Using GCC Auto-Vectorizer

Ira Rosen <ira.rosen@linaro.org>
Michael Hope <michael.hope@linaro.org>

r1

bzr branch lp:~michaelh1/+junk/using-the-vectorizer

Using GCC Vectorizer

- Vectorization is enabled by the flag -ftree-vectorize and by default at -O3:
  - gcc –O2 –ftree-vectorize myloop.c
  - or gcc –O3 myloop.c
- To enable NEON:

  -mfpu=neon -mfloat-abi=softfp or -mfloat-abi=hard
- Information on which loops got vectorized, and which didn't and why:
  - -fdump-tree-vect(-details)
    - dumps information into myloop.c.##t.vect
  - -ftree-vectorizer-verbose=[X]
    - dumps to stderr
- More information: http://gcc.gnu.org/projects/tree-ssa/vectorization.html

Other useful flags

- -ffast-math - if operating on floats in a reduction computation (to allow the vectorizer to change the order of the computation)
- -funsafe-loop-optimizations - if using "unsigned int" loop counters (can be assumed not to overflow)
- -ftree-loop-if-convert-stores - more aggressive if-conversion
- --param min-vect-loop-bound=[X] - if have loops with a short trip-count
- -fno-vect-loop-version- if worried about code size

What's vectorizable

- Innermost loops
  - countable
  - no control flow
  - independent data accesses
  - continuous data accesses

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

Example of not vectorizable loop:

```
while (a[i] != 8)        uncountable
  {
   if (a[i] != 0)        control flow
     a[i] = a[i-1];      loop carried dependence
   b[i+stride] = 0;
                         access with unknown stride
  }
```

Special features

- vectorization of outer loops
- vectorization of straight-line code
- if-conversion
- multiple data-types and type conversions
- recognition of special idioms (e.g. dot-product, widening operations)
- strided memory accesses
- cost model
- runtime aliasing and alignment tests
- auto-detection of vector size

Examples:
http://gcc.gnu.org/projects/tree-ssa/vectorization.html

GCC Versions

- Current Linaro GCC is based on FSF GCC 4.6
- Once FSF GCC 4.7 is released (in about six months) Linaro GCC will switch to GCC 4.7
- Some of GCC 4.7 vectorizer related features:
  - __builtin_assume_aligned – alignment hints
  - vectorization of conditions with mixed types
  - vectorization of bool

Special features

- vectorization of outer loops
- vectorization of straight-line code
- if-conversion
- multiple data-types and type conversions
- recognition of special idioms (e.g. dot-product, widening operations)
- strided memory accesses
- cost model
- runtime aliasing and alignment tests
- auto-detection of vector size

Examples:
http://gcc.gnu.org/projects/tree-ssa/vectorization.html

Vectorizing for NEON

libswscale/rgb2rgb_template.c:

```
static inline void rgb24tobgr16_c(
const uint8_t *src, uint8_t *dst,
int src_size) {
    const uint8_t *s = src;
    const uint8_t *end;
    uint16_t *d = (uint16_t *)dst;
    end = s + src_size;
    while (s < end) {
        const int b = *s++;
        const int g = *s++;
        const int r = *s++;
        *d++ = (b>>3) | ((g&0xFC)<<3)
                   | ((r&0xF8)<<8);
} }
```

strided access

& and shift right
performed on u8

widening shift

no over-promotion to s32

```
.L12:
mov       r1, r4
add       r0, r0, #1
vld3.8    {d16, d18, d20}, [r1]!
cmp       r0, r6
mov       r2, ip
add       r4, r4, #48
add       ip, ip, #32
vld3.8    {d17, d19, d21}, [r1]
vand      q12, q9, q14
vshr.u8   q11, q8, #3
vand      q8, q10, q13
vshll.u8  q9, d25, #3
vmovl.u8 q10, d22
vshll.u8  q15, d24, #3
vmovl.u8 q11, d23
vorr        q12, q15, q10
vorr        q11, q9, q11
vshll.u8    q10, d16, #8
vshll.u8    q9, d17, #8
vorr        q8, q12, q10
vorr        q11, q11, q9
vst1.16   {q8}, [r2]!
vst1.16   {q11}, [r2]
bcc     .L12
```

scalar: 75000 runs take 216.583ms
vector: 75000 runs take 48.8586ms
speedup: 4.433x

Writing vectorizer-friendly code

- Avoid aliasing problems
  - Use __restrict__ qualified pointers

```
void foo (int *__restrict__ pInput, int *__restrict__ pOutput)
```

- Don't unroll loops
  - Loop vectorization is more powerful than SLP

```
for (i=0; i<n; i+=4) {
  sum += a[0];
  sum += a[1];
  sum += a[2];
  sum += a[3];
  a += 4;}
```

$\longrightarrow$

```
for (i=0; i<n; i++)
   sum += a[i];
```

Writing vectorizer-friendly code (cont.)

- Use countable loops, with no side-effects
  - No function-calls in the loop (distribute into a separate loop)

```
for (i=0; i<n; i++) {
  if (a[i] == 0) foo();
  b[i] = c[i]; }
```

$\longrightarrow$

```
for (i=0; i<n; i++)
  if (a[i] == 0) foo();
for (i=0; i<n; i++)
  b[i] = c[i];
```

  - No ‘break’/’continue’

```
for (i=0; i<n; i++) {
  if (a[i] == 8) break;
  b[i] = c[i]; }
```

$\longrightarrow$

```
for (i=0; i<n; i++)
  if (a[i] == 8) {m = i; break;}
for (i=0; i<m; i++)
  b[i] = c[i];
```

Writing vectorizer-friendly code (cont.)

- Keep the memory access-pattern simple
  - Don't use indirect accesses, e.g.:

```
for (i=0; i<n; i++)
   a[b[i]] = x;
```

  - Don't use unknown stride, e.g.:

```
for (i=0; i<n; i++)
   a[i+stride] = x;
```

- Use "int" iterators rather than "unsigned int" iterators
  - The C standard says that the former cannot overflow, which helps the compiler to determine the trip count.

Some of our recent contributions

- Support of vldN/vstN
- NEON specific patterns: e.g. widening shift
- SLP (straight-line code vectorization) improvements
- RTL improvements:
  - reducing the number of moves and amount of spilling (both for auto- and hand-vectorised code)
  - improving modulo scheduling of NEON code

People

- Linaro Toolchain WG
- Ira Rosen (IRC: irar)

  ira.rosen@linaro.org

  – auto-vectorizer
- Richard Sandiford (IRC: rsandifo)
  richard.sandiford@linaro.org

  – NEON back-end/RTL optimizations

Helping us

Send us examples of code that are important to you to vectorize.

Output Example

ex.c:

```
1 #define N 128
2 int a[N], b[N];
3 void foo (void)
4   {
5      int i;
6
7      for (i = 0; i < N; i++)
8         a[i] = i;
9
10     for (i = 0; i < N; i+=5)
11        b[i] = i;
12  }
```

- What's got vectorized:

```
gcc -c -O3 -ftree-vectorizer-verbose=1 ex.c

ex.c:7: note: LOOP VECTORIZED.
ex.c:3: note: vectorized 1 loops in function.
```

- What's got vectorized and what didn't:

```
gcc -c -O3 -ftree-vectorizer-verbose=2 ex.c

ex.c:10: note: not vectorized: complicated access pattern.
ex.c:10: note: not vectorized: complicated access pattern.
ex.c:7: note: LOOP VECTORIZED.
ex.c:3: note: vectorized 1 loops in function.
...
```

- All the details:

```
gcc -c -O3 -ftree-vectorizer-verbose=9 ex.c
  or
gcc -c -O3 -fdump-tree-vect-details ex.c
```