Notes on Using GCC NEON Auto-vectorization

Table of Contents

1 NOTES ON USING GCC NEON AUTO-VECTORIZATION

1.1 Turning on auto vectorization

To switch on auto vecorization for the compiler, the following options should be given:

0) use softfp or hard vfp toolchain (or add *-mfloat-abi=softfp* or *-mfloat-abi=hard*);

1) add ***-mfpu=neon*** (must);

2) under ***-O3***, or "***-O2 -ftree-vectorize***";

3) to utilize NEON 128-bit register, add *-mvectorize-with-neon-quad* (better performance for most cases).

To view the detailed optimization information, use the additional option:

-ftree-vectorizer-verbose=<number>

use *-ftree-vectorizer-verbose=2* to show vectorized and unvectorized loops (=1 only shows vectorized loop).

1.2 Hint: use "restrict" attribute for function pointer argument when possible

```
void foo(char *p, char *q)
{
    int i;
    for (i = 0; i < 1000; i++)
        *p++ = *q++ + 1;
}
```

Compiler needs to analyze memory access via pointer and array access, and check if the vectorization is safe or not, if it cannot determinate it in compile time, it will add runtime check code and a non-vectorized form loop which will increase the code size. And if there are too many dependences need to be checked, compiler will give up vectorization on that loop. So, if it possible, use __restrict attribute when declare function arguments.

The compile result without __restrict attribute is:

```
foo:
    add     r3, r0, #8
    add     r2, r1, #8
    cmp     r1, r3
    cmpls   r0, r2
    bls     .L2
    vmov.i8     d17, #1   @ v8qi
```

```
    mov     r3, r1
    add     r1, r1, #1000
    rsb     r0, r3, r0
.L3:
    vld1.8    {d16}, [r3]
    add     r2, r0, r3
    vadd.i8     d16, d16, d17
    add     r3, r3, #8
    cmp     r3, r1
    vst1.8    {d16}, [r2]
    bne     .L3
    mov     r0, #0
    bx      lr
.L2:
    sub     r3, r1, #1
    add     r1, r0, #1000
.L5:
    ldrb    r2, [r3, #1]!    @ zero_extendqisi2
    add     r2, r2, #1
    strb    r2, [r0], #1
    cmp     r0, r1
    bne     .L5
    mov     r0, #0
    bx      lr
```

The red part is the alias check code, and the blue part is the non-vectorized form loop.

If define the function like:

```
void foo(char * __restrict p, char * __restrict q)
```

…

The compile result is :

```
foo:
     vmov.i8     d17, #1   @ v8qi
    add     r2, r1, #1000
    rsb     r0, r1, r0
    mov     r3, r1
.L2:
    vld1.8    {d16}, [r3]
    add     r1, r0, r3
    vadd.i8     d16, d16, d17
    add     r3, r3, #8
    cmp     r3, r2
    vst1.8    {d16}, [r1]
    bne     .L2
    mov     r0, #0
```

```
bx    lr
```

Note: *-fargument-noalias-global* and *–fargment-noalias-anything* can also help to save some runtime check code; but they're aggressive optimization options, and sometimes the result will be wrong if no-alias relation can't be guaranteed..

1.3 Avoid complicated memory access in loop

When Compiler vectorizes a loop, it needs to analyze all memory access patterns in that loop. If the access pattern is too complicated, it will give up vectorization.
There are 3 types of memory access in GCC,

- *Component reference*: access member of structure
- *Array reference*: access element in an array
- *Deference*: access memory via pointer

1.3.1 The base pointer is a component ref.

For example:

```
typedef struct _Stream {
   …
   unsigned char * pBsCurByte;------Pointer
   } Stream;

typedef struct _DecoderState {
   unsigned char *pPLTE;-------Pointer
     …
   } DecoderState;

Void foo(Stream *pStream, DecoderState *pDecoderState)
{
     for(i = 0; i < N; i++) {
         pDecoderState->pPLTE[i] = *pStream->pBsCurByte++;
     }
}
```

Actually for each memory access in this loop, compiler will generate two memory access instructions: first access the structure to find the address, and then access the memory based on the pointer value. In this case, Compiler cannot add code to check the alias between those pointers, and it won't vectorize this loop. So if you know this value won't be changed in that loop, just get

it out of loop.

Like:

```
unsigned char *pDst = pDecoderState->pPLTE;
unsigned char *pSrc = pStream->pBsCurByte;
For (i = 0; i < N; i++)
    pDst[i] = *pSrc++;
pStream->pBsCurByte = pSrc;
```

1.3.2 Base Pointer come from an array address

If pointer actually is an array address, don't get its value out of loop, otherwise compiler cannot recognize the memory access pattern.

For example:

```
typedef struct _DecoderParam {
    …
    ClrTbl GlbClrTbl[256];-------Array
}DecoderParam;

typedef struct _DecoderState {
    ClrTbl *GlbClrTbl; -----------Pointer
} DecoderState;

 Void foo (DecoderParam *pDecoderPar, DecoderState *pState)
{
 ClrTbl *GlbClrTbl = pState->GlbClrTbl;
 ClrTbl *ParClrTbl = pDecoderPar->GlbClrTbl;

   for (i = 0; i < N; i++) {
      ParClrTbl [i].red = GlbClrTbl[i].red;
      ParClrTbl [i].green = GlbClrTbl[i].green;
      ParClrTbl [i].blue = GlbClrTbl[i].blue;
    }
}
```

This loop cannot be vectorized because compiler cannot determinate the memory access pattern of ***ParClrTbl[i].red***, it is array ref, but now it is access via pointer. This isn't equal in Compiler. If written like this, it can be vectorized:

```
  for (i = 0; i < N; i++) {
     pDecoderPar->GlbClrTbl [i].red = GlbClrTbl[i].red;
     pDecoderPar->GlbClrTbl [i].green = GlbClrTbl[i].green;
     pDecoderPar->GlbClrTbl [i].blue = GlbClrTbl[i].blue;
  }
```

Note: this isn't against example 1 since the array address is a constant, cannot be changed during

the loop execution.

1.3.3 Loop Counter is a component reference

Don't use component reference as loop counter, please save it in a temp variable before loop. Otherwise Compiler cannot know this value is invariant in loop.

For example

```
struct A
{
    int counter;
    char aa;
};
void foo(char *p, char *q,  struct A* a)
{
    int i;
    For (i = 0; i < a->counter; i++)
    *p++ = *q++ +1;
}
```

Compiler cannot know `a->counter` is an invariant unless you use a temp variable to save its value before loop

Like:

```
    int counter = a->counter;
    for (i = 0; i < counter ; i++)
        *p++ = *q++ + 1;
```

1.4 Avoid abnormal loop form

1.4.1 Write canonical loop

When vectorize a loop, Compiler not only need analyze the memory access pattern in loop, but also need analyze the exit condition. So try to write loop in a standard way.

For example:

```
  do {
    *pOut++ = *pFrom++;
  } while (--temp && --len);
```

This loop cannot be vectorized since it has two exit conditions, but this one is easy to change to one condition. Just get the minimum of `temp` and `len`, and use it as loop counter.

1.4.2 Avoid control flow in loop

Current compiler only can handle straight line code sequence. If loop contain control flow, it

only can transform simple if-else statement to select statement.

For Example:

```
void foo(int *sign, int *sign1, short *pToepLiz1)
{
    int j, k;
    int *pS;
    short sTmp;
    int lTmp;

        for(k = 1; k < N; k += 2) {
            if(pS[k] < 0) {
                *pToepLiz1 = (short)(- *pToepLiz1);
            } else {
                sTmp = *pToepLiz1;
                lTmp = sTmp * 32767;    /* Just for bit-by-bit purpose */
                *pToepLiz1 = (short)(lTmp >> 15);
            }
            pToepLiz1 ++;
        }
}
```

We need to change this loop a lot in order to make it been vectorized. here to emphasis one, control flow. Now the compiler's ability to transform *if-else* is limited, here is the limitation:

1. Don't access memory inside *if-else* block especially if one memory is only access in one branch. That means all load/store `*pToepLiz1` should be taken place outside loop.
2. The *if-else* should be simple. The better way is write it in a conditional select form.
3. Now GCC only support vectorized condition execution in same element type, that means `pS[k]` is same type as `*pToepliz1` (This limitation may be removed after GCC vectorizer has been enhanced).

If written the loop this way, it can be vectorized:

```
    for(k = 1; k < N; k += 2) {
        int t0, t1, t2;
        t1 = (- *pToepLiz1);
        sTmp = *pToepLiz1;
        Tmp = sTmp * 32767;    /* Just for bit-by-bit purpose */
        T2 = (lTmp >> 15);
           if(pS[k] < 0) {
                t0 = t1;
           }
        else {
                t0 = t2;
           }
        *pToepLiz1 = (short)t0;
```

```
        pToepLiz1 ++;
    }
```

1.5 Avoid non-supported memory access stride

Memory access stride is the gap between two iterations for one memory access.

For example:

```
for (i = 0; i < N; i++)
{
  a[i]= b[2*i]+c[3*i]+d[5*i]
}
```

Here stride of `a` is 1, stride of `B` is 2, stride of `C` is 3, stride of `d` is 5.

Sometimes the stride isn't clear as this way.

Like

```
for(i = 0; i < N; i++)
{
  *p++ = *q++;
  *p++ = *q++;
  *p++ = *c++;
}
```

Here stride of each `p` is 3, stride of each `q` is 2, stride of `c` is 1.

Or this form:

```
for (i = 0; i < N; i += skip)
{
  a[i] = …
}
```

Here stride of `a` is `skip`.

NEON supports memory access stride 1,2,3,4, or stride > 4 but is a power of 2.

For example:

```
short const_parameter[] = {11,32,.....};
for (i = 0; i < N; i++, k += 5 ) {
      sumL += pSig[i] * const_parameter[k];
}
```

Here the stride of const_parameter is 5, and it is not supported by NEON. As this array is an static const array, so maybe developer can add another array which save value of `const_parameter[0,5,10,15…]`.