文件中的信息,它就能采取更广泛的优化手段.因此,能针对整个程序进行优化肯定是高效的,另外,整体优化也比单独编译可以采取优化的种类也更多.

本节着重介绍能进行过程间优化的编译单元,以及在用户程序中如何调用-IPA 选项,还有过程间分析的各种优化措施,以及 IPA 如何提高后端的优化质量.各种提高性能的 IPA 选项.最后举一个 spec cpu2000 的例子来验证加入 IPA 选项对程序基准的提高.

注：ipa 只支持单运算控制核心，不支持运算核心和混合程序。

7.3.1 IPA 编译模式

过程间编译是一套使编译器整体编译程序的机制.这必须在进行一次链接过程之后才有可能实现.通常链接过程处理的对象是.o 文件,而这时候所有的优化动作已经完成.在 IPA 编译模式中,链接过程出现在整个编译过程较早的时候,也就是在大多数优化动作和代码生成进行之前,在这种情况下,链接的程序代码并不以目标文件格式出现.

IPA 编译模式(参照图例 7.1)的实现是利用了各个目标文件.但在用户级，只需要在编译和链接时加入-IPA 选项.因此,用户在使用 IPA 时可以不用重新生成 Makefile,为了实现这一点,我们必须产生一个新的.o 文件类型,我们称为 IPA 的.o 文件.程序在这些.o 文件中以 IR 的格式存储,而不是一般的.o 那样的二进制格式.编译器使用-IPA –c 选项来生成 IPA 的.o 文件,这些.o 只能被 ipa_linker 链接.在链接时使用-ipa 选项可以调用 ipa_linker,使用起来类似真正的链接过程,实际上,该链接过程包含了下面的几项工作:

- 调用 ipa_link
- 针对被链接的程序进行过程间分析优化
- 调用编译器后端进行优化并产生真正的目标代码
- 调用真正的链接器产生最终的可执行文件

在使用 IPA 编译时,用户会发现编译单个文件的速度非常快,因为该过程没有调用后端优化,而链接过程则时间相对较长,因为这时候才包含了整个的编译过程和优化过程.

7.3.2 过程间分析和优化

我们将针对链接 IR 结构时进行的分析叫做过程间分析.该分析过程可分为两个步骤：

- 在整个程序中搜集信息
- 优化程序代码以便提高运行速度

IPA 首先建立程序调用图，图的每个顶点代表一个函数模块。调用关系图反映出调用者和被调用者的相互关系。

一旦建立调用关系图，基于不同种类的启发式的内联机制，IPA 提供了一个函数调用列表,反映出被调用函数内联的位置。

基于调用图，IPA 为变量计算出“赋值-引用”信息。该信息描述了一个变量在一个函数中是被赋值还是被引用。

IPA 还会统计出程序变量的别名信息。一旦一个变量被分配空间，就潜在地被分配了一个指向该空间的指针。通过指针访问该空间意味着访问该变量。IPA 的别名分析跟踪这些信息以便在存在通过指针访问变量的情况存在时，尽可能减少该变量的改动，以便后面做更多的优化。

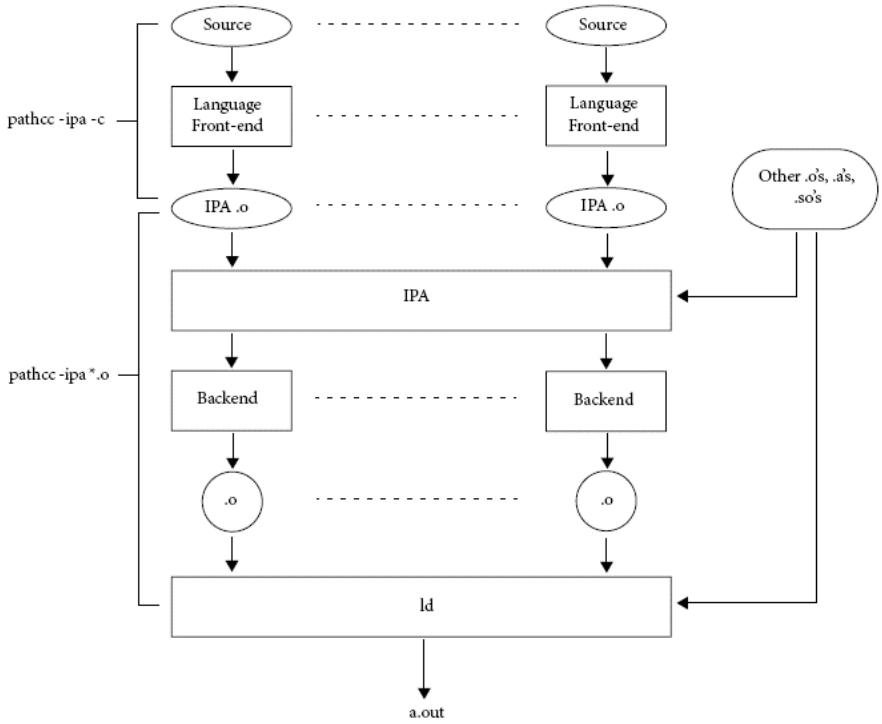
赋值－引用和别名信息由 IPA 搜集，只为 IPA 使用。这些信息也记录在程序表示中以便后端做更多的优化。

7.3.3 优化措施

IPA 提供的最重要的优化就是内联，它能将一个函数调用最终转化为函数实体。内联是 ipa 中用途最广泛的方法,因为优化效果对于所有用户程序中的函数都是可见的。除了省去一个函数调用的开销，内联还能使被内联的函数在更广泛的代码段中参与整体的后端优化。因此，内联作用于一个循环结构，使得编译器可以进行更加激进的循环变换。

内联需要进行仔细的收益分析，因为内联也有可能导致性能下降。因为程序代码长度的增加有可能加大指令 cache 的不命中。如果一个程序代码已经非常庞大，内联有可能导致可分配的寄存器全部用完，从而导致额外的访存开销使得效率降低，另外太多的内联会导致后面的优化处理变得很慢。

许多函数调用传递的参数是一个常数（包括变量地址）。将参数用常数替换可以对函数体的优化大有帮助，在这种情况下，函数的很多代码可以判断为无用而删除。函数克隆为一个函数产生很多不同的副本，被克隆的函数按照不同的参数形式来区分。它不会象内联那样



提高调用函数体的大小。但是会增加整个程序的大小。

如果 IPA 能确定所有函数调用都传递不变的参数，它就会对该参数进行常量传播。该优化和函数克隆效果相当，但并不增加程序的大小。常量传播也能应用于全局变量。如果一个全局变量在整个程序运行过程中都保持值不变，IPA 则将该全局变量替换成一个常数。

死代码删除能找到在整个程序中从未使用的全局变量并删除它们。这种情况一般出现在常量传播之后。

死函数删除能找到从未调用的函数并删除它们。该优化可以作为函数内联和克隆的附属。

公共块填充应用于 fortran 程序中。通常编译器不能改变用户在公共数据区中数据的分布情况，因为公共块是属于所有声明该公共块的子函数所共有的，而且子函数可以隶属于不同的编译单元。但是有了 IPA，所有的子函数都是可见的。采取公共块填充可以使数组对界，从而更有效率的进行访存和向量化，并有效地减少 cache 冲突。

公共块分割同样应用于 fortran 程序。这种将一个公共块分割成几个公共块的做法同样能减少 cache 冲突。

代码重排根据函数调用关系排列程序中的函数。这种做法可以减少指令 cache 中的脱靶率。

7.3.4 IPA 控制

尽管编译器试图对一个程序采取最好的优化措施，但一般不会达到最理想的效果。因此，编译器提供了许多编译选项以便用户可以进行性能调试，达到最优的效果。本节列举了与 IPA 相关的调试选项。

首先，有必要提到的是 IPA 是一个可以充分得益于反馈优化的优化阶段。在反馈编译阶段，生成的一个反馈数据文件包含了程序的运行特征，然后再提供给编译器。这使 IPA 能更

好的决定是否要克隆或者内联一个函数。并且 IPA 的代码重排能确保频繁调用的函数放在一起，从而使得优化更加有效。反馈优化通过选项-fb-create 和-fb-opt 打开。可以通过 6.6 节获取更多信息。

实际上，sw5cc 包含了两个内联器。使用哪一个由-IPA 选项控制。这是因为现在内联可以作为一个独立的语法分析动作而不必依托于 IPA。当没有-IPA 选项时使用的内联器是比较轻量级的，并且只能作用于一个编译单元。这种内联器不能做自动内联优化。它是严格按照 C++语法中的内联声明进行内联，或者 C 语言的内联关键字或者用户编译时加入-INLINE 选项。该选项可以是默认的。内联器控制内联的选项是：

- -inline 或-INLINE, 加入该选项可以在没有-IPA 的时候调用内联器。
- -INLINE:=off　　禁止使用内联该选项对内联器和 IPA 的内联都有效。
- -INLINE:all　允许所有可能的内联,由于因此而带来的代码量急剧增加，因此应该在程序代码量比较小的时候使用。
- -INLINE:list=ON 使得内联器打印内联动作。这个选项对用户查找哪个函数被内联，被内联的原因非常有帮助。因此如果用户希望内联一个函数或者不希望内联某个函数。基于该选项提供的内联列表中的信息可以很容易的找到实现的方法。
- -INLINE:must=name1[,name2,…]强制内联选项中声明的函数。
- -INLINE:never=name1[,name2,…]禁止内联选项中声明的函数。

当使用-IPA 选项时，IPA 将调用它自己的内联器而不调用独立的内联器。IPA 的内联器除了内联那些有内联声明的函数，还自动处理它认为应该内联的函数。代码量小的函数会被优先考虑内联。如果有反馈的 profile 做参考，调用频繁的函数会被优先考虑。否则会考虑循环中的函数，还会内联叶子函数（函数体中没有其他调用）。内联操作将所有符合标准的函数进行内联，可以使用下列选项控制：

- -IPA:inline=OFF　关掉 IPA 的内联，并且由于有-IPA 选项，独立的内联器也被关闭。IPA 默认是打开。
- -INLINE:none关掉 IPA 自动内联，但是仍然内联程序中由用户指定显式内联的函数。自动内联默认是打开的。
- IPA：specfile=filename 指定选项文件名。编译器打开该文件读取更多-IPA 和内联的指示信息。
- 下列操作可以调节内联的力度,过于激进的内联有可能降低程序效率：
- -OPT:Olimit=N　　指定一个限制量 N，N 是由函数包含的基本块数量计算而得，内联不得使函数超过该限制。默认是在-O2 情况下 N＝6000, -O3 情况下 N＝9000.N=0 表示没有限制。
- -IPA:space=N 指定内联可以在代码量增长率为 N%的范围内进行。默认是 100%，如果一个程序代码很小，可以增大 N 的值。
- -IPA:plimit =N　　一旦一个函数大小达到 N，则禁止该函数内联。N 的值基于基本块数量和函数中调用其他函数的数量。默认值是 2500。
- -IPA:small_pu=N　指定一个函数如果包含的基本块大小小于 N，则不受-IPA:plimit

的约束。默认值是 30。

- -IPA:callee_limit=n 指定一个函数的大小如果超过了 n 则不会自动内联。默认值是 500。
- -IPA:min hotness=N 该选项只在反馈式编译时有效。一个调用点在被内联之前，它的调用次数至少是 N 次。默认是 10.
- -INLINE:aggressive=ON 增加内联的强度，更多非叶子函数和非循环内函数都会被内联。默认是关闭的。

我们提到叶子函数是内联的最佳候选。这些函数不包括那些有可能受到后端优化约束的调用。为了扩大叶子函数得范围，IPA 提供了两个选项来拓展它的内联机制，这种机制是基于调用树的。这种做法是基于一个函数只调用叶子函数的情况，该函数在它调用的函数全部内联之后，也可以变成叶子函数。这些函数可以通过重复访问调用图来交替实现。在下面两个选项的描述中，一个函数如果在调用图中连接的叶子函数至多有 N 个边，就被称作深度为 N，一个叶子函数的深度为 0。

- -IPA:maxdepth=N IPA 内联所有在函数调用图中深度为 N 的符合空间大小限制的函数。
- -IPA:forcedepth=N IPA 内联所有在函数调用图中深度为 N 的函数，不受空间大小限制

7.3.5 克隆

有两个选项控制函数克隆：

- -IPA:multi_clone=N 指定从一个函数克隆出的最大数目。默认是 0，也就是克隆选项是默认关闭的。
- -IPA:node_bloat=N 指定克隆之后产生的程序总量超过原来的最大百分比。默认是 100。

7.3.6 其他 IPA 调节选项

下列选项虽然不涉及内联和克隆。但是对性能调节有帮助：

- -IPA:common_pad_size=N 指定公共块垫塞的大小为 N 字节。默认值是 0，即由编译器决定最佳的垫塞大小。
- -IPA:linear=ON 使数组引用线性化。当内联 fortran 子程序时，IPA 试图将形参映射到实参的结构上。该选项默认是关闭的，表示 IPA 如果无法映射则关闭内联。打开该选项则指示 IPA 仍然执行内联但是使数组引用线性化。这种线性化有可能导致性能问题，但是内联则会带来更多性能收益。
- -IPA:pu_reorder=ON 控制 IPA 的代码重排优化。N=0 表示关闭该选项。N＝1 表示基于不同的函数被调用的频繁程度，N=2 表示代码重排基于调用与被调用关系，默认 N＝0。
- -IPA:field_reorder=ON 该选项控制 IPA 的域重排优化，来减少数据 cache 不命中。

- -IPA:ctype=ON　　声明程序不在多线程环境下运行，对程序按照标准头文件 ctype.h 的定义构造优化界面。

下列选项用来关闭 IPA 中的优化。可以通过这些选项来研究 IPA 对性能的影响。

- -IPA:alias=OFF　　关闭 IPA 的别名和赋值－引用分析
- -IPA:addressing=OFF　　关闭 IPA 的地址分析，该分析属于别名分析。
- -IPA:cgi=OFF　　关闭全局变量的常数传播
- -IPA:cprop=OFF　　关闭参数的常数传播
- -IPA:dfe=OFF　　关闭无用函数删除
- -IPA:dve=OFF　　关闭无用变量删除
- -IPA:split=OFF　　关闭公共块分割

7.3.7 SPEC CPU2000 个案研究

IPA 优化在实际测试 SPEC CPU2000 基准测试程序中效果非常明显。我们的实验表明仅仅加上-IPA 和不加-IPA 选项测试 SPEC CPU2000，大多数 SPEC2000 课题都会有效果，其中 17 道课题性能提高了 1.3%~26.6%，6 道课题性能提高小于 0.5%，3 道浮点课题性能下降 1.2%~4.5%。

性能下降的课题说明 IPA 缺省设置不适合它，使用额外的 IPA 选项，这些性能损失会转为性能收益。最终 IPA 对 SPEC 测试课题大约能带来 6%的性能提升。

想了解有关 SPEC CPU2000 详细的测试数据请参看 SW 26010 处理器相关的测试报告。

7.3.8 调用 IPA

过程间分析可以通过-ipa，-IPA 调用，或者-Ofast 选项隐式地调用。IPA 可以和各种级别的优化同时使用，但是和-O3 编译收益最大。-Ofast 打开包括 IPA 在内的几种优化选项。

当使用-IPA 编译时，生成的.o 文件不是最终的.o 文件，而是一种中间表示文件，第二次编译时通过该文件中的信息生成可执行文件。

ipa_link 检验所有的程序文件是否使用同样的选项编，如果编译选项不同，IPA 会报警：

“Warning:文件使用了不同的选项编译”

例如，下列情况在编译 a.c 和 b.c 的时候会报警：

sw5cc –O2 –ipa –c a.c

sw5cc –O3 –ipa –c b.c

sw5cc –ipa　a.o b.o

用户可以传递相同的优化选项来消除报警。在上面的例子中可以同样都使用-O2 或者-O3 优化。

-IPA 选项可以和很多选项同时使用，但最典型的还是和-O 选项使用，例如：

-O3 –IPA 或者 -O2 –IPA 都可以在-O3 和-O2 的基础上额外提高性能,-IPA 选项必须在编译和链接阶段同时使用。

-IPA 选项的使用是非常便捷的，如果你的源文件比较少，可以这样编译：

sw5f90 -O3 –ipa main.f subs1.f subs2.f

如果你单独编译.o,那生成的这些.o 文件并不是二进制文件，它们包含了源文件的中间表示，在链接阶段才进行实际的编译过程。链接的命令行中也需要加入-IPA 选项。

例如，下面的例子就是单独编译文件，最后把它们链接在一起：

sw5f90 –c –O3 –ipa main.f

sw5f90 –c –O3 –ipa subs1.f

sw5f90 –c –O3 –ipa subs2.f

sw5f90 –O3 –ipa main.o subs1.o subs2.o

这里有一个限制，静态库文件中的.o 必须是同时使用-ipa 编译的.o，或者同时是未使用-ipa 编译的.o，不能混合。

引起注意的是，在未加 ipa 编译的过程中，多半的时间是在前面编译生成.o 的时候，链接过程很短。在加入 ipa 编译时，生成.o 的时间很短，但是链接时却要很长时间，整个的时间比不使用 ipa 要长。

当 ipa 链接阶段，可以使用并行处理单元并行处理该过程的几个重要的部分。可以使用-IPA:max_jobs 选项来实现，该选项描述如下：

-IPA:max_jobs＝N 该选项限制了同时编译 ipa 的进程数的最大值。该选项可以使用以下几个值：

0= 并行化规模等于 cpu 的总数，或者核心数目，或者超线程编译单元的数目。

1= 关闭并行化编译

>1= 明确指定并行化规模

7.3.9 IPA 的大小与正确性的缺陷

IPA 通常可以运行在不少于 10 万行的程序上，但是在这个版本中我们并不推荐基于大程序使用 IPA。

7.4 循环嵌套优化（LNO）

如果程序中有多重循环嵌套，可以尝试用一些循环嵌套优化选项，这些选项能对循环进行变换和改进。

SWCC 编译器一个很有价值的特性就是它实用的循环嵌套优化，可以使用-O3 选项来调用。这种特性可以在矩阵运算时通过-O3 优化带来将近 10～20 倍的收益。

- -LNO 选项可以控制系列循环嵌套优化子集：
- 循环分裂和融合
- 循环分块以便增大 cache 行重用性
- Cache 管理
- 根据 TLB 条目调整循环变换方式
- 预取

在本节着中讲述经常使用的比较有价值的 LNO 选项。

7.4.1 循环分裂和融合

有时候单个循环中的指令条数很少，需要将几个循环联合在一起以提高 cpu 利用率。这种做法叫做循环融合。

有时候循环内的指令太多，或者内层循环中处理很多数据条目，给寄存器分配带来很大压力，导致寄存器溢出到内存。在这种情况下，需要将循环分割成几个部分，这就是循环分裂。控制循环分裂和融合的选项如下：

- -LNO:fusion=n　　执行循环融合，n:0 是关闭，1 为保守，2 为激进。等于 2 意味着外层连续的循环都将被分裂，即便发现并不是各级的循环嵌套都适合分裂。该选项默认为 1.但是 level 2 可以提高很多代码的效率。
- -LNO:fission=n　　执行循环分裂，n:0 关闭，1 标准模式，2 在融合前先分裂。默认为 0. 但是 level 2 可以提高很多代码的效率。要注意同时使用上面两个选项的情况，因为融合的优先级比分裂高。当-LNO:fusion=[1，2]和-LNO:fission=[1,2]同时使用时，则只执行融合。
- -LNO:fusion_peeling_limit=n 该选项指定循环融合可以剥离的迭代数的最大值，默认值为 5,但是不能为负整数。在相邻的循环中，各自嵌套数不一致的情况下进行循环剥离，一些循环嵌套从循环体中复制出来。使嵌套数达到一致。

7.4.2 Cache 大小说明

SWCC 运算控制核心编译器是用于 SW 26010 运算控制核心的,因此它假定的二级 cache 大小是 4MB(每个核组个占用 1MB)。有一些可以使用的选项如下：

-LNO:cs1=n,cs2=n,cs3=n,cs4=n

这些选项指定 cache 大小，n 由 0 或者一个正整数加上一个字母 k,K,m 或者 M 来表示大小。K 表示千字节，M 表示兆字节。为零表示该级没有 cache。

Cs1 表示一级 cache。

Cs2 表示二级 cache。

Cs3 表示内存。

Cs4 表示存储体。

默认的各级 cache 的大小是依赖于体系结构的。使用-LIST:options=ON 可以看到编译时默认的 cache 大小。

-LNO:assoc1=n,assoc2=n,assoc3=n,assoc4=n

表示各级 cache 采取几路组联想。

7.4.3 Cache 分块，循环展开，循环交换，循环变换

对当前深度的嵌套循环进行循环交换和展开，以便于提高 cache 重用性，减少访存。LNO 默认是打开的，可以使用-LNO:blocking=off 来关闭。

- -LNO:blocking_size=n 指定编译器进行循环分块的分块大小,n 是一个表示迭代次数的正整数。

- -LNO:interchange 默认是打开的，但是设为 0 可以关闭循环嵌套优化中的循环交换。

LNO 的选项组控制外层循环展开，但是-OPT 选项组控制内层循环展开。下面列举控制循环展开的主要的 LNO 选项：

- -LNO:outer_unroll_max,ou_max=n 指定外层循环每层展开次数最多为 n 次。默认为 10。
- -LNO:ou_prod_max=n 表示给定循环嵌套的外层各级循环嵌套展开的数量总和的上限。N 是一个正整数，默认为 16。
- -LNO:outer_unroll,ou=n 指定如果条件允许，外层循环每层展开次数精确为 n 次。如果将某个循环的外层展开会引起问题，则不进行展开。

7.4.4 数据预取

LNO 选项组可以控制编译器产生各级预取。预取的强度由选项-LNO:prefetch=n 决定。n=1 表示默认级别，n=0 表示不预取，n=2 表示加大预取强度。

- -LNO:prefetch_ahead=n 设定比当前要装入内存的数据提前多少 cache 行进行预取。默认是 2 个 cache 行。
- -LNO:diverse_pf_ahead=n 调整预取距离。在 SW 26010 运算控制核心中，如果出现 cache 冲突，尽管 cache 是组联想的，但因为预取会引起 CPU 自陷，而在自陷后预取指令又被丢掉了，不但没有起到预取的作用，反而大量自陷导致性能明显下降。针对这一问题的一种办法是不同的预取指令预取距离不一样，避免预取指令 cache 冲突。缺省值为 0，预取距离不变

7.4.5 向量化

向量化是一种同时处理多个数据块的优化技术。例如编译器可以把循环中的数学计算函数 sin()变为 vsin()函数，可以达到两倍于原来的速度。

调用数学库中例如 sin(),cosin()的并行函数版本可以通过选项-LNO:vintr=0|1|2 实现。0 将关闭向量数学库的调用，1 为默认值，当-LNO:vintr=2 时编译器将并行化所有数学库函数。要注意 vintr=2 并不安全，因为有些函数的向量化处理有可能导致精度差距。

-LNO:vintr_verbose=ON 打印出内部函数是否进行向量化的信息。

除了向量化数学库，SW 26010 系统中也了专门提供了自动向量化工具，该工具能把串行程序翻译成 SWCC 可以接受的含 SIMD 扩展类型和内部函数的程序，从而达到发挥 SIMD 扩展指令的优势，具体自动 SIMD 识别器的内容请参看相关的文档。

7.5 代码生成（CG）

代码生成阶段的选项组可以控制指令的生成，以及指令级的优化。

- -CG:gcm=OFF 关闭全局代码重排，默认是打开的。
- -CG:load_exe=n n=0 表示关闭这种规类优化。默认 n=1，表示该优化只在 load 结果仅仅被引用一次的情况进行，如果 load 的结果会被引用超过 n 次，则不进行该

优化。N 为一个非负整数。我们发现 load_exe=2 或 0 偶尔会对性能有提高。针对 64 位 ABI 和 FORTRAN 的值为 2，否则默认值为 1.

7.6 激进的优化

跟其它现代编译器一样，神威编译器包含了大量的优化。一些优化可以保证程序的输出与优化前一致，但一些优化会稍微改变程序的行为。前一种优化认为是安全的，后一种优化是不安全的。一般的，-O1, -O2, -O3 选项通常只会进行安全的优化。但是一些不安全的优化会有很好的优化效果，也能产生足够精度的结果。一些不安全的优化根据实际的使用情况也可能是安全的。我们建议在编译程序时先使用安全的优化选项，然后逐步增加不安全的优化选项，并检查结果是否正确与否以及不安全优化带来的效果。

下面列举了一些典型的不安全优化。

7.6.1 别名分析

C 和 Fortran 都经常会出现两个变量占用相同的内存空间。例如，在 C 中，两个指针可能指向同一个地址，这样通过一个指针可以改变另一指针指向的变量的值。标准 C 中对一些别名做出了限制，许多实际的程序可能违反这些规则，所以编译器提供-OPT:alias 选项来控制别名处理的方式。详细情况参看 8.7.4。

别名通过如下方式隐式定义和使用数据：

- 通过指针访问
- 存储区域部分重叠(例如 C 的联合)
- 程序中使用非局部变量
- 异常处理程序

编译器通常会假设别名是存在的。然后编译器通过别名分析去确定没有别名，这样就可以进行后续的优化。特定的 C 和 C++语言规则可以带来某种级别的别名分析。Fortran 有额外的规则，从而有可能得到更准确的别名信息：子程序参数没有别名，函数调用的副作用只限于全局变量和实参。

对于 C 或 C++，编码风格能帮助编译器获得更准确的假设。使用类型限定词如 const, restrict 或 volatile 能帮助编译器。更进一步，如果你根据程序的具体情况使用一些假设，则可以进行更多的优化。下面是一些可以使用的别名模型，以字符串升序和潜在危险性进行排序，对应的提供给编译器的信息有：

-OPT:alias=any 缺省级别，表示任何两个访存都有可能别名。

-OPT:alias=typed 意味着启用 ANSI 规则，基本类型不同的变量之间不存在别名，该选项在-Ofast 下自动启用。

-OPT:alias=unnamed 假设指针不会指向有名变量。

-OPT:alias=restrict 告诉编译器所有指针都是严格指针，指向的变量不会重叠。这会让编译器进行许多优化，就好象程序是以 Fortran 写的一样。在源程序中也可以使用 C ’restrict’ 关键字声明严格指针。

-OPT:alias=disjoint 说的是两个指针表达式假设成不同的，不会重叠的变量。

要在你的程序中使用相反的断言，在相应名字前加“no_”。例如，-OPT:alias=no_restrict 意思是不同的指针可能会重叠。

对于 Fortran 程序中有额外的-OPT:alias 选项：

-OPT:alias=cray_pointer 认为 Cray 指针指向的对象与其它变量之间不会重叠。该选项还告诉编译器 Cray 指针指向的对象在调用外部子程序前把值存回内存，并在下一个访问把值读出内存。而且在子程序的 END 或 RETURN 语句之前完成 store。

OPT:alias=parm 保证 Fortran 参数不会与其它变量存在别名。这是缺省情况。no_parm 断言参数在程序中会存在别名。

7.6.2 影响精度的优化

对数学表达式进行重排和改变浮点运算的顺序会轻微改变结果，例如：

A = 2. * X

B = 4. * Y

C = 2. * (X + 2. * Y)

聪明的编译器会注意到 C = A + B。但操作的顺序是不同的，所以得到 C 的值也会有点差别。这种特殊的变换可以通过-OPT:roundoff 选项控制，但还有其它几个影响精度的不安全选项。

属于这一类的选项有：

控制 IEEE 行为的选项如-OPT:roundoff=N 和-OPT:IEEE_arithmetic=N。下面是相关的其它选项：

-OPT:div_split=(ON |OFF)该选项打开或关闭把型如 X/Y 的表达式转换成 X* (1/Y)的优化。倒数运算在精度上比直接进行除法差，但可能更快。

-OPT:recip=(ON |OFF)该选项允许把 1/X 转换成计算机中的倒数指令。精度低于除法，但更快。

这些选项与性能密切相关。更多相关信息请参看联机帮助。

7.6.3 IEEE 754 的兼容性

通过选项可以控制 IEEE 754 的兼容级别。放松编译器的兼容级别能更自由地进行代码变换从而达到更高性能。下一节将讨论相关的一些选项。

7.6.3.1 兼容级别

有时候编译器使用遵从 IEEE 754 标准的操作也能获得很好的性能，同时也能获得满意的数据精度。控制 ANSI/IEEE 754-1985 浮点舍入和溢出行为兼容级别的选项为：

-OPT:IEEE_arithmetic=N （N=1，2 或 3）

-OPT:IEEE_arithmetic

=1 严格遵从标准

=2 允许使用产生精确结果的任何操作。对于非精确结果的情况会影响精度。例如，X*0可以换成 0， X/X 可以换成 1，虽然当 X 为+inf, -inf 或 NaN 时结果是不准确的。

=3 意味着允许任何数学意义上合法的变换。例如，把 x/y 换成 x*（recip（y））。更多有关 IEEE 算术级别的内容请参看本手册其他相关描述。

7.6.3.2 舍入精度

使用-OPT:roundoff=选项控制舍入误差的范围。编译器提供：

0 没有舍入误差

1 允许有限的舍入误差

2 允许符合结合律的表达式变换带来的舍入误差

3 没有允许任何的舍入误差

在-O0，-O1 和-O2 下缺省的舍入级别为 0。-O3 时缺省的舍入级别为 1。

为了更好的理解各种舍入级别，下面列出不同舍入级别下启用的其它-OPT:选项。

-OPT:roundoff=1 隐含：

- -OPT:fast_exp=ON 打开指数函数优化为乘、平方根操作特定的编译时常数。
- -OPT:fast_trunc 指示内联 NINT, ANINT, AINT 和 AMOD 等 Fortran 内部函数。

-OPT:roundoff=2 打开下面的选项：

- -OPT:fold_reassociate 允许对浮点运算进行符合结合律的优化。

-OPT:roundoff=3 打开下述选项：

- -OPT:fast_complex 当打开时，复数绝对值和复数除法使用快速算法，但当操作数（除法操作的被除数）的绝对值大于最大可表示的浮点数的平方根时会溢出。
- -OPT:fast_nint 使用硬件特性实现单精度和双精度版本的 NINT 和 ANINT。

7.6.4 其它不安全的优化

一些高级优化可能使用特殊指令如 CMOV(条件传送)，从而改变了程序的行为。下面的程序在 if()语句的保护下对变量进行赋值：

```
if (a .eq. 1) then
  a=3
endif
```

在这个例子中，SW 26010 中最快的代码是通过始终对 a 进行赋值避免分支转移；如果条件失败，它把已存在的值赋给 a，否则把 3 赋给 a。如果 a 是一个等于 1 的只读的常数，这个优化可能会导致程序段违例，虽然程序稍显奇怪，但是完全正确的。

7.6.5 算术精度的假定

下表列出的是不同优化级别时有关数字精度的假定。

-OPT:选项名	-O0	-O1	-O2	-O3	-Ofast	备注
div_split	off	off	off	off	on	如果 IEEE_a=3

fast_complex	off	off	off	off	off	如果 roundoff=3
fast_exp	off	off	off	on	on	如果 roundoff>=1
fast_nint	off	off	off	off	off	如果 roundoff=3
fast_sqrt	off	off	off	off	off	
fast_trunc	off	off	off	on	on	如果 roundoff>=1
fold_reassociate	off	off	off	off	on	如果 roundoff>=2
fold_unsafe_relops	on	on	on	on	on	
fold_unsigned_relops	off	off	off	off	on	
IEEE_arithmetic	off	off	off	off	off	
IEEE_NaN_inf	off	off	off	off	off	
recip	off	off	off	off	on	如果 roundoff>=2
roundoff	0	0	0	1	2	
fast_math	off	off	off	off	off	如果 roundoff>=2
rsqrt	0	0	0	0	1	1 如果 roundoff>=2

例如，如果你在-O3 下使用-OPT:IEEE_arithmetic，该选项缺省设置是 IEEE_arithmetic=2。

7.7 编译优化指示

针对 SW 26010 结构特点，神威编译器为用户提供了灵活的编译指示，进行辅助优化。

7.7.1 分支频率指示

#pragma freq [never / always]

#pragma freq 为用户提供了一种机制，可以指定某代码片断在程序中执行的频率信息，指导编译器做出相应的优化。

该指示的语法为：

#pragma freq [never | always]

在 SW 26010 中硬件对条件转移只提供静态的分支预测机制，例如：

beq $1, L1

若条件转移的偏移为负值，则预测转移发生，否则预测转移不发生。

程序中的循环一般都是向负值方向跳，if 分支编译器通常会把概率高的分支做为 fall-through，与硬件的静态分支预测一致。

例如：

```
    if(a==64){
  #pragma freq never
  printf("error!\n");
  exit(1);
}
```

#pragma freq [manycores / fewcores]

编译器对于简单的条件语句会进行 if_conv 优化，使用条件选择指令来代替分支指令，以避免分支转移的开销。但在 SW 26010 运算核心组共享变量模式下，某些情况可能需要避免使用这种优化。比如：

```
if(myid==6)
        yy=5;
```

其中 yy 是一个共享变量，对这个简单条件语句的条件选择优化可能导致 yy 的值可能是 5 也可能是原始值。但用户的语义显然是希望 myid 为 6 的线程看到的 yy 值确定为 5。针对这种因为并行语义带来的问题，我们提供了下面几种解决方案

1、 使用 volatile 关键字修饰 yy，这将使得程序中所有对 yy 的访问必须访存，并不会做条件选择优化

2、 使用-WOPT：if_conv=0 编译选项关闭条件选择优化，增加该编译选项使得当前编译的文件的所有条件分支都不能进行条件选择优化

上面两种方法是保险的，但严重影响程序的性能。

3、 通过编译指示

编译器提供了编译指示 manycores 和 fewcores 来避免这个问题，对于上面的例子：

```
if(myid==6) {
#pragma freq fewcores
        yy=5;
    }
```

#pragma freq fewcores 指示表示下面的语句很少核会执行，这样编译器将不进行条件选择优化，它暗含的分支频率信息也会被编译器采用生成优化的 fall-through 条件分支。

注意：208 号版本以后的编译器改进了编译器 if_conv 优化的算法，即使不使用上述编译指示，仍然会进行条件分支优化，并生成满足正确语义的代码。

7.7.2 对界指示

运算核心在进行 DMA 操作的时候，通常要求主存地址按照一定的字节对界才能获得最好的 DMA 传输性能。下面给出的对界指示可以对 C 和 Fortran 程序中的全局数组或 common 块变量进行对界上的设定。

7.7.2.1 C 对界属性设置

C 对界属性设置：**__attribute__((aligned (align_size)))**

如对全局数组 int a[100];

int a[100] __attribute__ ((aligned (64)));会使得 a 数组首地址 64 字节对界。

注意：align_size 必须为 2 的幂

7.7.2.2 Fortran 对界属性设置

Fortran 对界属性设置：**!*$* align_symbol(symbol, align_size)**

对于 fortran 程序，align_symbol 指示可以控制符号 symbol 存放在 align_size 对界的内存位置。

注意：symbol 通常是一个 common 块变量，symbol 不可以是一个 common 块名或者数组元素；align_size 必须为 2 的幂，并且不可以低于 symbol 的自然对界属性

例如：

```
common/x/ a(1000),b,c
!*$* align_symbol(a,256)
! a 数组将开始于 256B 对界的内存位置
```

7.7.3 循环展开指示

#pragma unroll [unroll_times]

#pragma unroll 为用户提供了一种机制，可以指定某循环在进行循环展开优化时的展开次数。其中 unroll_times 为常数，默认的#pragma unroll[] 表示指定循环展开次数为 4。

使用示例如下：

```
#pragma unroll[8]
for(i=0; i<100; i++) {
    for-loop-body…
}
```

#pragma unroll[8] 指示该循环的展开次数为 8。

注意：一个循环可以有多个#pragma unroll 指示，此时以最后一个为准。另外，如果指定的展开次数编译器无法完成，则编译器会进行自行决定展开次数。

7.7.4 指针别名指示

#pragma alias[alias_rule]

#pragma alias 为用户提供了一种机制，可以指定某函数在进行编译优化时使用的别名分析规则。#pragma alias 在功能上与编译选项-OPT:alias=alias_rule 相同，但在粒度上更加精细，其作用范围为一个函数，而非整个源程序文件。

目前支持的 alias_rule 包括：disjoint、typed、restricted、strongly_typed。正确的语法格式为#pragma alias[disjoint / typed / restricted / strongly_typed]。

alias_rule 的解释如下：

restricted：指示不同的指针之间（*p 和*q）不存在别名；

disjoint：指示不同的间接指针之间（**p 和**q）不存在别名，disjoint 包含 restricted；

typed：指示指向不同基础数据类型的两个指针不存在别名；

strongly_typed：指示指向不同数据类型的两个指针不存在别名；

注意：指针别名指示要谨慎使用，虽然有可能提高程序的性能，但弱化的别名分析可能导致程序出错。

使用示例如下：

```
Func(){
   #pragma alias[disjoint]
   Func-body…
}
```

通过以上指示，相当于为该函数增加了-OPT:alias=disjoint 的优化选项。

注意：该指示一定要放在函数体内部，位置不限。在同时有多个#pragma alias 指示的时候，以最后一个为准。

第八章 使用 SIMD 扩展指令

在 SW 26010 中，运算控制核心和运算核心内部都增加了对 SIMD 扩展的支持。出于功耗以及应用需求等多方面的考虑，SW 26010 的运算控制核心和运算核心支持相同宽度的 SIMD 扩展结构，均为 256 位 SIMD 的设计。

SW 26010 用户要想充分利用 SIMD 扩展结构带来的性能，就必须了解运算控制核心和运算核心各自的SIMD编程接口以及为运算控制核心和运算核心SIMD兼容提供的接口和编程要素。下面的章节将对他们分别进行详细的描述。

8.1 SIMD 概述

8.1.1 有关 SIMD 的几个基本概念

SIMD（Single Instruction Multiple Data）是单指令流多数据流的缩写，SW 26010 运算控制核心和运算核心支持的 SIMD 处理长度均为 256 位 。

8.1.1.1 标量与向量

标量：运算粒度为单个元素

向量：运算粒度为一组有序的标量

数据

数据	数据	数据	数据

8.1.1.2 SW 26010 支持的几种 SIMD 操作

SW 26010 运算控制核心和运算核心都支持 256 位的 SIMD 类型声明。如下：

a) 32*8 的定点运算

即一次操作处理 8 个 32 位的定点运算

b) 256*1 的定点运算

即一次操作处理 1 个 256 位的长整型运算

c) 64*4 的单精度浮点运算

即一次操作处理 4 个单精度浮点运算

d) 64*4 的双精度浮点运算：

即一次操作处理 4 个双精度浮点运算

8.1.1.3 SIMD 的主要特点

我们知道，一个短向量中各个元素是互不相关的，对当前向量元素的操作结果不影响到其他向量元素。比如有一个数组，我们要使数组中的每一个元素 Ai 都乘以一个标量 b，那

么 b 与 A1 相乘的结果不影响 A2 的结果，各自独立。

一条 SIMD 指令相当于一个小的循环，所以可以减少指令数，从而可以降低对指令访问带宽的要求。并且减少了由循环引起的控制相关。例如：

```
int a (N),b (N),c (N)
for(i=0; i < N; i++) {
    a[i]=b[i]+c[i]
}
```

如果采用 SIMD 技术，则最终的实现相当于：

```
for(i=0; i<N; i=i+4){
    a[i:i+3]=b[i:i+3]+c[i:i+3]
}
```

可以发现，循环迭代次数减少为原来的四分之一，该循环的总指令数减少到原来的四分之一，提高了单机的效率。

然而，使用 SIMD 有下面一些局限性：

a) 仅在内存中连续的数据可以被取入向量寄存器进行向量运算；

b) 对 SW 26010 来说，数据在内存中要求 32 字节对界（64*4 单精度浮点要求 16 字节对界），不对界的向量访存会引起异常，然后由操作系统模拟，性能上有很大的降低；

c) 对 SW 26010 运算控制核心来说，定点的指令集不是完备的，例如 32*8 向量不包含乘法、除法等运算；256*1 向量不包含普通的算术运算，仅支持逻辑、移位等操作，因此，含有这些运算的程序难以使用 SIMD 提高性能；对 SW 26010 运算核心同样有相应的限制。

8.1.2 编译器对 C 语言的 SIMD 扩展

SWCC 编译器在 C 语言上扩展了一些新的数据类型和函数，让用户用 C 语言编程来使用 SIMD 功能部件。这些扩展的函数大多数都能直接映射到 SIMD 的指令，在编译的时候由编译器 inline 进来。使用这些扩展编程可以在 C 语言一级获得与汇编编程一样的性能。

8.1.3 SIMD 扩展的适用范围

神威太湖之光计算机系统的 C 语言用户主要使用的语言是并行 C，并行 C 只管并行处理部分，而本文要解决的是单个 CPU 上的短向量问题。并行程序中也可以使用这些 SIMD 的扩展，但最终这些扩展将交由单机编译器 SWCC 来处理。

8.1.4 用户如何让自己的代码使用 SIMD

1、用户程序用标准语言编写，编译器自动进行向量识别，识别出 SIMD 操作，并产生相应的指令。目前的编译器还不具有这种功能。

2、用户通过编译指示来指导编译器进行向量识别，例如：

!$vector

```
for(i=0; i < N; i++) {
```

```
    a[i]=b[i]+c[i];
    }
```

该编译指示就是提示编译器该循环可以进行向量化。目前的编译器还不支持。

3、用户通过显式的 SIMD 内部函数调用、扩充的数据类型等实现 SIMD 的功能，例如：

```
int256    a[M],b[M],c[M];
 for(i=0; i < M; i++) {
    a[i]=b[i]+c[i];
}
```

这个例子就是直接把数据类型改为 256 的整型达到充分利用 SIMD 的目的的。

4、使用汇编编程或使用 C 嵌入汇编，直接使用 SIMD 指令

8.1.5 使用 SIMD 可以预期获得多大的性能提升

SIMD 对性能的提升可由程序本身的特性和程序编写技巧等多方面因素决定。对于完全 SIMD 向量化的程序，对 SW 26010 处理器的运算控制核心和运算核心来说，向量运算的性能可以达到标量的 4 倍，如果浮点使用向量乘加部件，性能可提高到 8 倍。

8.2 SIMD 编程快速入门

本章给出一个简单的 SIMD 程序，用来说明语言的某些基本特征。本章没有详尽地描述 SIMD 的所有特征，目的在于让你对 SIMD 编程先有一个大致的概念。

8.2.1 一个简单的例子

例 1． 下面是用 C 语言编写的一个程序:

```
#include “simd.h”  //包含 SIMD 头文件
main()
{
    unsigned int arr[8] = {1,2,3,4,5,6,7,8};    int i,res[8],t=0;
    intv8 va,vb,vi;
    simd_load (va,arr);
    for (i=16;i>=1;i>>=1) {
        vi = simd_set_intv8(i,i,i,i,i,i,i,i);    //从标准类型向扩展类型赋值
        va ^= va>>i;   //扩展类型的变量使用运算符号
    }
    vb=simd_veqvw(va,vi); //扩展类型的变量使用扩展的内部函数接口
    simd_print_intv8(vb);    //用 intv8 类型的格式打印
    simd_print_intv8(va);    //用 intv8 类型的格式打印
    simd_store(va, res);//intv8 类型的存储
    for (i=0; i<8; i++)
```

```
            t =t+ res[i];
        printf (“%d\n”, t);
    }
```

8.2.2 变量声明

在 SIMD 程序中，变量的声明方式与 C 中是一致的，只是在标准 C 的基础上扩展了一些基本类型，上面的程序中使用了:

intv8 vi;

声明了一个 intv8 类型的 SIMD 变量，表示 vi 是 8 个 32 位整型组成的类型变量。

	224	192	160	128	96	64	32	0
vi:								

8.2.3 将标准类型的数据赋给扩展类型

SWCC 扩展了一些内部函数用于将标准类型的数据赋给扩展类型，上面的例子中：

vi = simd_set_intv8 (i, i, i, i，i，i，i，i);

表示将 8 个 int 类型的数赋给变量 vi，这个语句执行后，vi 中的值为：

	224	192	160	128	96	64	32	0
vi:	i	i	i	i	i	i	i	i

8.2.4 扩展类型变量的运算

SWCC 还对运算符进行了扩展，可以使用 C 语言中的一些运算符对扩展类型的变量进行运算。上面的例子中：

va ^= va>>i;

对 va 中的 8 个整型分别右移 i 位后再分别与原来 va 中的 8 个整型进行异或运算。例如，第一次迭代中，va 进行的右移操作可以同时完成 8 个 32 位整型的移位操作。

有些运算不能用运算符表示，只能使用扩展的内部函数。

例如例子中的“等效”运算：

vb = simd_veqvw(va,vi);

它表示：将 va 和 vi 中 8 个 32 位字分别对应按位逻辑等效。

8.2.5 扩展类型的类型转换

在 SWCC 中支持几种扩展类型的转换，SWCC 支持的几种类型转换和它们各自的用法在后面的章节中有详细的介绍。

8.2.6 扩展类型的打印

SWCC 为每一种扩展的数据类型扩展了两个内部函数用于打印，例如 intv8 类型的两个打印函数为：simd_print_intv8(intv8)和 simd_fprint_intv8(FILE *,intv8)，256 位整型被分成 8 部分打印。上面的例子中：

simd_print_intv8(va);　//用 intv8 类型的格式打印

输出如下：

[15, 5, 4, 6, 7, 2, 3, 1]

8.3 SW 26010 的运算控制核心和运算核心对 C 语言的扩展

本章主要介绍 SW 26010 编译器 SWCC 在 C 语言的数据类型、运算符和内部函数三个方面进行的 SIMD 方面的扩展。本节介绍的扩展对运算控制核心和运算核心是通用的。

8.3.1 数据类型的扩展

在标准 C 的基础上扩展了 4 种数据类型：

1) intv8：32*8 有符号整型
2) uintv8：32*8 无符号整型
3) int256：256*1 有符号长整型
4) uint256：256*1 无符号长整型
5) floatv4：64*4 单精度浮点
6) doublev4：64*4 双精度浮点

扩展类型	说明	值的范围
intv8	8 个 int	$-2^{31}\ldots2^{31}-1$
uintv8	8 个 unsigned int	$0\ldots2^{32}-1$
int256	256 位有符号长整型	—
uint256	256 位无符号长整型	—
floatv4	4 个 float	1.17549435e-38…3.40232847e38
doublev4	4 个 double	2．2250738585072013e-308 …1.7976931328623158e308

8.3.2 对界（Alignment）

本节描述 SIMD 的对界要求，当使用 SIMD 数据进行编程的时候，用户必须清楚这些对界问题，因为编译器不会在对界方面产生报警或错误信息，向量类型的数据是否出现了不对界问题需要用户判断。

向量类型的数据项在内存中除了 floatv4 类型是 16 字节对界其余都是 32 字节对界。编译器保证用向量类型定义的变量在内存中是 32 字节（16 字节）对界的，编译器还保证所有

类型数组的首地址是 32 字节对界的。在 SW 26010 处理器构架中，不对界的 Load/Store 会引发异常，操作系统收到异常信号后会将这些 Load/Store 拆分成标准类型的 Load/Store，然而这样会大大的降低性能。

包含向量类型的结构体（struct）/联合（union），编译器保证它们是 32 字节（或 16 字节）对界的（必要的时候在内部进行了垫塞）。例如，下面的结构体：

struct st { doublev4 vb; int a; }

sizeof (struct st)返回值是 64。

8.3.3 SIMD 对 C 操作的扩展

假设定义了下面的一些变量：

```
intv8 x;   int y;   long la;   floatv4 fv;   doublev4 *p, dv.
float f;   double d;
```

a) sizeof()

sizeof(x)和 sizeof(*p)的返回值都是 32。

b) 赋值

扩展类型可以在定义变量的时候像数组一样的赋初值，例如：

intv8 va = {1,2,3,4,5,6,7,8}； 赋值的结果为：

	224	192	160	128	96	64	32	0
va:	8	7	6	5	4	3	2	1

或者

floatv4 vf = {2.323}；赋值的结果为：

	192	128	64	0
vf:	0	0	0	2.323

另外，除了向量浮点类型和向量整型之间不能互相赋值之外，其他的赋值都是允许的。比如：

```
fv = 3;
x = 2.345;
*p = dv;
```

其中，fv=3 的赋值结果为：

	192	128	64	0
fv:	3.0	3.0	3.0	3.0

x=2.345 的赋值结果为：

	224	192	160	128	96	64	32	0
x:	2	2	2	2	2	2	2	2

等号左/右侧的类型不一样隐含着强制类型转换，强制类型转换的具体含义参见“SIMD 扩展类型的转换”。

这里需要注意，下面的两种赋值方法都是允许的，但是他们是有区别的：

i) intv8 va = {2};

ii) intv8 vb = 2;

其中，i)表示给 va 的最低 32 位赋初值 2，va 的其他部分为缺省值 0。而 ii)表示将 int 类型的常数强制转换为 intv8 类型，这种强制类型转换是扩展式的，vb 中的四个部分都是 2。

i)的结果

	224	192	160	128	96	64	32	0
va:	0	0	0	0	0	0	0	2

ii)的结果：

	224	192	160	128	96	64	32	0
va:	2	2	2	2	2	2	2	2

c) 取地址符号

对向量类型的取地址符号是允许的，&x 表示取变量 x 的地址。

d) 指针

指向扩展类型的指针运算与标准类型的相同。例如 p+1 表示指向下一个扩展类型变量的指针。指针引用*p 隐含着从 p 的地址进行 256 位的装入操作。

e) 运算符的扩充

同时，本文本对标准 C 中的运算符进行了扩充，例如“+、-、*、/”等运算符现在都可以用于表示扩展浮点数据之间的运算。下面列出了所有的扩展情况，运算符后跟的数据类型表示用这些类型定义的变量可以进行相应的运算操作（“：”前是标准 C 中的运算符号，“：”后面是该运算符扩展到的数据类型）：

+：intv8，uintv8，int256，uint256，floatv4，doublev4

-：intv8，uintv8，int256，uint256，floatv4，doublev4

*：floatv4，doublev4

/：floatv4，doublev4

&：intv8，uintv8，int256，uint256，

|：intv8，uintv8，int256，uint256，

^：intv8，uintv8，int256，uint256，

>>：intv8，uintv8，int256，uint256，

<<：intv8，uintv8，int256，uint256，

==：intv8，uintv8，floatv4，doublev4

<=：intv8，uintv8，floatv4，doublev4

<：intv8，uintv8，floatv4，doublev4

>=：intv8，uintv8

在编写 SIMD 程序时可以直接使用这些运算符来进行扩展数据类型上的运算。在使

用的时候需要注意以下几点：

（1）对 intv8 类型扩展的“>>”符号，与标准 C 中的约定一样，如果“>>”左边的操作数是有符号的，那么最终生成的指令为算术右移，如果是无符号的，最终生成的指令为逻辑右移。

（2）int256 类型支持的加减法相当于 64*4 的四个 long 类型的向量加减法，因为 SW 26010 指令集里面针对 64*4 这种长整型向量的操作仅包含加法和减法，而 int256 本身也没有算术意义，从编译器具体实现的角度，我们认为没有比较再单独提供一种新类型，因而采用 int256 类型来实现 64*4 的长整型向量的加减。

（3）“>>”运算符仅支持 uint256 类型，为逻辑右移，不支持 int256 类型。

8.3.4 SIMD 对 C 内部函数的扩展

对于扩展数据类型上的每一种运算操作我们都扩充了相应的内部函数或宏定义接口，供编程时调用。编译器成功安装后，只需要将“simd.h”include 到程序中就可以使用这些扩展数据类型和扩充的内部函数或宏定义了。

例如，要将一段普通的 C 程序向量化经常需要将标准类型的数据先映射到 SIMD 变量中，然后才进行运算。本文本扩充了 5 个 SIMD 宏定义接口，用于进行扩展数据类型与标准类型之间的映射操作。这 5 个宏定义是：

simd_load() —— 对界装入，将 256 位标准类型的数据从向量要求的对界内存地址映射到扩展类型变量

simd_loadu() —— 不对界装入，将 256 位标准类型的数据从不对界的内存地址映射到扩展类型变量

simd_loade() —— 装入并扩展，将 64 位标准类型的数据从内存映射到扩展类型变量的低 64 位并扩展到高 192 位

simd_store() —— 对界存储，将扩展类型变量中的数据映射到向量要求的 256 位长度的对界内存中

simd_storeu() —— 不对界存储，将扩展类型变量中的数据映射到 256 位长度的不对界内存中

其中 simd_load、simd_loade、simd_store 适用于所有扩展数据类型的装入/存储，simd_loadu、simd_storeu 支持 intv8、floatv4、doublev4 类型的不对界装入/存储。这些宏定义有两个参数，第一个是扩展类型，第二个是标准数据类型的指针，参数具体允许的类型请参照 9.7 节。

在编写 SIMD 程序的过程中还经常需要给这些扩展类型的变量赋初值，本文本提供了一些宏定义来做这件事情。例如定义一个 intv8 类型的变量 va，要给该变量赋初值，该初值应该包含四个整型常量，比如为{1,2,3,4,5,6,7,8}，那么可以这样编写：

```
va = simd_set_intv8 (1, 2, 3, 4,5,6,7,8);
```

在变量 va 中，这 8 个值的存放位置是这样的：

高 ←———————— 低

8	7	6	5	4	3	2	1

相应的 int256、doublev4 和 floatv4 类型也有类似的函数： simd_set_int256、simd_set_doublev4 和 simd_set_floatv4，也可以用“=”符号直接为该类型的变量赋初值，例如：

doublev4 vlint;

vlint = 123.456;

对于扩展数据类型，许多操作是用标准 C 的运算符号或库函数所无法表示的，这样的操作必须靠调用内部函数或宏定义来实现，例如：

simd_vinsf () —— 浮点向量插入

simd_vextf () —— 浮点向量提取

simd_vcpyf () —— 浮点向量拷贝

simd_vfseleq () —— 浮点向量等于选择

simd_vfsellt () —— 浮点向量小于选择

simd_vfselle () ——浮点向量小于等于选择

扩展类型的打印，我们提供了 12 个打印函数，分别用于打印 6 种扩展类型：

simd_print_intv8(),

simd_print_uintv8(),

simd_print_int256(),

simd_print_uint256(),

simd_print_doublev4(),

simd_print_floatv4(),

simd_fprint_intv8(),

simd_fprint_uintv8(),

simd_fprint_int256(),

simd_fprint_uint256(),

simd_fprint_doublev4(),

simd_fprint_floatv4(),

例如，

va = simd_set_intv8 (1,2,3,4,5,6,7,8);

simd_print_intv8(va);

打印的结果是：

$> [8,7,6,5,4, 3, 2, 1]

另外，还有一些操作，一般由编译器自动生成，这里也提供了一些内部函数接口供编程时直接调用，主要有扩展浮点类型的乘加、乘减、乘减和负乘减操作等。

所有扩展运算符的运算操作也都提供了内部函数接口。

这里只是对扩充内部函数接口进行了简单的介绍，详细的描述请查阅 9.5 和 9.6 节。

8.3.5 扩展类型的类型转换

在扩展类型中，SWCC 只支持如下几种转换：

a）扩展浮点类型单、双精度之间的转换

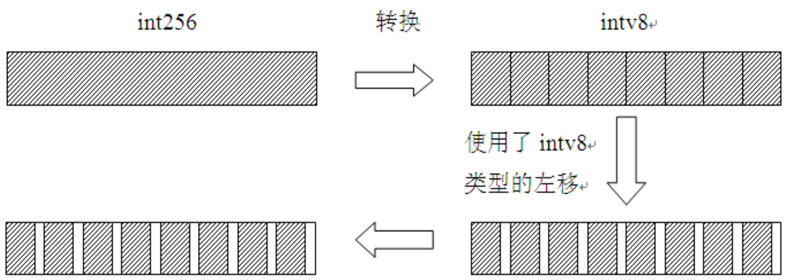
扩展浮点类型 floatv4 和 doublev4 之间的转换，意义上与标准类型的单、双精度的类型转换一样，编程中写法上也相同，例如：

```
floatv4    fv;       doublev4    dv;
fv = (floatv4) dv;
```

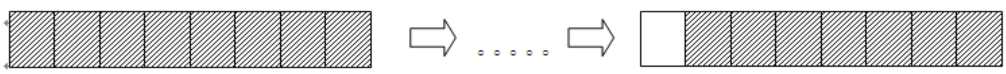
编程中如果出现 floatv4 与 doublev4 混用的情形，可以不用写成上述强制类型转换的形式，编译器会自动根据赋值语句左边的类型或内部函数参数的类型进行相应的类型转换。这些都和标准类型的 double 和 float 的用法一样。

b）扩展整数类型 intv8 和 int256 之间的转换

这两种类型之间的转换，变量中的数据不会产生任何的变化。编译器之所以支持这两种类型之间的转换，只是为了有效利用系统提供的指令。例如下面的例子中，一个 int256 的数转换成 intv8 后，使用 intv8 的移位操作可以达到使用 int256 类型很多条操作才能达到的效果：



同样的，intv8 类型的数据，转换成 int256 类型，并使用 int256 类型的操作后也可以收到事半功倍的效果。例如下图中先将 intv8 转换为 int256，使用 int256 的移位操作后可以很轻松的将最高的 32 位清空：



在编程过程中，用户灵活运用这两种类型的转换往往可以收到事半功倍的效果。

c）标准类型到扩展类型的转换

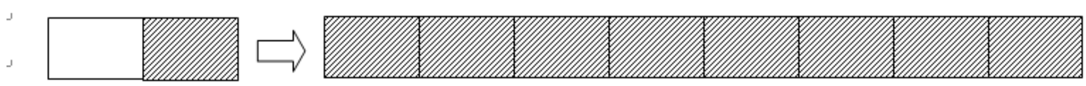
（1）值扩展

标准类型到 intv8，floatv4，doublev4 的转换是值的扩展。

例如：

```
intv8 va;         int x;
    va = (intv8)x;
```

把 x 的值复制 8 份放到 va 中：



在 SW 26010 运算控制核心中，32 位的 int 数据是放在一个 64 位定点寄存器的低 32 位的。所以上面的图中，左边是一个 64 位的寄存器，其中低 32 位是有效数据，扩展到右边一个 256 位的向量寄存器中。

再例如：

```
floatv4 fv;
fv = 3245.345;
```

结果是：

3245.345	3245.345	3245.345	3245.345

long 到 intv8 的转换，例如：intv8 vx;　　long l;

```
vx = (intv8) l;
```

相当于：

```
vx = (intv8)((int)l);
```

float 到 intv8 的转换，例如：float f;　　intv8 va;

```
va = (intv8)f;
```

相当于：

```
va = (intv8)(int)f;
```

（2）符号扩展

标准类型到 int256 的转换是值的扩展。

例如：

```
int256 va;          long x=-2;
va = (int256)x;
```

把 x 的值复制到 va 的低 64 位中，同时把符号位扩展到高 192，结果是：

fffff…fff	fffff…fff	fffff…fff	fffff…ffe

int 到 int256 的转换，例如：int256 vx;　　int　l;

```
vx = (int256) l;
```

相当于：

```
vx = (int256)((long)l);
```

d) 扩展类型到标准类型的转换

只是取扩展类型变量中最低的那部分。

例如：

```
int a;    intv8 va = {1,2,3,4,5,6,7,8};
a = va;
printf (“%d\n”,a);
```

打印结果为： 1

8.3.6 扩展类型与标准类型之间的数据交换

SWCC 可以通过以下两种方法在标准数据类型与扩展类型的变量之间进行数据交换：（1）通过内存交换；（2）通过 SW 26010 运算控制核心提供的一些特殊操作如：vcpyf，vextf 等，上一节中的标准类型到扩展类型的强制类型转换编译器内部也是使用了这些操作完成的。后一种的性能一般要好于前一种，但是后一种一般由编译器优化使用。例如要将一个整型数组的 8 个连续整型数据传递到一个 intv8 类型的向量变量中，下面的写法是错误的：

```
intv8 vsum;
int vertex[8];
vsum=(intv8)vertex[0]; /*  错误写法  */
```

由于 vertex[0]表示一个 int 类型的值，所以这种写法相当于上一节中的强制类型转换，是扩展方式的：

vertex[0]	vertex[0]	vertex[0]	vertex[0]	vertex[0]	vertex[0]	vertex[0]	vertex[0]

而我们想要的结果实际上是：

vertex[7]	vertex[6]	vertex[5]	vertex[4]	vertex[3]	vertex[2]	vertex[1]	vertex[0]

正确的方法主要有三种：

a)　使用指针：

```
intv8 vsum, *p;
int vertex[8];
p = (intv8 *) (&(vertex[0]));
vsum = *p;
```

b)　使用内部访存函数：

```
/*把以&(vertex[0])为起址，长度为 256 位的内容拷贝到 vsum 中*/
intv8 vsum;      int vertex[8]
simd_load(vsum, &(vertex[0]));
```

c)　使用内部赋值函数：

```
intv8 vsum;      int vertex[8];
vsum = simd_set_intv8 (vertex[0], vertex[1], vertex[2], vertex[3]，vertex[4]，
vertex[5]，vertex[6]，vertex[7]);
```

目前三种方法都使用了内存作为交换通道，在性能上没有差异。在本例子中，因为前两种方法都只需要一次访存，而第三种方法需要多次访存，前两种方法更好。但如果要传递的数据不在连续的内存区域中，比如，要将 8 个 int 类型的变量传递到一个 intv8 变量中，第三种方法就更好一些，例如：

```
intv8 va;
int x,y,z,w,m,n,s,t;
```

va = simd_set_intv8 (x, y, z, w,m,n,s,t);

8.3.7 应用程序二进制接口（ABI）

8.3.7.1 寄存器使用约定

扩展类型使用浮点寄存器，使用过程中的约定与普通浮点类型相同：

寄存器	使用说明
$f0	用于存放扩展类型、浮点类型或者复数类型的返回值，不能跨过程使用。
$f1-$f9	Save Registers.可以跨过程调用使用。
$f10-$f15	临时寄存器。不能跨过程使用。
$f16-$f21	用于传递前 6 个扩展或浮点类型的实参。
$f22-$f30	临时寄存器。不能跨过程使用。
$f31	浮点值总是 0.0。扩展类型中可用于表示 128 位全 0。

8.3.7.2 栈帧（Stack Frame）

在神威太湖之光系统中，栈指针保持 16 字节对界，但对于除 floatv4 的扩展类型，需要调整栈指针 32 字节对界。

那些可以跨过程使用的寄存器（$f1-$f9），如果要在本函数体内使用，需要在函数体的最前面将这些寄存器保留到栈中。因为不知道这些寄存器在使用以前是用作普通浮点寄存器还是用作向量寄存器，所以在保存的时候要把整个 256 位保存到栈中，恢复的时候恢复 256 位，分别使用 vldd 和 vstd 指令。

在函数调用中，与 Alpha 的约定一样，按照从左到右的顺序，前面的六个参数（浮点或者向量）使用$f16-$f21 寄存器，第七个往后的参数需要保存在栈中。

8.4 串行程序的向量化

在原有程序的基础上，要使用系统的 SIMD 功能，最直接的办法是先对程序进行向量化。

向量化时首先要考虑到一些限制：

硬件限制：

向量化受限于硬件的限制。在 SIMD 扩展中，向量化访存操作必须是地址连续的，且要求 32 字节（floatv4 为 16 字节）对界的。意味着虽然有些循环是可向量化的，但具体针对 SW 26010 体系结构，需要进行程序变换后才能向量化。另外，由于扩展定点类型的指令集是不完备的，例如 8*32 的乘法、除法等，这样的循环在 SW 26010 上没有办法向量化。

程序形式：

循环体中有复杂的关键字、操作、数据访问和内存操作等，都会影响对程序的相关性分析和向量化分析，从而不利于用户或编译器对程序进行向量化。

了解了这些限制，你可以修改程序，避开这些限制，实现高效向量化。下面介绍一些针对循环进行向量化的技术。

8.4.1 程序形式对向量化的限制

8.4.1.1 相关性的限制

向量化的基础是相关性分析。例如

例 2.

```
float data[N];
int i;
for (i=1; i<N-1; i++)
{
    data[i] = data[i-1]*0.25 + data[i]*0.5 + data[i+1]*0.25;
}
```

上述循环不能向量化，因为写当前元素 data[i]时依赖于使用前面的元素 data[i-1]，而它在前一个迭代发生了改变。为了更清楚一点，下面列出头两个迭代的数组访问图：

```
i=1：   READ data[0]
        READ data[1]
        READ data[2]
        WRITE data[1]
i=2：   READ data[1]
        READ data[2]
        READ data[3]
        WRITE data[2]
```

按照循环的执行顺序，第二次迭代读 data[1]的值就是第一次迭代所写的值。而向量化必须要求迭代间可以并行执行，且不改变原始循环的语义。

数据相关性分析包括寻找内存访问可能重叠的条件。给定程序的两个访问，相关的条件如下：

- 访问的变量在内存中是同一个（或者重叠）区域的别名
- 对数组访问，分析下标的关系是否可能相等

8.4.1.2 循环结构的限制

循环可以是常用的 for，while-do 或 repeat-until 结构，也可以是使用 goto 和标号组成的。然而，循环必须是只有一个入口和一个出口时才能被向量化。

例 3．一个可向量化的循环：

```
double a[100], b[100], c[100];
while ( i<n )
{
    a[i]= b[i] * c[i];
        i++;
```

```
    }
```

例 4．一个不可向量化的循环：

```
    while ( i<n )
    {
        if ( condition ) break;
        /*  循环体的另一个出口  */
        i++;
    }
```

8.4.1.3 循环退出条件的限制

循环退出条件决定循环执行的迭代次数。向量化需要迭代次数是可以计算的，也就是说迭代次数必须是下述表达式：

- 常数
- 循环不变量
- 外层循环控制变量的线性函数

8.4.2 循环向量化步骤

通常，确定循环可以向量化后，向量化一个循环可以分以下步骤：循环分裂、变量替换、操作替换。

8.4.2.1 循环分裂

例 5．对循环进行循环分裂

```
int vertex[n], sum;
for (i = 0; i < n; i++)
{
    sum=sum+vertex[i]；
};
```

变为：

```
for (i = 0; i < n/8; i+=8)
{
    sum=sum+vertex[8*i]；
    sum=sum+vertex[8*i+1]；
    sum=sum+vertex[8*i+2]；
    sum=sum+vertex[8*i+3]；
    sum=sum+vertex[8*i+4]；
    sum=sum+vertex[8*i+5]；
```

```
        sum=sum+vertex[8*i+6];
        sum=sum+vertex[8*i+7];
};
for (i = (n/8)*8; i < n; i++)
{
        sum=sum+vertex[i];
};
```

例如上面的例子中如果 n 不能被 8 整除，将最后不够一次向量操作的部分分裂出来，前面的部分就可以向量化了。

8.4.2.2 变量替换

声明扩展类型变量有两种策略，一种是在算法一级把关键变量都替换成扩展类型变量，这种方法改造程序彻底，但对程序改动较大；一种方法是在需要向量化的地方才引入扩展类型变量用于 SIMD 操作，该方法牵涉的地方较小，有利于快速向量化和调试。

```
int sum; → intv8 vsum;
int vertex[n]; → intv8 tx;
```

8.4.2.3 操作替换

若不是全局替换，在用 SIMD 操作替换普通操作之前，需要把标准类型变量映射（或复制）到扩展类型变量上，操作完之后，再把扩展类型变量映射（或复制）回标准类型变量。如

```
simd_load (tx, &(vertex[8*i]));    //将标准类型映射到扩展类型
simd_store (tx, &(vertex[8*i]));    //将扩展类型映射到标准类型
```

8.4.3 如何更好地向量化

8.4.3.1 合理使用数组

为了有效的进行向量化，程序中对数组的操作需要尽量的连续访问，这意味着对于多维数组，尽量选择在最右边的维（C 语言）进行向量化。

另外，尽量让数组作为结构的一个域，而不是使用结构类型的数组。例如下面的例子中，右边的更好向量化。

Array of Structures	Structure of Arrays

struct
{ float a, b, c, d, e, f }s[99];
int i, n;
for (i=0; i < n; i++)
x += s[i].a * s[i].e;
	struct { float a[99], b[99], c[99], d[99], e[99], f[99] } s;
	int i, n;
	for (i=0; i < n; i++)
	x += s.a[i] * s.e[i];

8.4.3.2 对函数调用的处理

如果要向量化的循环中包含一个函数调用，占用很大开销，可以通过将该函数 inline 进循环体或者将该函数体向量化的方法来解决。例如下面的一段程序：

原来的程序	inline	函数体向量化

unsigned fc[FCL],q,k;
for (k=0;k<FCL;k++)
if (pjoyl(fc[k]&q))
s—-;
else s++;
int pjoyl(unsigned s) {
int i;
for(i=16;i>=1;i>>=1)
s ^= s>>i;
return s%2;
}
	int fc[FCL],q,tmp[8],k,i;
	intv8 v,vq,vi;
	vq=simd_set_intv8(q,q,q,q,q,q,q,q);
	for(k=0;k<FCL;k+=8){
	simd_load(v,&fc[k]);
	v=v&vq;
	for(i=16;i>=1;i>>=1) {
	vi=
	simd_set_intv8(i,i,i,i,i,i,i,i);
	v ^= v>>vi;
	simd_store(v,tmp);
	for(i=0;i<8;i++)
	if (tmp[i]%2) s--;
	else s++;
	}
	}
		unsigned fc[FCL],q,t[8],k;
		intv8 vq,v;
		vq=simd_set_intv8(q,q,q,q);
		for(k=0;k<FCL;k+=8) {
		simd_load(v,&fc[k]);
		v=v&vq;
		vpjoyl(v,t);
		for(i=0;i<8;i++)
		if(t[i]%2) s--;
		else s++;
		}
		void vpjoyl(intv8 v,int*r){
		int i; intv8 vi;
		for(i=16;i>=1;i>>=1) {
		vi=
		simd_set_intv8(i,i,i,i);
		v ^= v>>vi
		}
		simd_store(v,r);
		}

在上面的例子中，扩展类型变量 vs（程序中是 v）作为函数的参数传递进 vpjoyl。在 SIMD 扩展中，扩展类型在函数调用过程中的使用约定与浮点类型的一致。最左边的六个参数通过寄存器$f16-$f21 传递，第七个往后的参数通过栈传递。返回值放在$f0 中。

8.4.3.3 更有效的使用 Cache

高效地使用各级 Cache 对于性能的提高是极为重要的，到主存储器中访问数据要比在一级 Cache 中访问数据慢数十倍。为了更好地使用 Cache，程序需要尽量使用同一个 Cache 行的所有数据而不是各不同 Cache 行的部分数据，而且程序最好能在数据被替换出 Cache 以前尽量多的重用这些数据。当然，为了从 SIMD 部件中获得性能的提升，也要求程序最好访问连续的内存区域，这一点来讲，Cache 与 SIMD 部件对程序的要求是一样的。

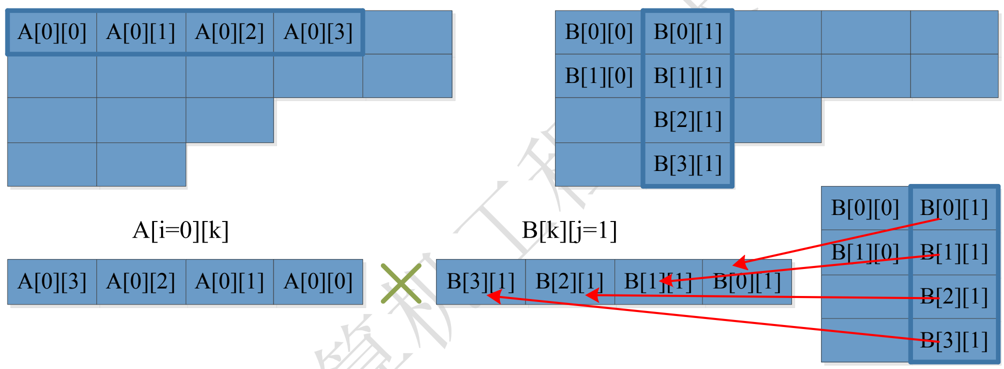
例如在矩阵乘的例子中，第一个矩阵通常按照行优先的顺序访问，第二个矩阵则要按照列优先的顺序访问，显然对第一个矩阵的访问具有空间局部性，因为访问的是连续地址，而对第二个矩阵的访问每一次访问的经常是一个新的 Cache 行。为了实施向量化，通常需要将

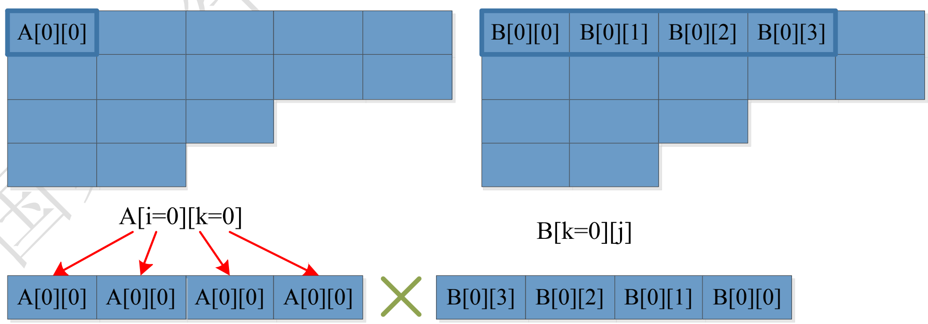
第二个矩阵的某一列数据逐一装入，然后重新组织到向量寄存器中，这样会增加很大的开销。我们以程序的形式来说明：

```
for (i=0;i<n;i++)
        for(j=0;j<n;j++)
              for(k=0;k<n;k++)
                c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

如果直接进行向量化，则程序变换为：

```
for(k=0;k<n;k += 4) {
        simd_load(va， &a[i][k]);
        vb <- (b[k][j], b[k+1][j], b[k+2][j], b[k+3][j])
        vt += va*vb;
}
simd_store(vt, temp);
c[i][j] = temp[0]+temp[1]+temp[2]+temp[3];
```





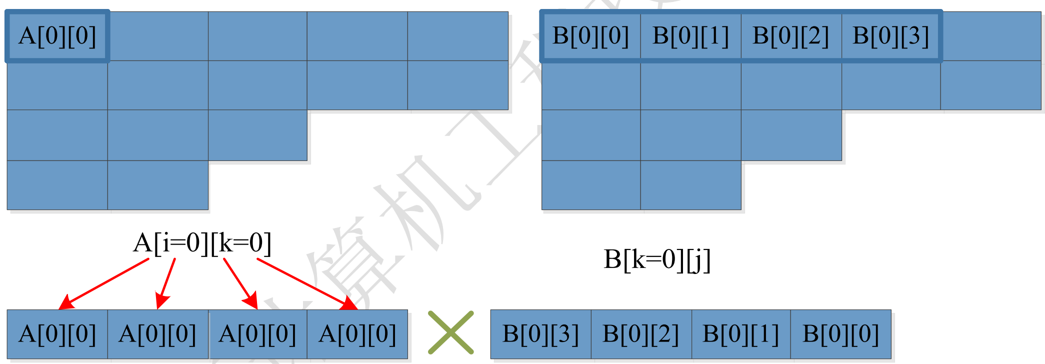
向量化的时候，数组 A 连续的 4 个元素可以直接装载到向量中，而数组 B 对应作乘法的元素在内存中并不连续，相当于是对多个 cache 行的访问，这样就需要花费较大的开销去拼接。但是，如果把程序做一下变换，就可以很好的避开访问不同 cache 行的开销，把 k 级循环和 j 级循环互换，然后再做向量化，如下：

```
for (i=0;i<n;i++)
```

```
        for(j=0;j<n;j++)
            for(k=0;k<n;k++)
                c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

向量化后程序变换为：

```
for(i=0;i<n;i++) {
    for (k=0;k<n;k++) {
        simd_loade(&a[i][k]);
        for (j=0;j<n;j += 4) {
            simd_load(vb,&b[k][j]);
            vt = va*vb;
            simd_load(vs, &c[i][j]);
            vs += vt;
            simd_store (vs, &c[i][j])
        }
    }
}
```



由于 j 是最内层循环，因此对 B[k][j]的访问是连续的，而 A[i][k]在 j 级循环里面仅相当于一个常数，这样就可以用数组 A 的一个元素同时去乘数组 B 的连续四个元素，完全屏蔽了不连续的 cache 访问。

8.4.3.4 对界问题的处理

在 9.4.2.3 节介绍的操作替换过程中，使用 simd_load 或 simd_store 操作进行变量映射的时候，需要保证标准类型变量为 32 字节对界（对于 floatv4，只需 16 字节对界）。如果出现了不对界的访存，执行的时候操作系统将最终处理该访存引起的异常，这将付出更昂贵的代价。

编译器可以保证数组变量的起始地址为 32 字节对界，其余要求用户保证。

对于有些不对界的情况，用户可以进行调整，例如：

例 6．处理扩展类型的对界问题

```
double a[n],b[n+2];
for ( i=0; i<n; i++) {
    a[i]=a[i]+b[i+2];
}
```

在向量化该循环时，用户若保证数组 a 的对界要求，必然就不能保证数组 b 的对界要求，我们可以采用两种方法来解决该问题。

1、采用 SIMD 整理指令对数组 b 进行拼接

由于对数组 b 的访问不能保证 32 字节对界的要求，因此需要降低对数组 b 的访问粒度，将 b[i+2]、b[i+3]、b[i+4]、b[i+5]分别取出，然后组合成向量与数组 a 进行向量运算。改写后的 SIMD 程序如下：

```
#include “simd.h”
double   a[n],b[n+2];
doublev4    aa, bb,bb1,vtmp;
for ( i=0; i<(n/4)*4; i +=4 ) {
    simd_load(aa, &(a[i]));
    simd_load(bb, &(b[i]));
    vtmp=simd_vextf2(bb);
    bb1=simd_vinsf0(vtmp,bb1);
    vtmp=simd_vextf3(bb);
    bb1=simd_vinsf1(vtmp,bb1);
    simd_load(bb, &(b[i+4]));
    vtmp=simd_vextf0(bb);
    bb1=simd_vinsf2(vtmp,bb1);
    vtmp=simd_vextf1(bb);
    bb1=simd_vinsf3(vtmp,bb1);
    aa=aa+bb1;
}
for ( i=(n/4)*4; i<n; i++) {
        a[i]=a[i]+b[i+2];
}
```

调整后，分别在&b[i]和&b[i+4]处使用两次对界的装入，然后拼接成从&b[i+2]开始的连

续的 4 个 64 位双精度浮点数进行运算。

2、使用不对界访存接口直接处理

根据结构手册提供的向量不对界访存指令，我们提供了两个不对界访存接口，分别是向量不对界取 simd_loadu 和向量不对界存 simd_storeu。使用不对界访存指令可以不考虑向量类型 32 字节对界（floatv4 为 16 字节对界）的要求，只需满足存取基本单元的对界要求即可，如 int 和 float 类型需要 4 字节对界，double 类型需要 8 字节对界。程序转换如下：

```
#include “simd.h”
double    a[n],b[n];
doublev4    aa, bb,bb1,vtmp;
for ( i=0; i<(n/4)*4; i += 4) {
      simd_load(aa, &(a[i]));
      simd_loadu(bb, &(b[i+2]));
      aa=aa+bb;
      simd_store(aa,&a[i]);
}
for ( i=(n/4)*4; i<n; i++) {
            a[i]=a[i]+b[i+2];
}
```

调用不对界访存接口直接装入 b[i+2]，然后直接做向量加法。第二种方法在性能上要优于第一种。

8.5 运算控制核心扩充内部函数、宏定义库

8.5.1 使用的符号说明

VRav：第一个向量参数

VRbv：第二个向量参数

VRcv：第三个向量参数

VRdv：第四个向量参数

VRc：向量返回值

Rbv：无符号长整型变量，一般用于表示地址

Va：一段内存地址中的数据 f

fp:FILE*类型的变量

8.5.2 装入/存储操作宏定义

宏定义	指令	操作	参数说明	
			VRav	Rbv
simd_load	vldd/vlds	装入	扩展类型	int*,long*,float*,double*
simd_loade	ldwe/ldde/ldse	装入并扩展	扩展类型	int*,long*,float*,double*

simd_store	vstd/vsts	存储	扩展类型	int*,long*,float*,double*
simd_loadu	vld(w/s/d)_ul/ vld(w/s/d)_uh	不对界装入	扩展类型	int*,long*,float*,double*
simd_storeu	vst(w/s/d)_ul/ vst(w/s/d)_uh	不对界存储	扩展类型	int*,long*,float*,double*

说明：当 Rbv 为 int * 或 long *类型时，VRav 只能是 intv8；Rbv 为 float *时，VRav 只能是 floatv4 类型；Rbv 为 double *类型时，VRav 只能是 doublev4 类型。如果上述类型不匹配，编译器将报错。

函数名

simd_load —— 扩展类型装入

对应指令

vldd/vlds

参数说明

a) intv8 类型的装入

[u]intv8 va;

[unsigned] int *addr;[[unsigned] long *addr;] //intv8 类型的装入

b) floatv4 类型的装入

floatv4 va;

float *addr; //floatv4 类型的装入

c) doublev4 类型的装入

doublev4 va;

double *addr; //doublev4 类型的装入

返回值

a) intv8 类型的装入

[u]intv8 //intv8 类型的装入

b) floatv4 类型的装入

floatv4; //floatv4 类型的装入

c) doublev4 类型的装入

doublev4; //doublev4 类型的装入

功能说明

扩展类型的装入，将 32 字节（floatv4 类型的装入是 16 字节）长度的数据从连续内存区域中装入到一个向量变量中。

函数名

simd_loade —— 扩展类型装入并扩展

对应指令

ldwe/ldde/ldse

参数说明

[u]intv8 va;

[unsigned] int *addr;[[unsigned] long *addr;] //intv8 类型的装入

floatv4 va;

float *addr; //floatv4 类型的装入

doublev4 va;

double *addr; //doublev4 类型的装入

返回值

[u]intv8 //intv8 类型的装入

floatv4; //floatv4 类型的装入

doublev4; //doublev4 类型的装入

功能说明

扩展类型的装入并扩展，将 8 字节（floatv4 类型的装入是 4 字节）长度的数据从连续内存区域中装入到一个向量变量的低 64 位，并且将该数据扩展到高 192 位，使向量寄存器的四个寄存器具有相同的值。intv8 是取出 4 字节，然后扩展成 8 个分量。

函数名

simd_store —— 扩展类型存储

对应指令

vstd/vsts

参数说明

[u]intv8 va;

[unsigned] int *addr;[[unsigned] long *addr;] //intv8 类型的存储

floatv4 va;

float *addr; //floatv4 类型的存储

doublev4 va;

double *addr; //doublev4 类型的存储

返回值

无

功能说明

扩展类型的存储，将一个向量变量中的数据存储到 32 字节（floatv4 类型的装入是 16 字节）连续内存区域中。

函数名

simd_loadu —— 扩展类型不对界装入

对应指令

(vldw_ul，vldw_uh)

(vlds_ul，vlds_uh)

(vldd_ul，vldd_uh)

参数说明

a) intv8 类型的装入

[u]intv8 va;

[unsigned] int *addr; //intv8 类型的装入

b) floatv4 类型的装入

floatv4 va;

float *addr; //floatv4 类型的装入

c) doublev4 类型的装入

doublev4 va;

double *addr; //doublev4 类型的装入

返回值

a) intv8 类型的装入

[u]intv8 //intv8 类型的装入

b) floatv4 类型的装入

floatv4; //floatv4 类型的装入

c) doublev4 类型的装入

doublev4; //doublev4 类型的装入

功能说明

扩展类型的不对界装入，将 32 字节（floatv4 类型的装入是 16 字节）长度的数据从连续内存区域中装入到一个向量变量中。

函数名

simd_storeu —— 扩展类型的不对界存储

对应指令

(vstw_ul，vstw_uh)

(vsts_ul，vsts_uh)

(vstd_ul，vstd_uh)

参数说明

[u]intv8 va;

[unsigned] int *addr; //intv8 类型的存储

floatv4 va;

float *addr; //floatv4 类型的存储

doublev4 va;

double *addr; //doublev4 类型的存储

返回值

无

功能说明

扩展类型的不对界存储，将一个向量变量中的数据存储到 32 字节（floatv4 类型的装入

是 16 字节）连续内存区域中。

8.5.3 定点向量运算操作函数

内部函数	指令	操作	参数说明			
			返回值	VRav	VRbv	VRcv
simd_vaddw	vaddw	+	intv8	intv8	intv8/lit8	—
simd_vsubw	vsubw	-	intv8	intv8	intv8/lit8	—
simd_vandw	vlogc0	&	intv8	intv8	intv8/lit8	—
simd_vbicw	vlog30	与非	intv8	intv8	intv8/lit8	—
simd_vbisw	vlogfc	|	intv8	intv8	intv8/lit8	—
simd_vornotw	vlogf3	或非	intv8	intv8	intv8/lit8	—
simd_vxorw	vlog3c	^	intv8	intv8	intv8/lit8	—
simd_veqvw	vlogc3	等效	intv8	intv8	intv8/lit8	—
simd_vsllw	vsllw	<<	intv8	intv8	int/lit5	—
simd_vsrlw	vsrlw	>>	intv8	intv8	int/lit5	—
simd_vsraw	vsraw	算术右移	intv8	intv8	int/lit5	—
simd_vcmpgew	vcmpgew	大于等于比较	int	intv8	intv8/lit8	—
simd_vcmpeqw	vcmpeqw	等于比较	intv8	intv8	intv8/lit8	—
simd_vcmplew	vcmplew	小于等于比较	intv8	intv8	intv8/lit8	—
simd_vcmpltw	vcmpltw	小于比较	intv8	intv8	intv8/lit8	—
simd_vcmpulew	vcmpulew	无符号小于等于比较	intv8	intv8	intv8/lit8	—
simd_vcmpultw	vcmpultw	无符号小于比较	intv8	intv8	intv8/lit8	—
simd_vrolw	vrolw	循环左移	intv8	intv8	int/lit5	—
simd_vaddl	vaddl	长字向量加法	int256	int256	int256	—
simd_vsubl	vsubl	长字向量减法	int256	int256	int256	—
simd_sllow	sllow	八倍字逻辑左移	int256	int256	int/lit8	—
simd_srlow	srlow	八倍字逻辑右移	int256	int256	int/lit8	—
simd_ctpopow	ctpopow	八倍字数 1	int	int256	—	—
simd_ctlzow	ctlzow	八倍字数头 0	int	int256	—	—
simd_vucaddw	vucaddw	字饱和加	intv8	intv8	intv8/lit8	—
simd_vucsubw	vucsubw	字饱和减	intv8	intv8	intv8/lit8	—
simd_vucaddh	vucaddh	半字饱和加	intv8	intv8	intv8/lit8	—
simd_vucsubh	vucsubh	半字饱和减	intv8	intv8	intv8/lit8	—
simd_vucaddb	vucaddb	字节饱和加	intv8	intv8	intv8/lit8	—
simd_vucsubb	vucsubb	字节饱和减	intv8	intv8	intv8/lit8	—
simd_vseleqw	vseleqw	等于比较选择	intv8	intv8	intv8	intv8/lit5

simd_vselltw	vselltw	小于比较选择	intv8	intv8	intv8	intv8/ lit5
simd_vsellew	vsellew	小于等于比较选择	intv8	intv8	intv8	intv8/ lit5
simd_vsellbcw	vsellbcw	低位为 0 比较选择	intv8	intv8	intv8	intv8/ lit5

内部函数	**指令**	**操作**	**参数说明**				
			VRe（返回值）	**VRav**	**VRbv**	**VRcv**	**VRdv**
simd_vlog	vlogzz	可重构逻辑	intv8	0xzz	intv8	intv8	intv8

说明：

a) VRav、VRbv、VRcv 分别为第一、第二、第三个参数，VRc 为返回值；可重构逻辑接口 VRav、VRbv、VRcv、VRdv 分别为第一、第二、第三、第四个参数，VRe 为返回值。

b) “—”表示对应接口的该参数无效

c) 比较指令、移位指令、字向量和长字向量加减法均支持运算符操作

d) simd_vlog 的 VRav 参数要求格式为 16 进制数，它指定 vlogzz 里的 zz 域

函数名

simd_vaddw —— intv8 的加法

对应指令

vaddw

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 //intv8 类型的加法

功能说明

intv8 类型的加法，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 相加。可以使用“+”符号替换。

函数名

simd_vsubw —— intv8 的减法

对应指令

vsubw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）；

返回值

[u]intv8 //intv8 类型的减法

功能说明

intv8 类型的减法，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 相减。可以使用“-”符号替换。

函数名

simd_vandw —— intv8 的逻辑与

对应指令

vandw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑与，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑与运算。可以使用“&”符号替换。

函数名

simd_vbicw —— intv8 的逻辑与非

对应指令

vbicw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑与非，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑与非运算。可以使用“&”和“~”符号替换。例如：

vc = simd_vbicw(va,vb);等价于：

vc = va & (~vb);

函数名

simd_vbisw —— intv8 的逻辑或

对应指令

vbisw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑或，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑或运算。可以使用“|”符号替换。

函数名

simd_vornotw —— intv8 的逻辑或非

对应指令

无

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑或非，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑与非运算。可以使用“|”和“~”符号替换。例如：

vc = simd_vornotw(va,vb);等价于：

vc = va | (~vb);

函数名

simd_vxorw —— intv8 的逻辑异或

对应指令

vxorw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑异或，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑异或运算。可以使用“^”符号替换。

函数名

simd_vsllw —— intv8 的逻辑左移

对应指令

vsllw

参数说明

[u]intv8 va;

int/lit5 b;

返回值

[u]intv8 //intv8 类型的逻辑左移

功能说明

intv8 类型的逻辑左移，将 va 中的 8 个 int 类型的整型数据进行逻辑左移，移出的空位用“0”补充，四个数移位的位数相同，由 b 中的最低 5 位或者立即数决定。可以使用“<<”符号替换。

函数名

simd_vsrlw —— intv8 的逻辑右移

对应指令

vsrlw

参数说明

[u]intv8 va;

int/lit5 b;

返回值

[u]intv8 //intv8 类型的逻辑右移

功能说明

intv8 类型的逻辑右移，将 va 中的 8 个 int 类型的整型数据进行逻辑右移，移出的空位用“0”补充，四个数移位的位数相同，由 b 中的最低 5 位或者 5 位立即数决定。可以使用“>>”符号替换。使用“>>”符号的时候，如果操作数是有符号的，生成 vsraw 指令，如果是无符号的，生成 vsrlw 指令。

函数名

simd_vsraw —— intv8 的算术右移

对应指令

vsraw

参数说明

[u]intv8 va;

int/lit5 b;

返回值

[u]intv8 //intv8 类型的算术右移

功能说明

intv8 类型的算术右移，将 va 中的 8 个 int 类型的整型数据进行逻辑右移，移出的空位用“0”补充，四个数移位的位数相同，由 b 中的最低 5 位或者 5 位立即数决定。可以使用“>>”符号替换。使用“>>”符号的时候，如果操作数是有符号的，生成 vsraw 指令，如果是无符号的，生成 vsrlw 指令。

函数名

simd_veqvw —— intv8 的逻辑等效

对应指令

veqvw

参数说明

intv8 va;

intv8 vb（int8 b）;

返回值

intv8

功能说明

intv8 类型的逻辑等效，将 va 中的 8 个 int 类型的整型数据和 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 分别对应按位逻辑等效（同或），结果保存到返回值中。

函数名

simd_vcmpgew —— intv8 的等于比较运算

对应指令

vcmpgew

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

int c

功能说明

对Va中8个32位的字整数向量元素和Vb中对应8个32位的字整数向量元素（或8位立即数# b）分别进行有符号的大于等于比较，只要有一个32位字整数元素满足比较条件，则c写入“1”，否则返回“0”。

函数名

simd_vcmpeqw —— intv8 的等于比较运算

对应指令

vcmpeqw

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

对Va中8个32位的字整数向量元素和Vb中对应8个32位的字整数向量元素（或8位立即数＃b）分别进行等于比较运算，若满足指定关系，则Vc中对应的字元素为“1”，否则为“0”。

函数名

simd_vcmplew —— intv8 的小于等于比较运算

对应指令

vcmplew

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

对Va中8个32位的字整数向量元素和Vb中对应8个32位的字整数向量元素（或8位立即数＃b）分别进行小于等于比较运算，若满足指定关系，则Vc中对应的字元素为“1”，否则为“0”。

函数名

simd_vcmpltw —— intv8 的小于比较运算

对应指令

vcmpltw

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

对Va中8个32位的字整数向量元素和Vb中对应8个32位的字整数向量元素（或8位立即数＃b）分别进行小于比较运算，若满足指定关系，则Vc中对应的字元素为“1”，否则为“0”。

函数名

simd_vcmpulew —— intv8 的无符号小于等于比较运算

对应指令

vcmpulew

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

对Va中8个32位的字整数向量元素和Vb中对应8个32位的字整数向量元素（或8位立即数# b）分别进行无符号小于等于比较运算，若满足指定关系，则Vc中对应的字元素为“1”，否则为“0”。

函数名

simd_vcmpultw —— intv8 的无符号小于比较运算

对应指令

vcmpultw

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

对Va中8个32位的字整数向量元素和Vb中对应8个32位的字整数向量元素（或8位立即数# b）分别进行无符号小于比较运算，若满足指定关系，则Vc中对应的字元素为“1”，否则为“0”。

函数名

simd_vrolw —— intv8 的循环左移

对应指令

vrolw

参数说明

[u]intv8 va;

int/lit5 b;

返回值

[u]intv8

功能说明

intv8 类型的循环左移，将 va 中的 8 个 int 类型的整型数据进行循环左移，移出的空位用移出位补充，四个数移位的位数相同，由 b 的低 5 位或 5 位立即数决定。

函数名

simd_sllow —— int256 的逻辑左移

对应指令

sllow

参数说明

[u]int256 va;

int b; （lit8 b）

返回值

[u]int256

功能说明

将 Va 中 256 位整体左移，移出的空位用“0”填充，移位位数由 b 低 8 位或 8 位立即数确定，结果写入 Vc 中。

函数名

simd_srlow —— int256 的逻辑右移

对应指令

srlow

参数说明

[u]int256 va;

int b; （lit8 b）

返回值

[u]int256

功能说明

将 Va 中 256 位整体右移，移出的空位用“0”填充，移位位数由 b 低 8 位或 8 位立即数确定，结果写入 Vc 中。

函数名

simd_vaddl —— 长字向量加法

对应指令

vaddl

参数说明

[u]int256 va;

[u]int256 vb(int8 b);

返回值

[u]int256

功能说明

将 Va 和 Vb 中 4 个 64 位长字（或 8 位立即数 b）分别对应相加，结果保存到 Vc 中对应位置。

函数名

simd_vsubl —— 长字向量减法

对应指令

vsubl

参数说明

[u]int256 va;

[u]int256 vb(int8 b);

返回值

[u]int256

功能说明

将 Va 和 Vb 中 4 个 64 位长字（或 8 位立即数 b）分别对应相减，结果保存到 Vc 中对应位置。

函数名

simd_ctpopow—— int256 的数 1

对应指令

ctpopow

参数说明

[u]int256 va;

返回值

int vc

功能说明

返回 va 中的 256 位八倍字整数中“1”的个数。

函数名

simd_ctlzow—— int256 的数头 0

对应指令

ctlzow

参数说明

[u]int256 va;

返回值

int vc

功能说明

返回 Va 中的 256 位八倍字整数从最高位起连续“0”的个数。

函数名

simd_vucaddw —— intv8 的饱和加运算

对应指令

vucaddw

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

将 Va 和 Vb 中对应的 8 个 32 位字整数（或 8 位立即数#b 零扩展的字整数）分别进行 32 位的有符号加，取低 32 位结果保存到 Vc。若结果产生上溢，则将对应的 32 位结果置为 0x7fff,ffff；若产生下溢，则将对应的 32 位结果置为 0x8000,0000。

函数名

simd_vucsubw —— intv8 的饱和减运算

对应指令

vucsubw

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

将 Va 和 Vb 中对应的 8 个 32 位字整数（或 8 位立即数#b 零扩展的字整数）分别进行 32 位的有符号减，取低 32 位结果保存到 Vc。若结果产生上溢，则将对应的 32 位结果置为 0x7fff,ffff；若产生下溢，则将对应的 32 位结果置为 0x8000,0000。

函数名

simd_vucaddh —— 有符号半字向量的饱和加运算

对应指令

vucaddh

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

将 Va 和 Vb 中对应的 16 个 16 位半字整数（或 8 位立即数#b 零扩展的半字整数）分别进行 16 位的有符号加，取低 16 位结果保存到 Vc。若结果产生上溢，则将对应的 16 位结果置为 0x7fff；若产生下溢，则将对应的 16 位结果置为 0x8000。

函数名

simd_vucsubh —— 有符号半字向量的饱和减运算

对应指令

vucsubh

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

将 Va 和 Vb 中对应的 16 个 16 位半字整数（或 8 位立即数#b 零扩展的半字整数）分别进行 16 位的有符号减，取低 16 位结果保存到 Vc。若结果产生上溢，则将对应的 16 位结果置为 0x7fff；若产生下溢，则将对应的 16 位结果置为 0x8000。

函数名

simd_vucaddb —— 有符号字节向量的饱和加运算

对应指令

vucaddb

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

将 Va 和 Vb 中对应的 32 个 8 位字节整数(或 8 位立即数#b)分别进行 8 位的有符号加，取低 8 位结果保存到 Vc。若结果产生上溢，则将对应的 8 位结果置为 0x7f；若产生下溢，则将对应的 8 位结果置为 0x80。

函数名

simd_vucsubb —— 有符号字节向量的饱和减运算

对应指令

vucsubb

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 vc

功能说明

将 Va 和 Vb 中对应的 32 个 8 位字节整数(或 8 位立即数#b)分别进行 8 位的有符号减，取低 8 位结果保存到 Vc。若结果产生上溢，则将对应的 8 位结果置为 0x7f；若产生下溢，则将对应的 8 位结果置为 0x80。

函数名

simd_vseleqw ——intv8 的等于选择运算

对应指令

vseleqw

参数说明

[u]intv8 va;

[u]intv8 vb;

[u]intv8 vc(int5 b);

返回值

[u]intv8 vd

功能说明

对 Va 中每个字元素与 0 进行等于比较测试，若条件成立，则将向量 Vb 对应的字元素写入 Vd 的对应位置，否则将向量 Vc 对应的字元素（或“0”扩展 5 位立即数#c 形成的字整数）写入 Vd 的对应位置。

函数名

simd_vsellew ——intv8 的小于等于选择运算

对应指令

vsellew

参数说明

[u]intv8 va;

[u]intv8 vb;

[u]intv8 vc(int5 b);

返回值

[u]intv8 vd

功能说明

对 Va 中每个字元素与 0 进行小于等于比较测试，若条件成立，则将向量 Vb 对应的字元素写入 Vd 的对应位置，否则将向量 Vc 对应的字元素（或“0”扩展 5 位立即数#c 形成的字整数）写入 Vd 的对应位置。

函数名

simd_vselltw ——intv8 的小于选择运算

对应指令

vselltw

参数说明

[u]intv8 va;

[u]intv8 vb;

[u]intv8 vc(int5 b);

返回值

[u]intv8 vd

功能说明

对 Va 中每个字元素与 0 进行小于比较测试，若条件成立，则将向量 Vb 对应的字元素写入 Vd 的对应位置，否则将向量 Vc 对应的字元素（或“0”扩展 5 位立即数#c 形成的字整数）写入 Vd 的对应位置。

函数名

simd_vsellbcw ——intv8 的低位为 0 选择运算

对应指令

vsellbcw

参数说明

[u]intv8 va;

[u]intv8 vb;

[u]intv8 vc(int5 b);

返回值

[u]intv8 vd

功能说明

对 Va 中每个字元素的低位进行测试，若为 0，则将向量 Vb 对应的字元素写入 Vd 的对应位置，否则将向量 Vc 对应的字元素（或“0”扩展 5 位立即数#c 形成的字整数）写入 Vd 的对应位置。

函数名

simd_vlog —— intv8 的可重构逻辑运算

对应指令

vlogzz

参数说明

zz;

[u]intv8 va;

[u]intv8 vb;

[u]intv8 vc;

返回值

[u]intv8 vd

功能说明

将 Va、Vb 和 Vc 中对应的每一位，组成 3 位的二进制数，从两位 16 进制参数 zz 的 8 位二进制值中选出一位，写入 Vd 的对应位中。

8.5.4 浮点向量运算操作函数

内部函数	指令	操作	参数说明			
			VRc（返回值）	VRav	VRbv	VRcv

simd_vadds	vadds	+	floatv4	floatv4	floatv4	/
simd_vsubs	vsubs	-	floatv4	floatv4	floatv4	/
simd_vmuls	vmuls	*	floatv4	floatv4	floatv4	/
simd_vmas	vmas	乘加	floatv4	floatv4	floatv4	floatv4
simd_vmss	vmss	乘减	floatv4	floatv4	floatv4	floatv4
simd_vnmas	vnmas	负乘加	floatv4	floatv4	floatv4	floatv4
simd_vnmss	vnmss	负乘减	floatv4	floatv4	floatv4	floatv4
simd_vaddd	vaddd	+	doublev4	doublev4	doublev4	/
simd_vsubd	vsubd	-	doublev4	doublev4	doublev4	/
simd_vmuld	vmuld	*	doublev4	doublev4	doublev4	/
simd_vmad	vmad	乘加	doublev4	doublev4	doublev4	doublev4
simd_vmsd	vmsd	乘减	doublev4	doublev4	doublev4	doublev4
simd_vnmad	vnmad	负乘加	doublev4	doublev4	doublev4	doublev4
simd_vnmsd	vnmsd	负乘减	doublev4	doublev4	doublev4	doublev4
simd_vseleq	vseleq	条件选择	扩展浮点	扩展浮点	扩展浮点	扩展浮点
simd_vsellt	vsellt		扩展浮点	扩展浮点	扩展浮点	扩展浮点
simd_vselle	vselle		扩展浮点	扩展浮点	扩展浮点	扩展浮点
simd_vcpys	vcpys	符号拷贝	扩展浮点	扩展浮点	扩展浮点	/
simd_vcpyse	vcpyse		扩展浮点	扩展浮点	扩展浮点	/
simd_vcpysn	vcpysn		扩展浮点	扩展浮点	扩展浮点	/
simd_vdivs	vdivs	除法	floatv4	floatv4	floatv4	—
simd_vdivd	vdivd	除法	doublev4	doublev4	doublev4	—
simd_vsqrts	vsqrts	求平方根	floatv4	floatv4	—	—
simd_vsqrtd	vsqrtd	求平方根	doublev4	doublev4	—	—
simd_vfcmpeq	vfcmpeq	等于比较	floatv4/doublev4	floatv4/doublev4	floatv4/doublev4	—
simd_vfcmple	vfcmple	小于等于比较	floatv4/doublev4	floatv4/doublev4	floatv4/doublev4	—
simd_vfcmplt	vfcmplt	小于比较	floatv4/doublev4	floatv4/doublev4	floatv4/doublev4	—
simd_vfcmpun	vfcmpun	无序比较	floatv4/doublev4	floatv4/doublev4	floatv4/doublev4	—

说明：

a) VRav、VRbv、VRcv 分别为第一、第二、第三个参数，VRc 为返回值；

b) “—”表示对应接口的该参数无效

c) 加法、减法、乘法、除法、乘加类运算支持运算符操作

函数名

simd_vaddd —— doublev4 的加法运算

对应指令

vaddd

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的加法运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数进行加法运算。可以用“+”符号替换。

函数名

simd_vadds —— floatv4 的加法运算

对应指令

vadds

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的加法运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点数进行加法运算。可以用“+”符号替换。

函数名

simd_vsubd —— doublev4 的减法运算

对应指令

vsubd

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的减法运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数进行减法运算。可以用“-”符号替换。

函数名

simd_vsubs —— floatv4 的减法运算

对应指令

vsubs

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的减法运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点数进行减法运算。可以用“-”符号替换。

函数名

simd_vmuld —— doublev4 的加法运算

对应指令

vmuld

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的乘法运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数进行乘法运算。可以用“*”符号替换。

函数名

simd_vmuls —— floatv4 的加法运算

对应指令

vmuls

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的乘法运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点

数进行乘法运算。可以用“*”符号替换。

函数名

simd_vmad —— doublev4 的乘加运算

对应指令

vmad

参数说明

doublev4 va;

doublev4 vb;

doublev4 vc;

返回值

doublev4

功能说明

doublev4 类型的乘加运算，将 va 中的 4 个 double 浮点数和 vb 中的 4 个 double 浮点数以及 vc 中的 4 个浮点数分别进行乘加运算。可以用“*”和“+”符号替换。

ret = va*vb + vc;

函数名

simd_vmas —— floatv4 的乘加运算

对应指令

vmas

参数说明

floatv4 va;

floatv4 vb;

floatv4 vc;

返回值

floatv4

功能说明

floatv4 类型的乘加运算，将 va 中的 4 个 float 浮点数和 vb 中的 4 个 float 浮点数以及 vc 中的 4 个浮点数分别进行乘加运算。可以用“*”和“+”符号替换。

ret = va*vb + vc;

函数名

simd_vmsd —— doublev4 的乘减运算

对应指令

vfmsd

参数说明

doublev4 va;

doublev4 vb;

doublev4 vc;

返回值

doublev4

功能说明

doublev4 类型的乘减运算，将 va 中的 4 个 double 浮点数和 vb 中的 4 个 double 浮点数以及 vc 中的 4 个浮点数分别进行乘减运算。可以用“*”和“-”符号替换。

ret=va*vb-vc；

函数名

simd_vmss —— floatv4 的乘减运算

对应指令

vmss

参数说明

floatv4 va;

floatv4 vb;

floatv4 vc;

返回值

floatv4

功能说明

floatv4 类型的乘减运算，将 va 中的 4 个浮点数和 vb 中的 4 个浮点数以及 vc 中的 4 个浮点数分别进行乘减运算。可以用“*”和“-”符号替换。

ret=va*vb-vc;

函数名

simd_vnmad —— doublev4 的负乘加运算

对应指令

vnmad

参数说明

doublev4 va;

doublev4 vb;

doublev4 vc;

返回值

doublev4

功能说明

doublev4 类型的负乘加运算，将 va 中的 4 个 double 浮点数和 vb 中的 4 个 double 浮点数以及 vc 中的 4 个浮点数分别进行负乘加运算。可以用“*”和“+”和“-”符号替换。

ret = -va*vb + vc；

函数名

simd_vnmas —— floatv4 的负乘加运算

对应指令

vnmas

参数说明

floatv4 va;

floatv4 vb;

floatv4 vc;

返回值

floatv4

功能说明

floatv4 类型的负乘加运算，将 va 中的 4 个 float 浮点数和 vb 中的 4 个 float 浮点数以及 vc 中的 4 个浮点数分别进行负乘加运算。可以用“*”和“+”和“-”符号替换。

ret = -va*vb + vc;

函数名

simd_vnmsd —— doublev4 的负乘减运算

对应指令

vnmsd

参数说明

doublev4 va;

doublev4 vb;

doublev4 vc;

返回值

doublev4

功能说明

doublev4 类型的负乘减运算，将 va 中的 4 个 double 浮点数和 vb 中的 4 个 double 浮点数以及 vc 中的 4 个浮点数分别进行负乘减运算。可以用“*”和“+”和“-”符号替换。

ret = -va*vb - vc;

函数名

simd_vnmss —— floatv4 的负乘减运算

对应指令

vnmss

参数说明

floatv4 va;

floatv4 vb;

floatv4 vc;

返回值

floatv4

功能说明

floatv4 类型的负乘减运算，将 va 中的 4 个 float 浮点数和 vb 中的 4 个 float 浮点数以及 vc 中的 4 个浮点数分别进行负乘减运算。可以用“*”和“+”和“-”符号替换。

ret = -va*vb - vc；

函数名

simd_vseleq —— doublev4/floatv4 的等于选择运算

对应指令

vseleq

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

doublev4/floatv4 vc;

返回值

doublev4/floatv4

功能说明

判断 doublev4/floatv4 类型的参数 va，如果等于 0，则返回 vb 的值；否则返回 vc 的值。

函数名

simd_vsellt —— doublev4/floatv4 的小于选择运算

对应指令

vsellt

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

doublev4/floatv4 vc;

返回值

doublev4/floatv4

功能说明

判断 doublev4/floatv4 类型的参数 va，如果小于 0，则返回 vb 的值；否则返回 vc 的值。

函数名

simd_vselle —— doublev4/floatv4 的小于等于选择运算

对应指令

vselle

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

doublev4/floatv4 vc;

返回值

doublev4/floatv4

功能说明

判断 doublev4/floatv4 类型的参数 va，如果小于等于 0，则返回 vb 的值；否则返回 vc 的值。

函数名

simd_vcpys —— floatv4/doublev4 的拷贝符号

对应指令

vcpys

参数说明

floatv4 va;[doublev4 va;]

floatv4 vb;[doublev4 vb;]

返回值

floatv4 [doublev4]

功能说明

浮点拷贝符号，返回值中的符号位是 va 的符号位，其余部分为 vb 中的数据。

函数名

simd_vcpyse —— floatv4/doublev4 的拷贝符号和指数

对应指令

vcpyse

参数说明

floatv4 va;[doublev4 va;]

floatv4 vb;[doublev4 vb;]

返回值

floatv4 [doublev4]

功能说明

浮点拷贝符号和指数，返回值中的符号位和指数位是 va 的数据，其余部分为 vb 中的数据。

函数名

simd_vcpysn —— floatv4/doublev4 的拷贝符号反码

对应指令

vcpysn

参数说明

floatv4 va;[doublev4 va;]

floatv4 vb;[doublev4 vb;]

返回值

floatv4 [doublev4]

功能说明

浮点拷贝符号反码，返回值中的符号位是 va 的符号位的反码，其余部分为 vb 中的数据。

函数名

simd_vdivd —— doublev4 的除法运算

对应指令

vdivd

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的除法运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数进行除法运算。

函数名

simd_vdivs —— floatv4 的除法运算

对应指令

vdivs

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的乘法运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点数进行除法运算。

函数名

simd_vsqrtd —— doublev4 的求平方根运算

对应指令

vsqrtd

参数说明

doublev4 va;

返回值

doublev4

功能说明

doublev4 类型的求平方根运算，将 va 中的 4 个 double 浮点数分别求平方根。

函数名

simd_vsqrts —— floatv4 的求平方根运算

对应指令

vsqrts

参数说明

floatv4 va;

返回值

floatv4

功能说明

floatv4 类型的求平方根运算，将 va 中的 4 个 float 浮点数分别求平方根。

函数名

simd_vfcmpeq —— doublev4/floatv4 的等于比较运算

对应指令

vfcmpeq

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4 vc

功能说明

将 Va 和 Vb 中的对应浮点向量元素进行等于比较，如果条件成立，则将非“0”浮点值（2.0）写入 vc 的对应位置，否则将真“0”写入 vc 的对应位置。

函数名

simd_vfcmple —— doublev4/floatv4 的小于等于比较运算

对应指令

vfcmple

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4 vc

功能说明

将 Va 和 Vb 中的对应浮点向量元素进行小于等于比较，如果条件成立，则将非“0”浮点值（2.0）写入 vc 的对应位置，否则将真“0”写入 vc 的对应位置。

函数名

simd_vfcmplt —— doublev4/floatv4 的小于比较运算

对应指令

vfcmplt

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4 vc

功能说明

将 Va 和 Vb 中的对应浮点向量元素进行小于比较，如果条件成立，则将非“0”浮点值(2.0)写入 vc 的对应位置，否则将真“0”写入 vc 的对应位置。

函数名

simd_vfcmpun —— doublev4/floatv4 的无序比较运算

对应指令

vfcmpun

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4 vc

功能说明

将 Va 和 Vb 中的对应浮点向量元素进行无序比较，如果条件成立，则将非“0”浮点值(2.0)写入 vc 的对应位置，否则将真“0”写入 vc 的对应位置。

8.5.5 SIMD 整理指令

内部函数	指令	操作	参数说明			
			VRc（返回值）	**VRav**	**VRbv**	**VRcv**
simd_vinswn	vinsw	字向量插入	intv8	intv8	intv8	/

n=(0,1,…,7)						
simd_vinsfn n=(0,1,2,3)	vinsf	浮点向量插入	floatv4/doublev4	floatv4/ doublev4	floatv4/ doublev4	/
simd_vextwn n=(0,1,…,7)	vextw	字向量提取	intv8	intv8	/	/
simd_vextfn n=(0,1,2,3)	vextf	浮点向量提取	floatv4/doublev4	floatv4/ doublev4	/	/
simd_vcpyw	vcpyw	字向量拷贝	intv8	floatv4/ doublev4	/	/
simd_vcpyf	vcpyf	浮点向量拷贝	floatv4/doublev4	floatv4/ doublev4	/	/

说明：

a)　simd_vinswn 和 simd_vextwn 中 n 的取值只能是 0、1、2、3、4、5、6、7；simd_vinsfn 和 simd_vextfn 中 n 的取值只能是 0、1、2、3。

函数名

simd_vinswn—— 字向量元素插入运算

对应指令

vinsw

参数说明

intv8 va;

intv8 vb;

返回值

intv8

功能说明

将浮点寄存器va低64位的寄存器格式字整数转换为32位字整数，替代vb中由n（有效值为0～7）指定32位字元素，组成并返回新的字整数向量。

该接口实际上对应8个接口，分别是simd_vinsw0、simd_vinsw1、simd_vinsw2、simd_vinsw3、simd_vinsw4、simd_vinsw5、simd_vinsw6、simd_vinsw7，比如：

vc=simd_vinsw2

表示va低64位的寄存器格式字整数先转换为32位字整数，然后替代vb的64：95这32位的字元素，组成新的vb返回给vc。

函数名

simd_vinsfn—— 浮点向量元素插入运算

对应指令

vinsf

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4

功能说明

将浮点寄存器 va 低 64 位浮点数据替代 vb 中由 n（有效值为 0～3）指定的 64 位浮点元素，组成并返回新的浮点向量。

该接口实际上对应4个接口，分别是simd_vinsf0、simd_vinsf1、simd_vinsf2、simd_vinsf3，比如：

vc=simd_vinsf1(va,vb)

表示 va 低 64 位浮点数据替代 vb 的 64：127 这 64 位的浮点元素，组成新的 vb 返回给 vc。

函数名

simd_vextwn—— 字向量元素提取运算

对应指令

vextw

参数说明

intv8 va;

返回值

intv8

功能说明

将 va 中由 n（有效值为 0～7）指定的 32 位字元素，变为 64 位寄存器格式的字整数写入目的浮点寄存器的低 64 位，目的寄存器的高 192 位内容不确定。

该接口实际上对应8个接口，分别是simd_vextw0、simd_vextw1、simd_vextw2、simd_vextw3、simd_vextw4、simd_vextw5、simd_vextw6、simd_vextw7，比如：

vc=simd_vextw2(va)

表示 va 的 64：95 这 32 位的字元素，变为 64 位寄存器格式的字整数写入 vc 低 64 位，其中 vc 高 192 位内容不确定。

函数名

simd_vextfn—— 浮点向量元素提取运算

对应指令

vextf

参数说明

doublev4/floatv4 va;

返回值

doublev4/floatv4

功能说明

将 va 中由 n(有效值为 0～3)指定的 S 浮点或 D 浮点元素存入目的寄存器的<63:0>中。此指令可实现 S 浮点和 D 浮点的提取。

该接口实际上对应 4 个接口，分别是 simd_vextf0、simd_vextf1、simd_vextf2、simd_vextf3，比如：

vc=simd_vextf1(va)

表示 va 的 64：127 这 64 位的浮点元素存入 vc 的<63:0>中。

函数名

simd_vcpyw—— 字向量元素拷贝运算

对应指令

vcpyw

参数说明

doublev4/floatv4 va;

返回值

intv8

功能说明

将 va 低 64 位的寄存器格式字整数转换位 32 位字整数，复制成 8 个相同元素组成新的字整数向量写入目的寄存器。

函数名

simd_vcpyf—— 浮点向量元素拷贝运算

对应指令

vcpyf

参数说明

doublev4/floatv4 va;

返回值

doublev4/floatv4

功能说明

将 va 低 64 位浮点数据，复制成 4 个相同元素，组成新的浮点向量写入目的寄存器。

8.5.6 赋值运算操作宏定义

宏定义	指令	操作	参数说明	
			返回值	参数
simd_set_intv8	/	赋值	32*8 向量类型	8 个 int 类型数
simd_set_uintv8	/	赋值	32*8 向量类型	8 个 unsigned int 类型数
simd_set_int256	/	赋值	256 位八倍字	4 个 long

simd_set_uint256	/	赋值	256 位八倍字	4 个 unsigned long
simd_set_floatv4	/	赋值	floatv4 类型	4 个单精浮点
simd_set_doublev4	/	赋值	doublev4 类型	4 个双精浮点

函数名

simd_set_doublev4—— doublev4 赋值函数

对应指令

无

参数说明

double a;

double b;

double c;

double d;

返回值

doublev4

功能说明

doublev4 类型的赋值函数，将 4 个 double 类型的数据传递到向量变量中。

例如：doublev4 va=simd_set_doublev4(1.0，2.0，3.0，4.0);

va 中的值是这样的：

4.0	3.0	2.0	1.0

255 0

函数名

simd_set_floatv4 —— floatv4 赋值函数

对应指令

无

参数说明

float a;

float b;

float c;

float d;

返回值

floatv4

功能说明

floatv4 类型的赋值函数，将 4 个 float 类型的数据传递到向量变量中。

例如：floatv4 va=simd_set_floatv4(1.0，2.0，3.0，4.0);

va 中的值是这样的：

4.0	3.0	2.0	1.0

255 0

函数名

simd_set_intv8 —— intv8 赋值函数

对应指令

无

参数说明

int a;

int b;

int c;

int d;

int e;

int f;

int g;

int h;

返回值

intv8

功能说明

intv8 类型的赋值函数，将 8 个 int 类型的数据传递到向量变量中。

例如：va=simd_set_intv8(1,2,3,4,5,6,7,8);

va 中的值是这样的：

8	7	6	5	4	3	2	1

255 0

函数名

simd_set_uintv8 —— uintv8 赋值函数

对应指令

无

参数说明

unsigned int a;

unsigned int b;

unsigned int c;

unsigned int d;

unsigned int e;

unsigned int f;

unsigned int g;

unsigned int h;

返回值

uintv8

功能说明

uintv8 类型的赋值函数，将 8 个 unsigned int 类型的数据传递到向量变量中。

例如：va=simd_set_uintv8(1,2,3,4,5,6,7,8);

va 中的值是这样的：

8	7	6	5	4	3	2	1

255 0

函数名

simd_set_int256 —— int256 赋值函数

对应指令

无

参数说明

long a;

long b;

long c;

long d;

返回值

int256

功能说明

int256 类型的赋值函数，将 4 个 long 类型的数据传递到向量变量中。

例如：va=simd_set_int256(1,2,3,4);

va 中的值是这样的：

4	3	2	1

255 0

函数名

simd_set_uint256 —— uint256 赋值函数

对应指令

无

参数说明

unsigned long a;

unsigned long b;

unsigned long c;

unsigned long d;

返回值

uint256

功能说明

uint256 类型的赋值函数，将 4 个 unsigned long 类型的数据传递到向量变量中。

例如：va=simd_set_uint256(1,2,3,4);

va 中的值是这样的：

4	3	2	1

255 0

8.5.7 扩展类型的打印函数

函数	指令	操作	参数说明	
			返回值	参数
simd_fprint_intv8	/	打印	/	FILE*, intv8
simd_fprint_uintv8	/	打印	/	FILE *, uint8
simd_fprint_int256	/	打印	/	FILE*, int256
simd_fprint_uint256	/	打印	/	FILE *, uint256
simd_fprint_floatv4	/	打印	/	FILE*, floatv4
simd_fprint_doublev4	/	打印	/	FILE*, doublev4
simd_print_intv8	/	打印	/	intv8
simd_print_uintv8	/	打印	/	uintv8
simd_print_int256	/	打印	/	int256
simd_print_uint256	/	打印	/	uint256
simd_print_floatv4	/	打印	/	floatv4
simd_print_doublev4	/	打印	/	doublev4

函数名

simd_fprint_intv8 —— intv8 打印到文件

对应指令

无

参数说明

FILE *file;

intv8 va;

返回值

无

功能说明

intv8 类型的数据以 8 个 int 的形式打印到文件 file 中。

例如：   va=simd_set_intv8(1,2,3,4,5,6,7,8);

simd_fprint_intv8(stderr, va);

打印的结果是这样的：

[8，7，6，5，4, 3, 2, 1]

函数名

simd_fprint_uintv8 —— uintv8 打印到文件

对应指令

无

参数说明

FILE *file;

uintv8 va;

返回值

无

功能说明

uintv8 类型的数据以 8 个 int 的形式打印到文件 file 中。

例如：   va=simd_set_uintv8(1,2,3,4,5,6,7,8);

simd_fprint_uintv8(stderr, va);

打印的结果是这样的：

[8，7，6，5，4, 3, 2, 1]

函数名

simd_fprint_int256 —— int256 打印到文件

对应指令

无

参数说明

FILE *file;

int256 va;

返回值

无

功能说明

int256 类型的数据以 4 个 long 的形式打印到文件 file 中。

例如：   va=simd_set_int256(1,2,3,4);

simd_fprint_int256(stderr, va);

打印的结果是这样的：

[0x4, 0x3, 0x2, 0x1]

函数名

simd_fprint_uint256—— uint256 打印到文件

对应指令

无

参数说明

FILE *file;

uint256 va;

返回值

无

功能说明

uint256 类型的数据以 4 个 long 的形式打印到文件 file 中。

例如： va=simd_set_uint256(1,2,3,4);

simd_fprint_uint256(stderr, va);

打印的结果是这样的：

[0x4, 0x3, 0x2, 0x1]

函数名

simd_fprint_floatv4 —— floatv4 打印到文件

对应指令

无

参数说明

FILE *file;

floatv4 va;

返回值

无

功能说明

floatv4 类型的数据以 4 个 float 的形式打印到文件 file 中。

例如： va=simd_set_floatv4(1.0,2.0,3.0,4.0);

simd_fprint_floatv4(stderr, va);

打印的结果是这样的：

[3.0， 4.0， 2.0, 1.0]

函数名

simd_fprint_doublev4 —— doublev4 打印到文件

对应指令

无

参数说明

FILE *file;

doublev4 va;

返回值

无

功能说明

doublev4 类型的数据以 4 个 double 的形式打印到文件 file 中。

例如： va=simd_set_doublev4(1.0,2.0,3.0,4.0);

simd_fprint_doublev4(stderr, va);

打印的结果是这样的：

[4.0，3.0，2.0, 1.0]

函数名

simd_print_intv8 —— intv8 打印到屏幕

对应指令

无

参数说明

intv8 va;

返回值

无

功能说明

intv8 类型的数据以 8 个 int 的形式打印到屏幕。

例如： va=simd_set_intv8(1,2,3,4,5,6,7,8);

simd_print_intv8(va);

打印的结果是这样的：

[8，7，6，5，4, 3, 2, 1]

函数名

simd_print_uintv8 —— uintv8 打印到屏幕

对应指令

无

参数说明

uintv8 va;

返回值

无

功能说明

uintv8 类型的数据以 8 个 int 的形式打印到屏幕。

例如： va=simd_set_uintv8(1,2,3,4,5,6,7,8);

simd_print_uintv8(va);

打印的结果是这样的：

[8， 7， 6， 5， 4, 3, 2, 1]

函数名

simd_print_int256 —— int256 打印到屏幕

对应指令

无

参数说明

int256 va;

返回值

无

功能说明

int256 类型的数据以 4 个 long 的形式打印到屏幕。

例如： va=simd_set_int256(1,2,3,4);

simd_print_int256(va);

打印的结果是这样的：

[0x4, 0x3, 0x2, 0x1]

函数名

simd_print_uint256—— uint256 打印到屏幕

对应指令

无

参数说明

uint256 va;

返回值

无

功能说明

uint256 类型的数据以 4 个 long 的形式打印到屏幕。

例如： va=simd_set_uint256(1,2,3,4);

simd_print_uint256(va);

打印的结果是这样的：

[0x4, 0x3, 0x2, 0x1]

函数名

simd_print_floatv4 —— floatv4 打印到屏幕

对应指令

无

参数说明

floatv4 va;

返回值

无

功能说明

floatv4 类型的数据以 4 个 float 的形式打印到屏幕。

例如： va=simd_set_floatv4(1.0,2.0,3.0,4.0);

simd_print_floatv4(va);

打印的结果是这样的：

[3.0， 4.0， 2.0, 1.0]

函数名

simd_print_doublev4 —— doublev4 打印到屏幕

对应指令

无

参数说明

doublev4 va;

返回值

无

功能说明

doublev4 类型的数据以 4 个 double 的形式打印到屏幕。

例如： va=simd_set_doublev4(1.0,2.0,3.0,4.0);

simd_print_doublev4(va);

打印的结果是这样的：

[4.0， 3.0， 2.0, 1.0]

8.6 运算核心扩展内部函数、宏定义库

8.6.1 使用的符号说明

VRav：第一个向量参数

VRbv：第二个向量参数

VRcv：第三个向量参数

VRc：向量返回值

Rbv：无符号长整型变量，一般用于表示地址

Va：一段内存地址中的数据 f

fp：FILE*类型的变量

8.6.2 装入/存储操作宏定义

同运算控制核心的装入/存储操作宏定义，见 8.5.2。

8.6.3 定点向量运算函数接口

整数SIMD运算指令实现8个32位整数向量的加减、移位运算，包括采用简单运算指令格式的8条指令，每条指令都有寄存器格式和立即数格式。

序号	指 令	寄存器格式操 作 码	立即数格式操作码	描 述
1	vaddw Va, Vb/#b, Vc	0x1a.00	0x1a.20	字向量加
2	vsubw Va, Vb/#b, Vc	0x1a.01	0x1a.21	字向量减
3	vsllw Va, Vb/#b, Vc	0x1a.08	0x1a.28	字向量逻辑左移
4	vsrlw Va, Vb/#b, Vc	0x1a.09	0x1a.29	字向量逻辑右移
5	vsraw Va, Vb/#b, Vc	0x1a.0a	0x1a.2a	字向量算术右移
6	vrotlw Va, Vb/#b, Vc	0x1a.0b	0x1a.2b	字向量循环左移
7	vaddl Va, Vb/#b, Vc	0x1a.0e	0x1a.2e	长字向量加
8	vsubl Va, Vb/#b, Vc	0x1a.0f	0x1a.2f	长字向量减

函数列表：

内部函数	指令	操作	参数说明		
			返回值	VRav	VRbv
simd_vaddw	vaddw	+	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vsubw	vsubw	-	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vandw	vlog2x	&	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vbicw	vlog2x	与非	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vbisw	vlog2x	|	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vornotw	vlog2x	或非	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vxorw	vlog2x	^	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_veqvw	vlog2x	等效	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vrotlw	vrotlw	循环左移	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vsllw	vsllw	<<	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vsrlw	vsrlw	>>	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vsraw	vsraw	算术右移	32*8 向量类型	32*8 向量类型	32*8 向量类型
simd_vaddl	vaddl	+	64*4 向量类型	64*4 向量类型	64*4 向量类型
simd_vsubl	vsubl	-	64*4 向量类型	64*4 向量类型	64*4 向量类型
simd_sllow	sllow	逻辑左移	int256	int256	int/lit8
simd_srlow	srlow	逻辑右移	int256	int256	int/lit8

说明：

a) 参数 VRav 和 VRbv 可以是 intv8 或者 uintv8 类型。

b) 在三个移位运算（vsllw，vsrlw，vsraw）中，参数 VRvb 可以是 0-31 的整型常数；除此之外，其余指令的参数 VRvb 都可以是 8bit 的立即数，表示 VRav 的每个 32

位都和该立即数进行相应的操作。

c) 当 VRav 是无符号类型时，simd_vsral 等价于 simd_vsrll。

d) 在编程时，simd_vaddw、simd_vsubw、simd_vsllw、simd_vsrlw、simd_vandw、simd_vbisw、simd_vxorw、simd_vaddl、simd_vsubl 都可以用它们对应的运算符表示，不必调用内部函数。

函数名

simd_vaddw —— intv8 的加法

对应指令

vaddw

参数说明

[u]intv8 va;

[u]intv8 vb(int8 b);

返回值

[u]intv8 //intv8 类型的加法

功能说明

intv8 类型的加法，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 相加。可以使用“+”符号替换。

函数名

simd_vsubw —— intv8 的减法

对应指令

vsubw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8 //intv8 类型的减法

功能说明

intv8 类型的减法，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 相减。可以使用“-”符号替换。

函数名

simd_vandw —— intv8 的逻辑与

对应指令

vandw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑与，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑与运算。可以使用“&”符号替换。

函数名

simd_vbicw —— intv8 的逻辑与非

对应指令

vbicw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑与非，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑与非运算。可以使用“&”和“~”符号替换。例如：

vc = simd_vbicw(va,vb);等价于：

vc = va & (~vb);

函数名

simd_vbisw —— intv8 的逻辑或

对应指令

vbisw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑或，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑或运算。可以使用“|”符号替换。

函数名

simd_vornotw —— intv8 的逻辑或非

对应指令

无

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑或非，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑与非运算。可以使用“|”和“~”符号替换。例如：

vc = simd_vornotw(va,vb);等价于：

vc = va | (~vb);

函数名

simd_vxorw —— intv8 的逻辑异或

对应指令

vxorw

参数说明

[u]intv8 va;

[u]intv8 vb（int8 b）;

返回值

[u]intv8

功能说明

intv8 类型的逻辑异或，将 va 中的 8 个 int 类型的整型数据分别与 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 进行逻辑异或运算。可以使用“^”符号替换。

函数名

simd_vsllw —— intv8 的逻辑左移

对应指令

vsllw

参数说明

[u]intv8 va;

int/lit5 b;

返回值

[u]intv8 //intv8 类型的逻辑左移

功能说明

intv8 类型的逻辑左移，将 va 中的 8 个 int 类型的整型数据进行逻辑左移，移出的空位用“0”补充，四个数移位的位数相同，由 b 中的最低 5 位或者立即数决定。可以使用“<<”符号替换。

函数名

simd_vsrlw —— intv8 的逻辑右移

对应指令

vsrlw

参数说明

[u]intv8 va;

int/lit5 b;

返回值

[u]intv8 //intv8 类型的逻辑右移

功能说明

intv8 类型的逻辑右移，将 va 中的 8 个 int 类型的整型数据进行逻辑右移，移出的空位用"0"补充，四个数移位的位数相同，由 b 中的最低 5 位或者 5 位立即数决定。可以使用">>"符号替换。使用">>"符号的时候，如果操作数是有符号的，生成 vsraw 指令，如果是无符号的，生成 vsrlw 指令。

函数名

simd_vsraw —— intv8 的算术右移

对应指令

vsraw

参数说明

[u]intv8 va;

int/lit5 b;

返回值

[u]intv8 //intv8 类型的算术右移

功能说明

intv8 类型的算术右移，将 va 中的 8 个 int 类型的整型数据进行逻辑右移，移出的空位用"0"补充，四个数移位的位数相同，由 b 中的最低 5 位或者 5 位立即数决定。可以使用">>"符号替换。使用">>"符号的时候，如果操作数是有符号的，生成 vsraw 指令，如果是无符号的，生成 vsrlw 指令。

函数名

simd_veqvw —— intv8 的逻辑等效

对应指令

veqvw

参数说明

intv8 va;

intv8 vb（int8 b）;

返回值

intv8

功能说明

intv8 类型的逻辑等效，将 va 中的 8 个 int 类型的整型数据和 vb 中的 8 个 int 类型的整型数据或 8 位立即数 b 分别对应按位逻辑等效（同或），结果保存到返回值中。

函数名

simd_vrolw —— intv8 的循环左移

对应指令

vrolw

参数说明

[u]intv8 va;

int/lit5 b;

返回值

[u]intv8

功能说明

intv8 类型的循环左移，将 va 中的 8 个 int 类型的整型数据进行循环左移，移出的空位用移出位补充，四个数移位的位数相同，由 b 的低 5 位或 5 位立即数决定。

函数名

simd_sllow —— int256 的逻辑左移

对应指令

sllow

参数说明

[u]int256 va;

int b; （lit8 b）

返回值

[u]int256

功能说明

将Va中256位整体左移，移出的空位用“0”填充，移位位数由b低8位或8位立即数确定，结果写入Vc中。

函数名

simd_srlow —— int256 的逻辑右移

对应指令

srlow

参数说明

[u]int256 va;

int b; （lit8 b）

返回值

[u]int256

功能说明

将Va中256位整体右移，移出的空位用“0”填充，移位位数由b低8位或8位立即数确定，结果写入Vc中。

函数名

simd_vaddl —— 长字向量加法

对应指令

vaddl

参数说明

[u]int256 va;

[u]int256 vb(int8 b);

返回值

[u]int256

功能说明

将Va和Vb中4个64位长字（或8位立即数b）分别对应相加，结果保存到Vc中对应位置。

函数名

simd_vsubl —— 长字向量减法

对应指令

vsubl

参数说明

[u]int256 va;

[u]int256 vb(int8 b);

返回值

[u]int256

功能说明

将Va和Vb中4个64位长字（或8位立即数b）分别对应相减，结果保存到Vc中对应位置。

函数名

simd_vlog2x —— 两个 intv8 操作数的可重构逻辑运算

对应指令

vlog2x

参数说明

int n;

[u]intv8 va;

[u]intv8 vb;

返回值

[u]intv8 //intv8 类型的可重构逻辑运算

功能说明

两个 intv8 类型操作数的可重构逻辑运算，按照 n 的逻辑真值，将 va 中的 8 个 int 类型的整型数据与 vb 中的 8 个 int 类型的整型数据进行逻辑运算。

函数名

simd_vlog3x —— 三个 intv8 操作数的可重构逻辑运算

对应指令

vlog3r

参数说明

int n;

[u]intv8 va;

[u]intv8 vb;

[u]intv8 vc;

返回值

[u]intv8 //intv8 类型的可重构逻辑运算

功能说明

三个 intv8 类型操作数的可重构逻辑运算，按照 n 的逻辑真值，将 va、vb、vc 中的 8 个 int 类型的整型数据进行逻辑运算。

8.6.4 浮点向量运算函数接口

采用简单运算指令格式的浮点 SIMD 运算指令共 17 条，如下表所示：

序号	指 令		操 作 码	描 述
1	vadds	Va, Vb, Vc	0x1a.80	SP 浮点向量加
2	vaddd	Va, Vb, Vc	0x1a.81	DP 浮点向量加
3	vsubs	Va, Vb, Vc	0x1a.82	SP 浮点向量减
4	vsubd	Va, Vb, Vc	0x1a.83	DP 浮点向量减
5	vmuls	Va, Vb, Vc	0x1a.84	SP 浮点向量乘
6	vmuld	Va, Vb, Vc	0x1a.85	DP 浮点向量乘
7	vdivs	Va, Vb, Vc	0x1a.86	SP 浮点向量除

8	vdivd Va, Vb, Vc	0x1a.87	DP 浮点向量除
9	vsqrts Vb, Vc	0x1a.88	SP 浮点向量平方根
10	vsqrtd Vb, Vc	0x1a.89	DP 浮点向量平方根
11	vfcmpeq Va, Vb, Vc	0x1a.8c	浮点向量等于比较
12	vfcmple Va, Vb, Vc	0x1a.8d	浮点向量小于等于比较
13	vfcmplt Va, Vb, Vc	0x1a.8e	浮点向量小于比较
14	vfcmpun Va, Vb, Vc	0x1a.8f	浮点向量不可排序比较
15	vcpys Va, Vb, Vc	0x1a.90	浮点向量拷贝符号
16	vcpyse Va, Vb, Vc	0x1a.91	浮点向量拷贝符号和指数
17	vcpysn Va, Vb, Vc	0x1a.92	浮点向量拷贝符号的反码

采用浮点复合运算指令格式的浮点 SIMD 运算指令共 11 条，如下表所示：

序号	指 令	操 作 码	描 述
1	vmas Va, Vb, Vc, Vd	0x1b.00	SP 浮点向量乘加
2	vmad Va, Vb, Vc, Vd	0x1b.01	DP 浮点向量乘加
3	vmss Va, Vb, Vc, Vd	0x1b.02	SP 浮点向量乘减
4	vmsd Va, Vb, Vc, Vd	0x1b.03	DP 浮点向量乘减
5	vnmas Va, Vb, Vc, Vd	0x1b.04	SP 浮点向量负乘加
6	vnmad Va, Vb, Vc, Vd	0x1b.05	DP 浮点向量负乘加
7	vnmss Va, Vb, Vc, Vd	0x1b.06	SP 浮点向量负乘减
8	vnmsd Va, Vb, Vc, Vd	0x1b.07	DP 浮点向量负乘减
9	vseleq Va, Vb, Vc, Vd	0x1b.10	浮点向量等于“0”条件判断并选择
10	vsellt Va, Vb, Vc, Vd	0x1b.12	浮点向量小于“0”条件判断并选择
11	vselle Va, Vb, Vc, Vd	0x1b.13	浮点向量小于等于“0”条件判断并选择

表 5-3 浮点运算操作函数

内部函数	指令	操作	参数说明			
			VRc（返回值）	**VRav**	**VRbv**	**VRcv**
simd_vadds	vadds	+	floatv4	floatv4	floatv4	/
simd_vaddd	vaddd	+	doublev4	doublev4	doublev4	/
simd_vsubs	vsubs	-	floatv4	floatv4	floatv4	/
simd_vsubd	vsubd	-	doublev4	doublev4	doublev4	/

simd_vmuls	vmuls	*	floatv4	floatv4	floatv4	/
simd_vmuld	vmuld	*	doublev4	doublev4	doublev4	/
simd_vdivs	vdivs	除法	floatv4	floatv4	floatv4	
simd_vdivd	vdivd	除法	doublev4	doublev4	doublev4	
simd_vsqrts	vsqrts	平方根	floatv4	floatv4	floatv4	
simd_vsqrtd	vsqrtd	平方根	doublev4	doublev4	doublev4	
simd_vfcmpeq	vfcmpeq	等于比较	floatv4/	floatv4/	floatv4/	/
simd_vfcmple	vfcmple	小于等于比较	doublev4	doublev4	doublev4	/
simd_vfcmplt	vfcmplt	小于比较	floatv4/	floatv4/	floatv4/	/
simd_vfcmpun	vfcmpun	无序比较	doublev4	doublev4	doublev4	/
simd_vcpys	vcpys	符号拷贝	floatv4/	floatv4/	floatv4/	/
simd_vcpyse	vcpyse	浮点向量拷贝符号和指数	doublev4	doublev4	doublev4	/
simd_vcpysn	vcpysn	浮点向量拷贝符号的反码	floatv4/	floatv4/	floatv4/	/
simd_vmas	vmas	乘加	floatv4	floatv4	floatv4	floatv4
simd_vmad	vmad	乘加	doublev4	doublev4	doublev4	doublev4
simd_vmss	vmss	乘减	floatv4	floatv4	floatv4	floatv4
simd_vmsd	vmsd	乘减	doublev4	doublev4	doublev4	doublev4
simd_vnmas	vnmas	负乘加	floatv4	floatv4	floatv4	floatv4
simd_vnmad	vnmad	负乘加	doublev4	doublev4	doublev4	doublev4
simd_vnmss	vnmss	负乘减	floatv4	floatv4	floatv4	floatv4
simd_vnmsd	vnmsd	负乘减	doublev4	doublev4	doublev4	doublev4
simd_vseleq	vseleq	条件选择	floatv4/ doublev4	floatv4/ doublev4	floatv4/ doublev4	floatv4/ doublev4
simd_vsellt	vsellt		floatv4/ doublev4	floatv4/ doublev4	floatv4/ doublev4	floatv4/ doublev4
simd_vselle	vselle		floatv4/ doublev4	floatv4/ doublev4	floatv4/ doublev4	floatv4/ doublev4

说明：

d) 编程时，simd_vadds、simd_vsubs、simd_vmuls、simd_vaddd、simd_vsubd、simd_vmuld、simd_vdivs、simd_divd、simd_vmad、simd_vmas、simd_vmsd、simd_vmss、simd_vnmad、simd_vnmas、simd_vnmsd、simd_vnmss 可以用它们对应的运算符表示，不必调用内部函数。

e) 编程时向量化操作应尽量减少或避免使用条件语句。

f) 浮点条件选择指令的参数类型可以为 floatv4 或 doublev4，返回值与参数 VRbv 或 VRcv 的数据类型相同。

g) 浮点条件选择 simd_vseleq、simd_vselne、simd_vsellt、simd_vselle、simd_vselgt、 simd_vselge 和浮点符号拷贝 simd_vcpys、simd_vcpyse、simd_vcpysn 的参数可以是 floatv4 和 doublev4，返回值参数与 Rbv 相同。

函数名

simd_vadds —— floatv4 的加法运算

对应指令

vadds

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的加法运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点数进行加法运算。可以用“+”符号替换。

函数名

simd_vaddd —— doublev4 的加法运算

对应指令

vaddd

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的加法运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数进行加法运算。可以用“+”符号替换。

函数名

simd_vsubs —— floatv4 的减法运算

对应指令

vsubs

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的减法运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点数进行减法运算。可以用"-"符号替换。

函数名

simd_vsubd —— doublev4 的减法运算

对应指令

vsubd

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的减法运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数进行减法运算。可以用"-"符号替换。

函数名

simd_vmuls —— floatv4 的乘法运算

对应指令

vmuls

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的乘法运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点数进行乘法运算。可以用"*"符号替换。

函数名

simd_vmuld —— doublev4 的乘法运算

对应指令

vmuld

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的乘法运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数进行乘法运算。可以用“*”符号替换。

函数名

simd_vdivs —— floatv4 的除法运算

对应指令

vdivs

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的除法运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点数进行除法运算。可以用“÷”符号替换。

函数名

simd_vdivd —— doublev4 的除法运算

对应指令

vdivd

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的除法运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数进行除法运算。可以用“÷”符号替换。

函数名

simd_vsqrts —— floatv4 的求平方根运算

对应指令

vsqrts

参数说明

floatv4 va;

floatv4 vb;

返回值

floatv4

功能说明

floatv4 类型的求平方根运算，将 va 中的 4 个 float 浮点数分别与 vb 中的 4 个 float 浮点数分别求平方根。

函数名

simd_vsqrtd —— doublev4 的求平方根运算

对应指令

vsqrtd

参数说明

doublev4 va;

doublev4 vb;

返回值

doublev4

功能说明

doublev4 类型的求平方根运算，将 va 中的 4 个 double 浮点数分别与 vb 中的 4 个 double 浮点数分别求平方根。

函数名

simd_vfcmpeq —— doublev4/floatv4 的等于比较运算

对应指令

vfcmpeq

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4 vc

功能说明

将va和vb中的对应浮点向量元素进行等于比较，如果条件成立，则将非“0”浮点值（2.0）写入vc的对应位置，否则将真“0”写入vc的对应位置。

函数名

simd_vfcmple —— doublev4/floatv4 的小于等于比较运算

对应指令

vfcmple

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4 vc

功能说明

将va和vb中的对应浮点向量元素进行小于等于比较，如果条件成立，则将非“0”浮点值（2.0）写入vc的对应位置，否则将真“0”写入vc的对应位置。

函数名

simd_vfcmplt —— doublev4/floatv4 的小于比较运算

对应指令

vfcmplt

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4 vc

功能说明

将va和vb中的对应浮点向量元素进行小于比较，如果条件成立，则将非“0”浮点值（2.0）写入vc的对应位置，否则将真“0”写入vc的对应位置。

函数名

simd_vfcmpun —— doublev4/floatv4 的无序比较运算

对应指令

vfcmpun

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4 vc

功能说明

将va和vb中的对应浮点向量元素进行无序比较，如果条件成立，则将非“0”浮点值（2.0）写入vc的对应位置，否则将真“0”写入vc的对应位置。

函数名

simd_vcpys —— floatv4/doublev4 的拷贝符号

对应指令

vcpys

参数说明

floatv4 va;[doublev4 va;]

floatv4 vb;[doublev4 vb;]

返回值

floatv4 [doublev4]

功能说明

浮点拷贝符号，返回值中的符号位是 va 的符号位，其余部分为 vb 中的数据。

函数名

simd_vcpyse —— floatv4/doublev4 的拷贝符号和指数

对应指令

vcpyse

参数说明

floatv4 va;[doublev4 va;]

floatv4 vb;[doublev4 vb;]

返回值

floatv4 [doublev4]

功能说明

浮点拷贝符号和指数，返回值中的符号位和指数位是 va 的数据，其余部分为 vb 中的数据。

函数名

simd_vcpysn —— floatv4/doublev4 的拷贝符号反码

对应指令

vcpysn

参数说明

floatv4 va;[doublev4 va;]

floatv4 vb;[doublev4 vb;]

返回值

floatv4 [doublev4]

功能说明

浮点拷贝符号反码，返回值中的符号位是 va 的符号位的反码，其余部分为 vb 中的数据。

函数名

simd_vmas —— floatv4 的乘加运算

对应指令

vmas

参数说明

floatv4 va;

floatv4 vb;

floatv4 vc;

返回值

floatv4

功能说明

floatv4 类型的乘加运算，将 va 中的 4 个 float 浮点数和 vb 中的 4 个 float 浮点数以及 vc 中的 4 个浮点数分别进行乘加运算。可以用“*”和“+”符号替换。

ret = va*vb + vc;

函数名

simd_vmad —— doublev4 的乘加运算

对应指令

vmad

参数说明

doublev4 va;

doublev4 vb;

doublev4 vc;

返回值

doublev4

功能说明

doublev4 类型的乘加运算，将 va 中的 4 个 double 浮点数和 vb 中的 4 个 double 浮点数以及 vc 中的 4 个浮点数分别进行乘加运算。可以用“*”和“+”符号替换。

ret = va*vb + vc;

函数名

simd_vmss —— floatv4 的乘减运算

对应指令

vmss

参数说明

floatv4 va;

floatv4 vb;

floatv4 vc;

返回值

floatv4

功能说明

floatv4 类型的乘减运算，将 va 中的 4 个浮点数和 vb 中的 4 个浮点数以及 vc 中的 4 个浮点数分别进行乘减运算。可以用“*”和“-”符号替换。

ret=va*vb-vc;

函数名

simd_vmsd —— doublev4 的乘减运算

对应指令

vfmsd

参数说明

doublev4 va;

doublev4 vb;

doublev4 vc;

返回值

doublev4

功能说明

doublev4 类型的乘减运算，将 va 中的 4 个 double 浮点数和 vb 中的 4 个 double 浮点数以及 vc 中的 4 个浮点数分别进行乘减运算。可以用“*”和“-”符号替换。

ret=va*vb-vc；

函数名

simd_vnmas —— floatv4 的负乘加运算

对应指令

vnmas

参数说明

floatv4 va;

floatv4 vb;

floatv4 vc;

返回值

floatv4

功能说明

floatv4 类型的负乘加运算，将 va 中的 4 个 float 浮点数和 vb 中的 4 个 float 浮点数以及 vc 中的 4 个浮点数分别进行负乘加运算。可以用“*”和“+”和“-”符号替换。

ret = -va*vb + vc；

函数名

simd_vnmad —— doublev4 的负乘加运算

对应指令

vnmad

参数说明

doublev4 va;

doublev4 vb;

doublev4 vc;

返回值

doublev4

功能说明

doublev4 类型的负乘加运算，将 va 中的 4 个 double 浮点数和 vb 中的 4 个 double 浮点数以及 vc 中的 4 个浮点数分别进行负乘加运算。可以用“*”和“+”和“-”符号替换。

ret = -va*vb + vc;

函数名

simd_vnmss —— floatv4 的负乘减运算

对应指令

vnmss

参数说明

floatv4 va;

floatv4 vb;

floatv4 vc;

返回值

floatv4

功能说明

floatv4 类型的负乘减运算，将 va 中的 4 个 float 浮点数和 vb 中的 4 个 float 浮点数以及 vc 中的 4 个浮点数分别进行负乘减运算。可以用“*”和“+”和“-”符号替换。

ret = -va*vb - vc;

函数名

simd_vnmsd —— doublev4 的负乘减运算

对应指令

vnmsd

参数说明

doublev4 va;

doublev4 vb;

doublev4 vc;

返回值

doublev4

功能说明

doublev4 类型的负乘减运算，将 va 中的 4 个 double 浮点数和 vb 中的 4 个 double 浮点数以及 vc 中的 4 个浮点数分别进行负乘减运算。可以用“*”和“+”和“-”符号替换。

ret = -va*vb - vc;

函数名

simd_vseleq —— doublev4/floatv4 的等于选择运算

对应指令

vseleq

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

doublev4/floatv4 vc;

返回值

doublev4/floatv4

功能说明

判断 doublev4/floatv4 类型的参数 va，如果等于 0，则返回 vb 的值；否则返回 vc 的值。

函数名

simd_vsellt —— doublev4/floatv4 的小于选择运算

对应指令

vsellt

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

doublev4/floatv4 vc;

返回值

doublev4/floatv4

功能说明

判断 doublev4/floatv4 类型的参数 va，如果小于 0，则返回 vb 的值；否则返回 vc 的值。

函数名

simd_vselle —— doublev4/floatv4 的小于等于选择运算

对应指令

vselle

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

doublev4/floatv4 vc;

返回值

doublev4/floatv4

功能说明

判断 doublev4/floatv4 类型的参数 va，如果小于等于 0，则返回 vb 的值；否则返回 vc 的值。

8.6.5 数据整理函数接口

序号	指 令		寄存器格式操作码	立即数格式操作码	描 述
1	VINSW	Ra, Vb, #c, Vd	—	0x1b.20	字向量元素插入
2	VINSF	Ra, Vb, #c, Vd	—	0x1b.21	浮点向量元素插入
3	VEXTW	Va, #c, Rd	—	0x1b.22	字向量元素提取
4	VEXTF	Va, #c, Rd	—	0x1b.23	浮点向量元素提取
5	VSHFW	Va, Vb, Rc, Vd	0x1b.26	—	字向量元素混洗
6	VSHFF	Va, Vb, Rc/#c, Vd	0x1b.27	0x1b.28~2f	浮点向量元素混洗

表 7-4 浮点运算操作函数

内部函数	指令	操作	参数说明			
			VRc（返回值）	**VRav**	**VRbv**	**VRcv**
simd_vinswn n=(0,1,…,7)	vinsw	字向量插入	intv8	intv8	intv8	/
simd_vinsfn n=(0,1,2,3)	vinsf	浮点向量插入	floatv4/doublev4	floatv4/ doublev4	floatv4/ doublev4	/
simd_vextwn n=(0,1,…,7)	vextw	字向量提取	intv8	intv8	/	/
simd_vextfn n=(0,1,2,3)	vextf	浮点向量提取	floatv4/doublev4	floatv4/ doublev4	/	/
simd_vshuffle	vshfw	字向量混洗	intv8	intv8	intv8	int8
simd_vshuffle	vshff	浮点向量混洗	floatv4/doublev4	floatv4/ doublev4	floatv4/ doublev4	int8

说明：

1）simd_vinswn 和 simd_vextwn 中 n 的取值只能是 0、1、2、…、7，simd_vinsfn 和 simd_vextfn 中 n 的取值只能是 0、1、2、3；

函数名

simd_vinswn—— 字向量元素插入运算

对应指令

vinsw

参数说明

intv8 va;

intv8 vb;

返回值

intv8

功能说明

将浮点寄存器va低32位字整数，替代vb中由n（有效值为0～7）指定32位字元素，组成并返回新的字整数向量。

该接口实际上对应8个接口，分别是simd_vinsw0、simd_vinsw1、simd_vinsw2、simd_vinsw3，simd_vinsw4，simd_vinsw5，simd_vinsw6，simd_vinsw7，比如：

vc=simd_vinsw2（va，vb）

表示va低64位的寄存器格式字整数先转换为32位字整数，然后替代vb的64：95这32位的字元素，组成新的vb返回给vc。

函数名

simd_vinsfn—— 浮点向量元素插入运算

对应指令

vinsf

参数说明

doublev4/floatv4 va;

doublev4/floatv4 vb;

返回值

doublev4/floatv4

功能说明

将浮点寄存器 va 低 64 位浮点数据替代 vb 中由 n（有效值为 0～3）指定的 64 位浮点元素，组成并返回新的浮点向量。

该接口实际上对应 4 个接口，分别是 simd_vinsf0、simd_vinsf1、simd_vinsf2、simd_vinsf3，比如：

vc=simd_vinsf1(va,vb)

表示 va 低 64 位浮点数据替代 vb 的 64：127 这 64 位的浮点元素，组成新的 vb 返回给 vc。

函数名

simd_vextwn—— 字向量元素提取运算

对应指令

vextw

参数说明

intv8 va;

返回值

intv8

功能说明

将 va 中由 n（有效值为 0～7）指定的 32 位字元素，变为 64 位寄存器格式的字整数写入目的浮点寄存器的低 64 位，目的寄存器的高 192 位内容不确定。

该接口实际上对应 8 个接口，分别是 simd_vextw0、simd_vextw1、simd_vextw2、simd_vextw3、simd_vextw4、simd_vextw5、simd_vextw6、simd_vextw7，比如：

vc=simd_vextw2(va)

表示 va 的 64：95 这 32 位的字元素，变为 64 位寄存器格式的字整数写入 vc 低 64 位，其中 vc 高 192 位内容不确定。

函数名

simd_vextfn—— 浮点向量元素提取运算

对应指令

vextf

参数说明

doublev4/floatv4 va;

返回值

doublev4/floatv4

功能说明

将 va 中由 n（有效值为 0～3）指定的 S 浮点或 D 浮点元素存入目的寄存器的<63:0>中。此指令可实现 S 浮点和 D 浮点的提取。

该接口实际上对应 4 个接口，分别是 simd_vextf0、simd_vextf1、simd_vextf2、simd_vextf3，比如：

vc=simd_vextf1(va)

表示 va 的 64：127 这 64 位的浮点元素存入 vc 的<63:0>中。

函数名

simd_vshuffle—— 字/浮点向量元素混洗

对应指令

vshfw

参数说明

a.字向量元素混洗

intv8 va;

intv8 vb

int8 Rc

b.浮点向量元素混洗

doublev4/floatv4 va;

doublev4/floatv4 vb

int8 Rc/#c

返回值

a.字向量元素混洗

intv8

b.浮点向量元素混洗

doublev4/floatv48

功能说明

a．字向量元素混洗

返回值的八个 32bit 的数由 Va 和 Vb 装填，Va 负责装填 4 个高 32bit，Vb 负责装填 4 个低 32bit，具体如何装填由 Rc 说明：

Rc 的<30:28>指出 vd 的<255:224>来自 Va 的位置

Rc 的<26:24>指出 vd 的<223:192>来自 Va 的位置

Rc 的<22:20>指出 vd 的<191:160>来自 Va 的位置

Rc 的<18:16>指出 vd 的<159:128>来自 Va 的位置

Rc 的<14:12>指出 vd 的<127:96>来自 Vb 的位置

Rc 的<10:8>指出 vd 的<95:64>来自 Vb 的位置

Rc 的<6:4>指出 vd 的<63:32>来自 Vb 的位置

Rc 的<2:0>指出 vd 的<31:0>来自 Vb 的位置

Va/Vb 的位置（0~7）含义为，Va/Vb 分成自然对界的 8 个 32bit，位置为 0 表示第 0 个 32bit（[31:0]），依次类推。

示例：

intv8 va,vb,vd;

int rc = 0x12345670;

vd = vshuffle（va, vb, rc）。

含义是从高到低，vd 的 8 个 32bit 分别来源于：

0x1：va 的第 1 个 32bit；

0x2：va 的第 2 个 32bit；

0x3：va 的第 3 个 32bit；

0x4：va 的第 4 个 32bit；

0x5：vb 的第 5 个 32bit；

0x6：vb 的第 6 个 32bit；

0x7：vb 的第 7 个 32bit；

0x0：vb 的第 0 个 32bit。

b．浮点向量元素混洗

返回值的四个 64bit 的数由 Va 和 Vb 装填，Va 负责装填 Vd 的 2 个高 64bit，Vb 负责装填 Vd 的 2 个低 64bit，具体如何装填由 Rc/#c 说明：

Rc/#c 的<7:6>指出 vd 的<255:192>来自 Va 的位置

Rc/#c 的<5:4>指出 vd 的<191:128>来自 Va 的位置

Rc/#c 的<3:2>指出 vd 的<127:64>来自 Vb 的位置

Rc/#c 的<1:0>指出 vd 的<63:0>来自 Vb 的位置

Va/Vb 的位置（0~3）含义为，Va/Vb 分成自然对界的 4 个 64bit，位置为 0 表示第 0 个 64bit（[63:0]），依次类推。

示例：以立即数操作数为例说明

```
doublev4 va,vb,vd;
vd = simd_shuffle(va, vb, 8’b01111000)
```

含义是从高到低，vd 的 4 个 64bit 分别来源于 va 的第 1 个 64 比特，va 的第 3 个 64bit，vb 的第 2 个 64bit，vb 的第 0 个 64bit。

8.6.6 赋值函数接口

同运算控制核心的运算操作宏定义，见 9.5.6.

8.6.7 访问存储器并广播函数接口

实现对存储器的访问，并将数据广播到行总线或列总线，共4条。

序号	指 令	操 作 码	描 述
1	vldr Va, disp(Rb)	0x24	从局部存储器 LD 一个向量（256 位）并行广播
2	vldc Va, disp(Rb)	0x2c	从局部存储器 LD 一个向量（256 位）并列广播
3	ldder Va, disp(Rb)	0x27	从局部存储器 LD 一个双精度浮点或长字（64 位），进行向量扩展，然后行广播
4	lddec Va, disp(Rb)	0x2f	从局部存储器 LD 一个双精度浮点或长字（64 位），进行向量扩展，然后列广播

宏定义	指令	操作	参数说明	
			VRav	Rbv
simd_loadr	vldr	装入并行广播	扩展类型	int*,long*,double*
simd_loadc	vldc	装入并列广播	扩展类型	int*,long*,double*
simd_loader	ldder	装入、扩展并行广播	扩展类型	int*,long*, double*

simd_loadec	lddec	装入、扩展并列广播	扩展类型	int*,long*, double*

说明：当 Rbv 为 int * 或 long *类型时，VRav 可以是 intv8 或 int256， Rab 为 double *类型时，VRav 只能是 doublev4 类型。如果上述类型不匹配，编译器将报错。

函数名

simd_loadr —— 扩展类型装入并广播

对应指令

vldr

参数说明

a) intv8 类型的装入

[u]intv8 va;

[unsigned] int *addr;[[unsigned] long *addr;] //intv8 类型的装入

b) int256 类型的装入

[u]int256va;

[unsigned] int *addr;[[unsigned] long *addr;] //int256 类型的装入

c) doublev4 类型的装入

doublev4 va;

double *addr; //doublev4 类型的装入

返回值

a) intv8 类型的装入

[u]intv8 //intv8 类型的装入

b) int256 类型的装入

[u]int256va; //int256 类型的装入

c) doublev4 类型的装入

doublev4; //doublev4 类型的装入

功能说明

扩展类型的装入，将 32 字节长度的数据从连续内存区域中装入到一个向量变量中，并进行行广播。

函数名

simd_loadc —— 扩展类型装入并广播

对应指令

vldc

参数说明

a) intv8 类型的装入

[u]intv8 va;

[unsigned] int *addr;[[unsigned] long *addr;] //intv8 类型的装入

b) int256 类型的装入

[u]int256 va;

[unsigned] int *addr;[[unsigned] long *addr;] //int256 类型的装入

c) doublev4 类型的装入

doublev4 va;

double *addr; //doublev4 类型的装入

返回值

a) intv8 类型的装入

[u]intv8//intv8 类型的装入

b) int256 类型的装入

[u]int256va; //int256 类型的装入

c) doublev4 类型的装入

doublev4; //doublev4 类型的装入

功能说明

扩展类型的装入，将 32 字节长度的数据从连续内存区域中装入到一个向量变量中，并进行列广播。

函数名

simd_loader —— 扩展类型装入、向量扩展并行广播

对应指令

ldder

参数说明

a) intv8 类型的装入

[u]intv8va;

[unsigned] int *addr;[[unsigned] long *addr;] //intv8 类型的装入

b) int256 类型的装入

[u]int256 va;

[unsigned] int *addr;[[unsigned] long *addr;] //int256 类型的装入

c) doublev4 类型的装入

doublev4 va;

double *addr; //doublev4 类型的装入

返回值

a) intv8 类型的装入

[u]intv8 //intv8 类型的装入

b) int256 类型的装入

[u]int256 va; //int256 类型的装入

c) doublev4 类型的装入

doublev4; //doublev4 类型的装入

功能说明

扩展类型的装入并扩展。从存储器装载一个双精度浮点或长字（64 位），同时写到向量寄存器 Va 的四个双精度浮点向量元素的位置，并进行行向广播。

函数名

simd_loadec —— 扩展类型装入、向量扩展并列广播

对应指令

lddec

参数说明

a) intv8 类型的装入

[u]intv8 va;

[unsigned] int *addr;[[unsigned] long *addr;] //intv8 类型的装入

b) int256 类型的装入

[u]int256va;

[unsigned] int *addr;[[unsigned] long *addr;] //intv8 类型的装入

c) doublev4 类型的装入

doublev4 va;

double *addr; //doublev4 类型的装入

返回值

a) intv8 类型的装入

[u]intv8 //intv8 类型的装入

b) int256 类型的装入

[u]int256 va; //int256 类型的装入

c) doublev4 类型的装入

doublev4; //doublev4 类型的装入

功能说明

扩展类型的装入并扩展。从存储器装载一个双精度浮点或长字（64 位），同时写到向量寄存器 Va 的四个双精度浮点向量元素的位置，并进行列向广播。

8.6.8 寄存器通信函数接口

实现核组内同行、同列运算核心间细粒度通信。

序号	指 令	寄存器格式操作码	立即数格式操作码	描 述
1	getr Va	0x2d.00	—	读行通信缓冲
2	getc Va	0x2d.01	—	读列通信缓冲
3	putr Va, Rb/#dest	0x2d.02	0x2d.12	向同行目标运算核心送数
4	putc Va, Rb/#dest	0x2d.03	0x2d.13	向同列目标运算核心送数

宏定义	指令	操作	参数说明	
			VRav	**Rbv**
simd_getr	getr	读行通信缓冲	扩展类型	
simd_getc	getc	读列通信缓冲	扩展类型	
simd_putr	putr	向同行目标运算核心送数	扩展类型	int,unsigned int, long,unsigned long
simd_putc	putc	向同列目标运算核心送数	扩展类型	int,unsigned int, long,unsigned long

说明：1）读行（或列）通信缓冲，缓冲宽度为向量寄存器宽度。通信缓冲是一个 FIFO 的缓冲，并且具有读后清的属性。如果通信缓冲为空则停顿。2）向同行（或同列）上目标运算核心发送数据，数据宽度为向量寄存器宽度。PUT 可以点对点操作或广播操作，当广播时目标方至少一个满时停顿。Rb 低 4 位有效，其中 Rb[3]（或#dest[3]）为通信类型位，为 0 时表示点对点操作，则 Rb[2:0]（或#dest[2:0]）是一个 3 位目标位向量，指示目标运算核心号；当 Rb[3]（或#dest[3]）为 1 时表示广播操作，忽略 Rb[2:0]位。

函数名

simd_getr ——读行通信缓冲

对应指令

getr

参数说明

a) intv8 类型的装入

[u]intv8va;

b) int256 类型的装入

[u]int256 va;

c) doublev4 类型的装入

doublev4 va;

d) floatv4 类型的装入

floatv4 va;

返回值

a) intv8 类型的装入

[u]intv8va;

b) int256 类型的装入

[u]int256 va;

c) doublev4 类型的装入

doublev4 va;

d) floatv4 类型的装入

floatv4 va;

功能说明

读行通信缓冲，将缓冲中的向量读入向量寄存器中，缓冲执行读后清。

函数名

simd_getc ——读行通信缓冲

对应指令

getc

参数说明

a) intv8 类型的装入

[u]intv8va;

b) int256 类型的装入

[u]int256 va;

c) doublev4 类型的装入

doublev4 va;

d) floatv4 类型的装入

floatv4 va;

返回值

a) intv8 类型的装入

[u]intv8va;

b) int256 类型的装入

[u]int256 va;

c) doublev4 类型的装入

doublev4 va;

d) floatv4 类型的装入

floatv4 va;

功能说明

读列通信缓冲，将缓冲中的向量读入向量寄存器中，缓冲执行读后清。

函数名

simd_putr —— 向同行目标运算核心送数

对应指令

putr

参数说明

a) intv8 类型的装入

[u]intv8va;

b) int256 类型的装入

[u]int256 va;

c) doublev4 类型的装入

doublev4 va;

d) floatv4 类型的装入

floatv4 va;

返回值

a) intv8 类型的装入

[u]intv8va;

b) int256 类型的装入

[u]int256 va;

c) doublev4 类型的装入

doublev4 va;

d) floatv4 类型的装入

floatv4 va;

功能说明

向同行上目标运算核心发送数据，数据宽度为向量寄存器宽度。PUT 可以点对点操作或广播操作，当广播时目标方至少一个满时停顿。Rb 低 4 位有效，其中 Rb[3]（或#dest[3]）为通信类型位，为 0 时表示点对点操作，则 Rb[2:0]（或#dest[2:0]）是一个 3 位目标位向量，指示目标运算核心号；当 Rb[3]（或#dest[3]）为 1 时表示广播操作，忽略 Rb[2:0]位。

函数名

simd_putc —— 向同行目标运算核心送数

对应指令

putc

参数说明

a) intv8 类型的装入

[u]intv8va;

b) int256 类型的装入

[u]int256 va;

c) doublev4 类型的装入

doublev4 va;

d) floatv4 类型的装入

floatv4 va;

返回值

a) intv8 类型的装入

[u]intv8va;

b) int256 类型的装入

[u]int256 va;

c) doublev4 类型的装入

doublev4 va;

d) floatv4 类型的装入

floatv4 va;

功能说明

向同列上目标运算核心发送数据，数据宽度为向量寄存器宽度。PUT 可以点对点操作或广播操作，当广播时目标方至少一个满时停顿。Rb 低 4 位有效，其中 Rb[3]（或#dest[3]）为通信类型位，为 0 时表示点对点操作，则 Rb[2:0]（或#dest[2:0]）是一个 3 位目标位向量，指示目标运算核心号；当 Rb[3]（或#dest[3]）为 1 时表示广播操作，忽略 Rb[2:0]位。

8.6.9 向量查表函数接口

序号	指 令	寄存器格式操 作 码	立即数格式操作码	描 述
1	selldw Va, Rb, #c, Vd	—	0x00.10	字向量选择装入

内部函数	指令	操作	参数说明			
			VRc（返回值）	VRav	VRbv	VRcv
simd_selldw	selldw	字向量元素插入	intv8（VRcv=0 时有效）	intv8	Int *	int
simd_lookup	selldw	字向量元素提取	/	intv8	Int *	intv8

说明：

1）simd_selldw 中 VRcv 的值只取低 3 位有效，当 VRcv 不为 0 时，VRd 无返回值；

2）simd_lookup 的返回值直接写入 Vrcv 的地址。

函数名

simd_selldw—— 字向量选择装入运算

对应指令

selldw

参数说明

intv8 va;

intv8 Rb;

Int3 #c;

返回值

intv8（c=0 时有效）

功能说明

根据#c，从 Va 指出的 8 个表内偏移中选择 1 个，与 Rb 中的首地址相加，得到访问 LDM 的地址；然后访问 LDM，将查到的 32bit 数据写入硬件 256b 缓冲相应位置。如果#c 为 3'b0，访问 LDM 结束后会把硬件 256b 缓冲的结果写回 Vd。

函数名

simd_lookup——查表，完成四次字向量选择装入运算

对应指令

selldw

参数说明

intv8 va;

intv8 Rb;

intv8 vc;

返回值

功能说明

生成 8 条查表指令：

simd_selldw (va,b,7);

simd_selldw (va,b,6);

simd_selldw (va,b,5);

simd_selldw (va,b,4);

simd_selldw (va,b,3);

simd_selldw (va,b,2);

simd_selldw (va,b,1);

vc=simd_selldw (va,b,0);

8.6.10 扩展类型的打印函数

同运算控制核心的扩展类型的打印函数，见 8.5.7.

第九章 调试与典型问题的解决办法

9.1 通过 gprof 工具调试性能

本节使用 CPU2000 标准程序测试包中的浮点课题 168.wupwise 作为例子。168 实现的是量子论色动力学，源程序用 Fortran77 编写而成，共有 2100 行代码和 23 个源文件。实验全部采用 reference 规模（CPU2000 的最大输入数据规模）执行，测试平台是主频为 1GHz 的 SW 26010 运算控制核心芯片。

不同的优化选项对程序的性能有着不同的影响，我们首先采用-O2 和-O3 选项分别编译，得到执行时间如下：

-O2　　568.91s

-O3　　480.02s

很明显，经-O3 编译的程序性能要高于-O2，这是因为相比于-O2，-O3 使得编译器在编译时做了更多的优化。然而，优化越多并不意味着性能一定会越高，不同的优化之间也可能由于相互冲突而导致总体性能的下降。因此，要想确定究竟哪种优化才能进一步提升 168.wupwise 的性能，我们必须要获取更多程序执行方面的信息。

gprof 工具能够很好的满足我们的需求。我们在编译时加入-pg 选项，程序执行结束后就会生成一个包含有过程间调用信息的“gmon.out”文件，然后 gprof 就能够通过目标文件和 gmon.out 文件获取过程执行时间，过程调用次数等一系列程序执行方面的信息。

下面我们采用-O3 –pg 选项重新编译 168.wupwise 课题，生成的目标文件为 wupwise。

并采用如下的方式执行：

```
time -p ./wupwise > wupwise.out
```

执行时间为 1249.59s，最后生成 gmon.out 文件，接着再用如下命令，

```
gprof wupwise gmon.out
```

此时屏幕会打印出下面的信息：

==

Flat profile:

Each sample counts as 0.000976562 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
51.41	267.10	267.10	155648000	0.00	0.00	zgemm_
21.99	381.36	114.26	603648604	0.00	0.00	zaxpy_
7.40	419.79	38.43	933888000	0.00	0.00	lsame_
6.02	451.08	31.29	214528306	0.00	0.00	zcopy_
3.06	466.99	15.91	155648000	0.00	0.00	gammul_
2.96	482.38	15.38	603648604	0.00	0.00	dcabs1_
1.38	489.57	7.19	512301	0.00	0.00	zdotc_

```
 1.28    496.23     6.66   155648000   0.00    0.00   su3mul_
 1.19    502.39     6.17    39936075   0.00    0.00   zscal_
 0.84    506.74     4.35     1024077   0.00    0.00   dznrm2_
 0.72    510.47     3.73         152   0.02    1.65   muldeo_
 0.71    514.18     3.70         152   0.02    1.65   muldoe_
……
 0.00    519.55     0.00           1   0.00  516.49   MAIN__
 0.00    519.55     0.00           1   0.00    0.00   init_
 0.00    519.55     0.00           1   0.00    0.22   phinit_
================================================================
```

加入-pg 选项后执行时间慢了很多，这是因为程序在执行过程中要不断收集过程执行信息，消耗了一部分时间，而经 gprof 统计出的程序执行时间 519.55s 也比原先的 480.02s 要慢一些。

gprof 打印出的第二部分输出包含有函数间的调用关系，其中列出了函数调用图中每个分支花费时间所占的百分比，比如 muldoe_调用 su3mul_，su3mul_调用 zgemm_这条分支占去总时间的 60%。

```
================================================================
Call graph (explanation follows)
granularity: each sample hit covers 4 byte(s) for 0.00% of 519.55 seconds
index % time    self  children    called          name
                0.00  516.49       1/1                main [2]
[1]     99.4    0.00  516.49       1              MAIN__ [1]
                0.00  502.29     152/152              matmul_ [3]
                0.17   13.59       1/1                uinith_ [13]
                0.00    0.22       1/1                phinit_ [23]
                0.02    0.20       1/2                rndphi_ [22]
                0.00    0.00     301/512301           zdotc_ [14]
                0.00    0.00      77/1024077          dznrm2_ [16]
                0.00    0.00     452/603648604          zaxpy_ [9]
                0.00    0.00     154/214528306          zcopy_ [11]
                0.00    0.00      75/39936075         zscal_ [15]
                0.00    0.00       1/1                init_ [28]
-----------------------------------------------------------------------------------------
                                                       <spontaneous>
[2]     99.4    0.00  516.49                      main [2]
                0.00  516.49       1/1                MAIN__ [1]
-----------------------------------------------------------------------------------------
                0.00  502.29     152/152              MAIN__ [1]
```

```
[3]      96.7      0.00   502.29      152              matmul_ [3]
                   3.73   247.43      152/152              muldeo_ [6]
                   3.70   247.43      152/152              muldoe_ [7]
                   0.00     0.00      152/603648604          zaxpy_ [9]
                   0.00     0.00      152/214528306          zcopy_ [11]
-----------------------------------------------------------------------------------------------
                   3.33   152.77 77824000/155648000         muldeo_ [6]
                   3.33   152.77 77824000/155648000         muldoe_ [7]
[4]      60.1      6.66   305.53 155648000               su3mul_ [4]
                   267.10    38.43 155648000/155648000         zgemm_ [5]
-----------------------------------------------------------------------------------------------
                   267.10    38.43 155648000/155648000         su3mul_ [4]
[5]      58.8      267.10    38.43 155648000               zgemm_ [5]
                   38.43      0.00 933888000/933888000        lsame_ [10]
=======================================================================
```

下面我们做一些分析。函数 zgemm 占去了执行总时间的 51%，是一个明显的热点，还有一些小函数，比如 zaxpy，zcopy，lsame, dcabs1，分别被调用了很多次，可以考虑把它们作为内联的候选，以减少函数调用的开销。因此，我们需要让编译器做一些过程间的优化，在编译时加入-ipa 选项，编译器就能够分析过程间的各种信息，并根据这些信息做一些函数内联，常数传播之类的优化。

接下来我们采用-O3 -ipa 选项重新编译执行，时间为 295.52s，性能提升了 38.4%。当然，还可以根据具体情况尝试其它的优化选项，比如在-O3 -ipa 选项的基础上，我们可以加入选项-LNO:pf_ahead=3，该选项用于调整预取的距离，这样执行时间又缩短到 279.83s。需要注意的是，并不是所有的优化选项组合起来就一定能提升性能，它们之间也有可能互相冲突，从而影响性能，这就需要具体情况具体分析。

9.2 debug 调试

为了方便调试，SWCC 编译器提供了对 debug 调试接口信息的支持。用户可以使用“-g”选项编译一个.c 或.f 源文件，这样能够在目标码中产生附加的调试信息，这些调试信息描述了源文件中的一些特征，比如行号信息、变量的类型和范围信息、函数的名字及参数和范围信息等。这些符号表信息按照 dwarf 格式生成，用以支持 gdb 的调试。

如果不想采用过多的优化调试，则可以采用-O0 和-g 选项编译。当使用-O0 选项时，-g 选项是默认打开的。

9.3 对未初始化变量的处理

如果程序中存在没有被初始化的变量，那么执行后很可能因此而得到无法预期的结果。为此，SWCC 编译器中加入了可以识别处理未初始化变量的选项，分别是

-trapuv | -Wuninitialized | –zerouv

选项-trapuv 能够把局部变量初始化为浮点非数（NaN），同时能够让 CPU 识别出包含浮点非数在内的浮点运算，比如说加，减，乘，除，求正弦，求平方根以及比较运算等等。当存在非数时，运算即被中止，并报浮点异常，这样用户可以尽快发现问题。-trapuv 选项对局部标量变量，数组以及经函数 alloca()分配的内存地址起作用，而对全局变量，Fortran 中的公共块以及经函数 malloc()分配的内存地址无效。

选项-Wuninitialized 能够发出变量没有被初始化的报警信息，如果使用-Wno-uninitialized 选项，则编译器不会发出警告信息。

选项-zerouv 用于把用户程序中未初始化的变量置 0。使用该选项对程序性能会有一点点影响。该选项支持局部标量变量，数组以及经函数 alloca()分配的内存地址；对全局变量，Fortran 中的公共块以及经函数 malloc()分配的内存地址无效。

9.4 对大的静态数据的处理

静态分配的数据大小不能超过 2GB，比如 Fortran 公共块和某些 C 变量等数据结构，一旦超出了这个范围（不加-mcmodel=medium 选项时），编译时则会报出下面的信息：

relocation truncated to fit: R_Alpha_PC32

对 Fortran 而言，如果公共块的大小没有超过 2GB，则程序可以被正确编译执行。否则，可以使用-OPT:reorg_common 选项，该选项能够把公共块超过 2GB 范围的部分分裂出来成为一个新的块，从而把它限制在 2GB 的范围内。

当使用-O3 或-Ofast 选项时，该选项是默认打开的。如果-O2 或者更低级的选项能够编译正确，而-O3 或-Ofast 编译错误，则可以尝试把-OPT:reorg_common 选项关掉。同样，使用-mcmodel=medium 选项也可以达到相同的优化效果。

9.5 嵌汇编

用户在使用嵌汇编时，所使用的寄存器总数应当不超过 SW 26010 运算控制核心的寄存器个数。SW 26010 运算控制核心包含 32 个定点寄存器和 32 个浮点寄存器，除去特殊 GP，SP，ZERO 寄存器，用户嵌汇编能够使用的定点或浮点寄存器总数应当小于 29 个。

9.6 使用-ipa 和-Ofast 选项

如果使用-ipa 选项，则刚开始生成的.o 文件并不是真正的规则的.o 文件，它们仅仅被 IPA 用于收集用户程序中函数间的一些信息，这些信息将被用于 IPA 第二阶段的优化。

因此，当使用-ipa 选项时，生成的假.o 文件必须加上-ipa 选项一起被第二遍编译，从而生成最终的目标码。如果需要链接.a 形式的库文件（比如说 libfoo.a），则必须保证.a 文件中要么全都是假的.o 文件，要么全都是真的.o 文件，不能出现两种.o 文件混合的情况。

当需要使用-ipa 选项和库文件链接时，则要先把.a 形式的库文件分离成.o 形式，然后再加-ipa 选项编译。

当使用-Ofast 选项时，-ipa 选项默认是打开的，用户仍要注意上述问题。

9.7 识别循环归纳变量

如果循环体中的变量和循环索引变量之间成线性关系，则 SWCC 可以把该变量优化成线性迭代的的变量，我们称之为“归纳变量”。 然而，由于变量类型之间的差异，优化有可能会导致归纳变量产生越界的情况，这将增加结果的不确定性。然而，出错的情况是很少见的，如果要确定是否因为该优化导致程序执行失败，可以尝试选项

-OPT:wrap_around_unsafe_opt=OFF

该选项的功能是关掉上述优化。