

Автоматическая векторизация в GCC для архитектуры PowerPC

исходный код программы на языке C:

```
#define N 16

void fbar (float *);
void ibar (int *);
void sbar (short *);

/* multiple loops */

foo (int n)
{
    float a[N+1];
    float b[N];
    float c[N];
    float d[N];
    int i;

    /* Strided access. Vectorizable on platforms that support load of strided
       accesses (extract of even/odd vector elements). */
    for (i = 0; i < N/2; i++){
        a[i] = b[2*i+1] * c[2*i+1] - b[2*i] * c[2*i];
        d[i] = b[2*i] * c[2*i+1] + b[2*i+1] * c[2*i];
    }
    fbar (a);
}
```

Строка компиляции: **ppc-linux-gnu-gcc -O3 -maltivec -ftree-vectorizer-verbose=1 -ftree-vectorize -S vect-1.c**

Каждая команда PowerPC имеет длину 32 бита. Первые 6 бит определяют команду, а остальные имеют различное значение, зависящее от команды. Тот факт, что команды имеют фиксированную длину, позволяет процессору выполнять их более эффективно. Поскольку команды PowerPC имеют длину только 32 бита, внутри команд, загружающих постоянные величины, в наличии имеется только 16 бит. Поэтому, так как адрес может быть до 64 бит в длину, мы должны загружать его небольшими порциями. Значок @ в ассемблере указывает ассемблеру использовать специальную форму.

Результат:

```
.file "vect-1.c"
```

```
.section      ".text"
.align 2
.globl foo
.type   foo, @function

foo:
# Переместить значение из Link Register (адрес возврата) в регистр 0
mflr 0

# Store Word with Update (сохранить значение регистра 1(биты 32...63), в
адрес памяти(EA) = <значение регистра 1>+<число -224>
# EA <- (1) - 224
# MEM(EA, 4) <- (1)32:63
# (1) <- EA
# это -- организация кадра стека и помещение текущего указателя стека (r1) на дно
кадра
stwu 1,-224(1)

# Load Immediate Shifted(непосредственная загрузка)
# Она загружает величину (биты 16-31 адреса LC1)
# сдвигает число на 16 бит налево и затем сохраняет результат в регистре 11
# Биты 16-31 регистра 11 содержат биты 16-31 адреса.
lis 11,.LC1@ha

lis 9,.LC0@ha

# Load Address
# la RT,SI(RS) (equivalent to: addi RT,RA,SI)
# if RA = 0 then RT <- EXTS(SI)
# else          RT <- (RA) + EXTS(SI)
# The sum (RA|0) + SI is placed into register RT.
# Поместить в 11 регистр сумму 11 регистра и битов 0:15 LC1
# в 11 регистре окажется полный адрес переменной .LC1
la 11,.LC1@l(11)

# в 10 регистре сумма значение 1 регистра + 16 (число)
# r10 = адрес b[0]
addi 10,1,16

# Store Word
# stw RS,D(RA)
# if RA = 0 then b <- 0
# else          b <- (RA)
# EA <- b + EXTS(D)
# MEM(EA, 4) <- (RS)32:63
# Let the effective address (EA) be the sum (RA|0)+ D. (RS)32:63 are
stored into the word in storage addressed by EA.
# биты 32:63 регистра 0 будут помещены по адресу значение регистра 1 + 228 (в
биты 0-31?)
# сначала мы отняли 224 потом прибавили 228, в итоге в регистре 0 лежит начальное
```

```
для функции значение регистра 1 + 4
# это -- помещение адреса возврата (взятого из LR) на вершину текущего кадра
стека.
    stw 0,228(1)

# прибавить к значению регистра 10 число 16 и положить результат в 8
# до этого в регистре 10 был регистр 1 увеличенный на 16
    # r8 = адрес b[4]
    addi 8,10,16

    # Load Vector Indexed
    # lvx vD, rA, rB
    # Let the effective address EA be the sum of the contents of
register rA, or the value '0' if rA is equal '0', and the contents of
register rB
    # Load the quadword in memory addressed by the EA into vD
# помещает в регистр v11 данные по адресу в r11
# это -- загрузка таблицы .LC1 в v11. .LC1 -- таблица переноса четных float
    lvx 11,0,11

# в 11 регистр помещается сумма значения 1 регистра и 16
    # r11 = адрес b[0]
    addi 11,1,16

# это -- загрузка b[0:3] в v13
    lvx 13,0,11

# в 11 регистр сумму значения 1 регистра и 80 -- адрес переменной c
    addi 11,1,80

# это -- загрузка c[0:3] в v10
    lvx 10,0,11

    addi 11,1,96
    # c[4:7] в v1
    lvx 1,0,11

    addi 11,1,112
    # c[8:11] в v8
    lvx 8,0,11

# Поместить в 9 регистр сумму 9 регистра и битов 0:15 LC0
# в 9 регистре окажется полный адрес переменной .LC0
    la 9,.LC0@l(9)

    addi 11,1,128

# это -- загрузка таблицы .LC0 в v7
    lvx 7,0,9

    # b[4:7] в v6
```

```
lvs 6,0,8

# r9 = адрес b[12]
addi 9,8,32

# c[12:15] в v0
lvs 0,0,11

# r8 = адрес b[8]
addi 8,8,16

# b[12:5] в v4
lvs 4,0,9

# r9 = адрес a
addi 9,1,144

# b[8:11] в v12
lvs 12,0,8

# mr Rx, Ry на самом деле or Rx,Ry,Ry
# or RA, RS, RB
# RA <- (RS) | (RB)
mr 3,9

# Vector Permute
# vperm vD, vA, vB, vC
# temp[0:255] <- (vA) || (vB) // || --- конкатенация
# do i=0 to 127 by 8
#   b <- (vC)[i+3:i+7] || 0b000
#   (vD)[i:i+7] <- temp[b:b+7]
# end
# Let the source vector be the concatenation of the contents of
register vA followed by the contents of register vB.
# For each integer i in the range 0-15, the contents of the byte
element in the source vector specified in bits [3-7]
# of byte element i in vC are placed into byte element i of register
vD.
# судя из картинки в документации работает так:
# vA и vB -- исходные вектора по 16 8-битных элементов. после их
конкатенции все элементы пронумерованы от 0x00 до 0x1F
# в vC[i] написано число от 0x00 до 0x1F и означает какой из 32 элементов vA||vB
положить в vD[i]

# это -- помещение c[8,10,12,14] в v3
vperm 3,8,0,11
# это -- помещение c[0,2,4,6] в v5
vperm 5,10,1,11
# это -- помещение b[0,2,4,6] в v9
```

```

vperm 9,13,6,11

# это -- помещение c[9,11,13,15] в v8
vperm 8,8,0,7
# это -- помещение c[1,3,5,7] в v10
vperm 10,10,1,7
# это -- помещение b[1,3,5,7] в v13
vperm 13,13,6,7

# Vector Splat Immediate Signed Word
# vspltisw vD, SIMM // SIMM ( 11-15) this Immediate field is used
to specify a (5 bit) signed integer
# do i=0 to 127 by 32
# (vD)[i:i+31] <- SignExtend(SIMM,32)
# end
# Each element of wspltisq is a word. The value of the SIMM field,
sign-extended to 32 bits, is replicated into each element of register vD
# регистр ноль будет забит числами -1
vspltisw 0,-1

# Vector Shift Left Integer Word
# wslw vD, vA, vB
# do i=0 to 127 by 32
# sh <- (vB)[i+27:i+31]
# (vd)[i:i+1] <- (vA)[i:i+31] <<ui sh
# end
# Each element a word. Each word element in register vA is shifted
left by the number of bits specifed in the
# low-order 5 bits of the corresponding word element in register vB.
Bits shifted out to bit [0] of the word
# element are lost. Zeros are supplied to the vacated bits on the
roght. The result is placed into the corresponding word element of register
vD
# Каждое слово в регистре сдвигается влево на 31 бит, получается 0x80000000

vslw 0,0,0

# Vector Multiply Add Floating Point
# vmaddfp vD, vA, vC, vB
# vD[i] = vA[i]*vC[i]+vB[i]

# v13 = b[1,3,5,7]*c[1,3,5,7]+0x80000000
vmaddfp 13,13,10,0
# v9 = b[0,2,4,6]*c[0,2,4,5]+0x80000000
vmaddfp 9,9,5,0

# Vector Substract Floating Point
# vsubfp vD, vA, vB
# vD[i] = vA[i]-vB[i]

```

```
# v13 = b[1,3,5,7]*c[1,3,5,7]-b[0,2,4,6]*c[0,2,4,5]
vsubfp 13,13,9

# Store Vector Indexed
# stvx vS,rA,rB
# if rA=0 then b ← 0
# else b ← (rA)
# EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFF0
# MEM(EA,16) ← (vS)
# Let the effective address EA be the result of ANDing the sum of
the contents of register rA, or the value '0' if
# rA is equal to '0', and the contents of register rB with
0xFFFF_FFFF_FFFF_FFF0.
# The contents of register vS are stored into the quadword addressed
by EA. Figure 6-5 shows how a store
# instruction is performed for a vector register.

# a[0]=b[1]*c[1]-b[0]*c[0]
# a[1]=b[3]*c[3]-b[2]*c[2]
# a[2]=b[5]*c[5]-b[4]*c[4]
# a[3]=b[7]*c[7]-b[6]*c[6]
stvx 13,0,9

# r9 -- адрес a[4]
addi 9,9,16

# это -- помещение b[8,10,12,14] в v11
vperm 11,12,4,11
# это -- помещение b[9,11,13,15] в v12
vperm 12,12,4,7

# v11 = b[8,10,12,14]*c[8,10,12,14]+0x8000000
vmaddfp 11,11,3,0
# v12 = b[9,11,13,15]*c[9,11,13,15]+0x8000000
vmaddfp 12,12,8,0
# v12 = b[9,11,13,15]*c[9,11,13,15]-b[8,10,12,14]*c[8,10,12,14]
vsubfp 12,12,11

# a[4]=b[9]*c[9]-b[8]*c[8]
# a[5]=b[11]*c[11]-b[10]*c[10]
# a[6]=b[13]*c[13]-b[12]*c[12]
# a[7]=b[15]*c[15]-b[14]*c[14]
stvx 12,0,9

# Vector Logical OR (0x1000 0484)
# vor vD,vA,vB
vor 1,0,0
```

```
# Вызов функции fbar
bl fbar
# загрузка адреса возврата с вершины кадра стека в r0
lwz 0,228(1)
# восстановление предыдущего кадра стека
addi 1,1,224
    # Move To Link Register
mtlr 0
# возврат из функции
blr
    .size    foo, .-foo
    .section .rodata.cst16,"aM",@progbits,16
    .align  4
.LC0:
    .byte   4
    .byte   5
    .byte   6
    .byte   7
    .byte  12
    .byte  13
    .byte  14
    .byte  15
    .byte  20
    .byte  21
    .byte  22
    .byte  23
    .byte  28
    .byte  29
    .byte  30
    .byte  31
.LC1:
    .byte   0
    .byte   1
    .byte   2
    .byte   3
    .byte   8
    .byte   9
    .byte  10
    .byte  11
    .byte  16
    .byte  17
    .byte  18
    .byte  19
    .byte  24
    .byte  25
    .byte  26
    .byte  27
    .ident  "GCC: (GNU) 4.3.0 20080202 (experimental)"
    .section .note.GNU-stack,"",@progbits
```

Last update: 2008/02/04 04:31 etc:common_activities:gcc_vectorization:autovect_ppc http://wiki.osll.ru/doku.php/etc:common_activities:gcc_vectorization:autovect_ppc

From: <http://wiki.osll.ru/> - **Open Source & Linux Lab**

Permanent link: http://wiki.osll.ru/doku.php/etc:common_activities:gcc_vectorization:autovect_ppc

Last update: **2008/02/04 04:31**

