

组会报告

徐益

2018 年 9 月 20 日

1 工作内容

1. 数据采集测试;
2. 低时延 AVX2-LDPC 译码实现;

2 数据采集测试

2.1 ramdisk

2.1.1 传统 ramdisk

```
1 # mkdir /mnt/test
2 # mke2fs /dev/ram0
3 # mount /dev/ram /mnt/test
```

写入速率: 700MB/s 800MB/s

2.1.2 ramfs

```
1 # mkdir /testRam
2 # mount -t ramfs none /testRAM
3 # mount -t ramfs none /testRAM -o maxsize=2000
```

写入速率: 900MB/s 1100MB/s

2.1.3 tmpfs

```
1 # mkdir -p /mnt/tmpfs
2 # mount tmpfs /mnt/tmpfs -t tmpfs
3 # mount tmpfs /mnt/tmpfs -t tmpfs -o size=32g
```

写入速率: 1.2GB/s 1.3GB/s

2.2 测试结果

```
mimo5g1@mimo5g1-sever: /media/mimo5g1/0c73c5cf-4c13-4876-8bc1-d3b3e21a0254/data_
Port statistics =====
Statistics for port 0 -----
Packets sent: 0
Packets received: 10077337
Packets dropped: 0
Aggregate statistics =====
Total packets sent: 0
Total packets received: 10077337
Total packets dropped: 0
=====
err_pkg_times = 8663, err_pkg_num = 1909364, nvld_pkg_num=10070581
send_rate = 0.00Gbps, receive_rate = 4.24Gbps
mimo5g1@mimo5g1-sever: /media/mimo5g1/0c73c5cf-4c13-4876-8bc1-d3b3e21a0254/data_c
ollection$
```

图 1: 向 ssd 写入

```
mimo5g1@mimo5g1-sever: /mnt/tmpfs/data_collection
Port statistics =====
Statistics for port 0 -----
Packets sent: 0
Packets received: 12338169
Packets dropped: 0
Aggregate statistics =====
Total packets sent: 0
Total packets received: 12338169
Total packets dropped: 0
=====
err_pkg_times = 0, err_pkg_num = 0, nvld_pkg_num=12338169
send_rate = 0.00Gbps, receive_rate = 5.03Gbps
mimo5g1@mimo5g1-sever: /mnt/tmpfs/data_collection$
```

图 2: 向 tmpfs 写入

The image shows a terminal window with a dark background. The top part of the window displays a hex dump of data, with each line containing a decimal offset followed by two columns of hexadecimal values. The data appears to be a sequence of bytes, possibly representing a file or a network packet. The bottom part of the window shows a terminal prompt where a user has entered the command 'make' to compile a program. The output of the compilation is visible, showing the compiler (gcc) and the resulting executable file (main.o). The terminal window also has a status bar at the bottom indicating the current line and column, as well as the encoding (UTF-8) and the text mode (LF).

```
1  FF FF 00 04 00 05 00 00 FF FE FF FE FF FD FF FD 00 00 FF FF 00 01 00 02 00 00 FF FE 00 02 FF FD 00 01 FF FF FF FE 00 02 00 01 FF FE
2  00 00 00 00 00 00 FF FD 00 05 FF FE FF FF 00 00 00 01 00 01 00 00 00 00 03 00 01 00 03 00 02 FF FF 00 02 FF FA 00 01 FF FD FF FB
3  FF FD 00 00 00 01 FF FD 00 01 FF FE FF FD 00 01 FF FF 00 01 FF FD FF FD 00 00 00 01 00 01 00 03 FF FE 00 01 00 02 FF FC 00 05 00 00
4  00 01 FF FE FF FF FC 00 01 00 02 00 01 FF FF 00 02 00 01 00 03 00 02 00 04 00 01 00 01 FF FF 00 05 FF FE 00 00 03 00 00 00 01
5  00 00 00 01 00 02 00 04 00 01 00 03 FF FD 00 01 FF FF FC 00 03 00 02 FF FC 00 00 00 01 FF FE 00 03 00 05 FF FD 00 01 00 04 00 00
6  00 03 00 02 00 01 00 00 FF FF 00 02 FF FE 00 04 00 01 FF FD 00 03 00 03 FF FB 00 00 00 04 00 00 FF FE 00 04 FF F9 FF FE FF FE 00 02
7  00 00 00 00 00 00 FF FE 00 03 FF FE 00 02 00 03 00 00 00 00 00 00 00 00 02 00 03 00 01 00 07 FF F9 00 02 00 00 FF FD 00 03 00 01
8  00 02 00 00 01 00 00 00 01 00 01 00 00 FF FF 00 01 00 00 00 00 FF FF 00 03 FF FC 00 03 FF FD 00 03 00 03 00 00 00 00 FF FE 00 00
9  FF FA 00 06 FF FB FF FD 00 05 00 01 FF FC 00 04 FF FD 00 01 FF FF 00 00 00 00 FF FD FF FF FF FF 00 00 00 00 00 01 00 02 00 01 00 00
10 FF FE 00 01 FF FE 00 02 FF FE 00 06 FF FE 00 00 03 FF FC 00 02 00 03 FF FC FF FE FF FE FF FE 00 00 00 06 00 00 00 00 01 FF FF
11 00 02 00 09 FF FC FF FE 00 01 FF FD 00 01 00 00 FF FE FF 00 02 00 03 FF FE 00 03 FF FD FF FF 00 01 00 01 FF FE 00 00 02 FF FD
12 FF FC 00 01 FF FE FF FD 00 02 00 03 FF FB 00 06 FF FA FF FE 00 02 FF FF FF FF 00 03 00 01 FF FC 00 01 FF FA FF FF FF FD FF FC 00 00
13 FF FE FF FE 00 03 00 00 00 01 00 00 00 07 FF FB 00 04 FF FD 00 03 FF FA 00 02 FF FC 00 00 00 00 FF FE 00 00 FF FE FF FC 00 02 FF FC
14 00 02 FF FD 00 00 FF FE FF FF FF FE 00 00 00 01 FF FC FF FE 00 01 00 00 00 03 00 03 FF FC FF FF 00 03 FF FF 00 00 00 02
15 00 02 00 01 FF FD FF FB 00 03 FF FC 00 03 00 00 FF FE 00 02 00 02 00 00 00 00 07 FF FD 00 02 00 04 00 00 00 02 00 04 FF FF FF FF
16 00 01 00 01 FF FC FF FC 00 03 FF FE 00 02 00 02 00 01 FF FC 00 02 FF FD 00 00 00 01 FF FE FF FE 00 04 00 02 FF FE 00 04 FF FC FF FF
17 00 00 FF FB 00 05 00 00 FF FF FF FE 00 04 FF FC 00 02 00 00 FF FC FF FB 00 02 FF FF FF 00 00 FF FF 00 01 FF FF 00 01 00 00 00 00
18 00 02 FF FC 00 01 00 02 FF FE FF FC 00 02 FF FF 00 01 00 01 FF FF FF FF 00 01 FF FF 00 02 00 00 FF FE 00 01 00 01 00 01 FF FF 00 03
19 FF FF 00 02 00 01 00 06 FF FC 00 03 FF FB 00 00 FF FC FF FE 00 00 FF FE 00 02 00 01 00 04 FF FF 00 06 FF FF FE 00 00 FF FB FF FF
20 00 00 00 02 00 00 00 01 FF FF 00 00 00 01 00 01 00 01 00 01 FF FE 00 02 00 01 00 02 00 00 00 01 00 04 FF FC 00 01 00 03 FF FD FF FD
21 FF FE 00 01 FF FB 00 01 FF FE 00 01 FF FD FF FE 00 03 FF FB 00 03 00 02 FF FF 00 02 00 01 00 00 00 02 00 03 00 02 00 01 00 03 FF FB
22 FF FE FF FC 00 05 00 00 00 01 00 05 FF FD FF FF FD 00 02 FF FD 00 01 00 00 00 00 01 00 03 00 00 00 02 FF FD FF FE 00 04 FF FB
23 FF FC FF FC FF FE FF FE FF FF 00 00 00 05 00 02 FF FE 00 00 00 01 FF FF FF FF FF FE FF FC FF FD 00 01 FF FF 00 01 00 00 00 00 00 01
24 00 03 00 00 02 FF FF FF 00 05 FF FB 00 05 00 02 FF FF 00 01 00 00 00 00 FF FF FF FE 00 03 00 01 00 02 FF FE 00 07 FF FF 00 01 00 03
25 FF FF 00 01 FF FC FF FD 00 03 00 01 FF FF 00 05 00 02 FF FE 00 00 FF FF FF FF FF FF FD 00 01 00 01 FF FC 00 03 00 02 FF FE 00 01
26 00 08 00 01 00 03 00 03 00 01 00 02 00 01 00 01 00 00 FF FF 00 02 FF FC FF FF FF FE FF FF 00 01 FF FF 00 01 FF FE FF FF 00 01 00 01
27 00 03 00 01 00 04 00 01 FF FF 00 03 FF FE 00 02 00 02 00 00 00 01 00 03 00 00 00 03 FF FF FF FE 00 03 00 00 00 04 00 05 00 00 FF FF
28 FF FC FF FE 00 01 FF FF FF FD 00 04 FF FE FF FF 00 01 FF FE FF FF 00 01 00 00 FF FF FF FF 00 03 00 02 FF FE 00 02 00 01 FF FF 00 01
29 00 00 00 02 FF FE FF FE 00 00 FF FE 00 01 FF FF 00 00 FF FE 00 02 00 02 00 02 00 06 FF FA FF FF FF FB FF FE 00 02 00 00 FF FD FF FF
30 FF FD 00 03 FF FE 00 02 00 00 00 00 FF FC FF FC FF FD FF FC FF FE FF FD FF FF 00 00 FF FD 00 01 FF FF FF FD 00 02 00 00 FF FE 00 01
31 FF FF 00 00 FF FE FF FD 00 03 FF FF 00 00 00 01 00 00 FF FE 00 01 FF FF 00 02 FF FA 00 09 FF FE 00 02 00 03 FF FE 00 04 FF FD FF FC
32 00 04 00 02 FF FE FF FE 00 00 FF FF FF FF FF FE FF FF 00 01 00 04 00 01 00 04 FF FD 00 02 FF FF 00 00 FF FD FF FE 00 01 FF F9
33 00 04 00 01 FF FC 00 00 FF FF FF F9 00 02 FF FF FF FE FF FF FF FB FF FE 00 00 FF FE 00 02 00 01 FF FD FF FD 00 02 FF FF 00 00 FF FF
34 00 07 FF FC 00 06 00 00 00 00 FF FB 00 03 FF FF 00 01 00 03 FF FC 00 02 00 01 FF FE 00 01 00 01 00 00 FF FF 00 04 FF FC 00 03 00 00
35 00 01 FF FD 00 04 00 00 00 01 00 01 FF FE FF FC FF FF FF FD 00 03 00 00 00 06 FF FD 00 01 00 02 FF FB 00 03 00 01 00 01 FF FE
```

```
mimo5g1@mimo5g1-sever:/media/mimo5g1/0c73c5cf-4c13-4876-8bc1-d3b3e21a0254/data_read$ make
gcc -c -o main.o main.c
gcc -o main main.o
mimo5g1@mimo5g1-sever:/media/mimo5g1/0c73c5cf-4c13-4876-8bc1-d3b3e21a0254/data_read$ ./main
mimo5g1@mimo5g1-sever:/media/mimo5g1/0c73c5cf-4c13-4876-8bc1-d3b3e21a0254/data_read$ ./main
mimo5g1@mimo5g1-sever:/media/mimo5g1/0c73c5cf-4c13-4876-8bc1-d3b3e21a0254/data_read$ ./main
mimo5g1@mimo5g1-sever:/media/mimo5g1/0c73c5cf-4c13-4876-8bc1-d3b3e21a0254/data_read$
```

图 3: 转码结果

3 低时延 AVX2-LDPC 译码实现

3.1 设计系统时遇到的问题

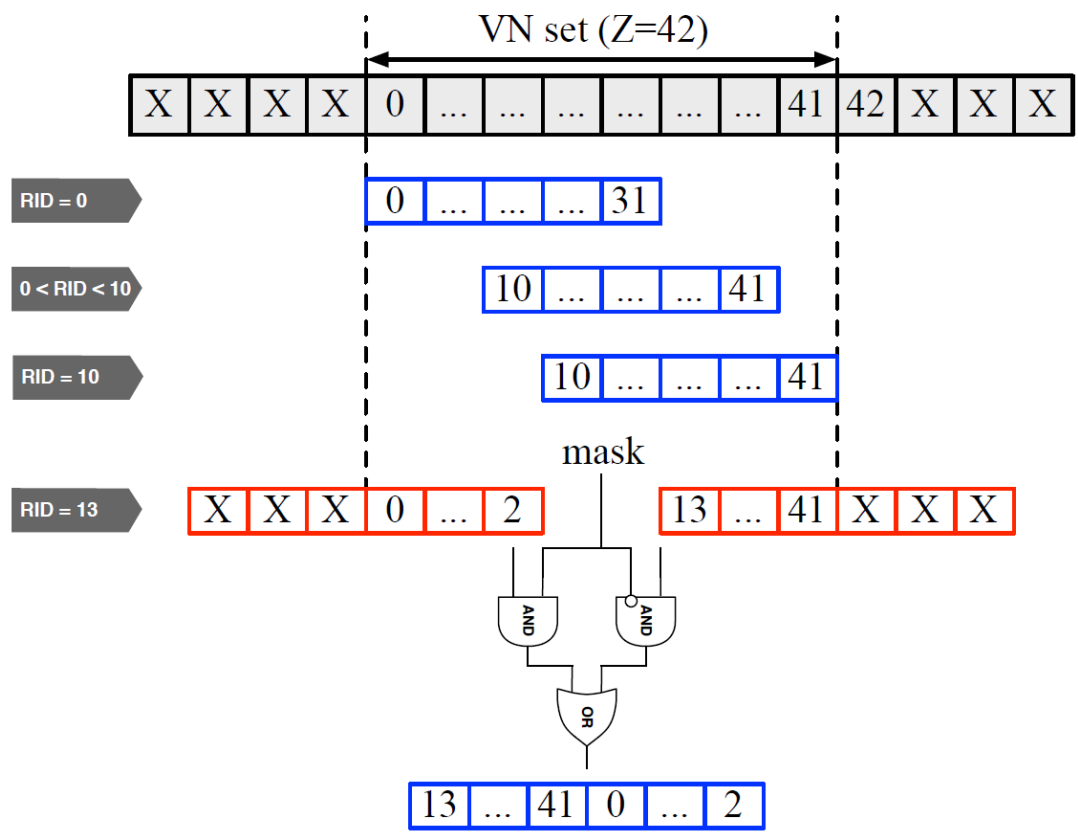


图 4: 论文提到的 VN 接入方式

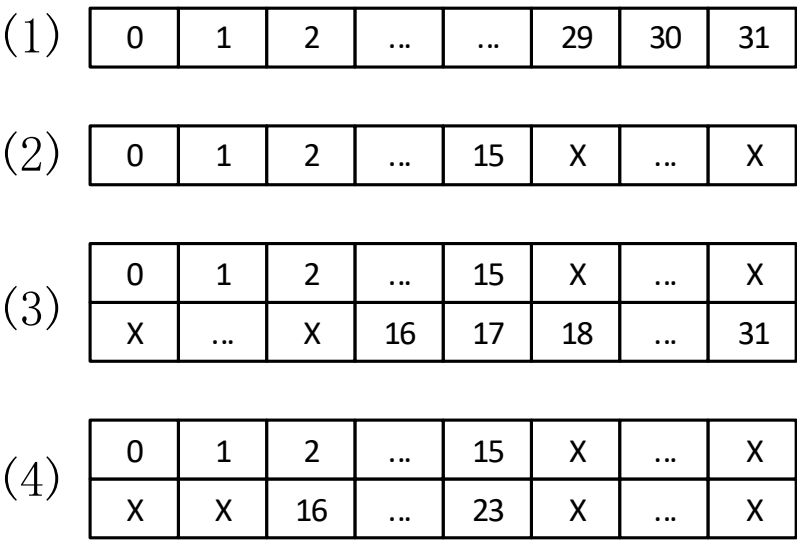


图 5: 实际所有的 VN 接入方式

3.2 设计的数据结构

```

1  typedef struct nr15_ldpc_simd_t
2  {
3      .....
4
5      /* Low-Latency Decoder Parameters */
6      int32_t units;           // floor(Z_c/REG_SIZE)
7      int32_t whole_degree;   // sum of degree of every hbg_row_d
8      int8_t* degree;         // number of connective check nodes
9                              // (length: hbg_row_d)
10     __m256i* cn_msg_avx2;    // avx2 message from cn to vn
11                              // (length: whole_degree*units)
12     __m256i* vn_msg_avx2;    // temp avx2 message from vn to cn (length: 19)
13     int8_t* llr_fixed;       // fixed llr info (length: 2*REG_SIZE+Nd)
14     int8_t* llr_addr_flag;   // flag for access type
15                              // (0: no mask; 1: 1 mask; 2: 2 mask)
16                              // (length: whole_degree*units)
17     int8_t** llr_addr;       // llr address (length: whole_degree*units)
18     int8_t** llr_addr_pre;   // extra llr address for flag=2
19                              // (length: whole_degree)
20     __m256i* mask;           // mask1 for flag=2 (length: whole_degree)
21     __m256i* mask_pre;       // mask2 for flag=2 (length: whole_degree)
22     __m256i endmask;         // mask for flag=1
23
24 } nr15_ldpc_simd_t;

```

3.3 使用掩码时遇到的问题

__m256i _mm256_maskload_epi32 (int const* mem_addr, __m256i mask)
vpmaskmovd

Synopsis
__m256i _mm256_maskload_epi32 (int const* mem_addr, __m256i mask)
#include <immintrin.h>
Instruction: vpmaskmovd ymm, ymm, m256
CPUID Flags: AVX2

Description
Load packed 32-bit integers from memory into dst using mask (elements are zeroed out when the highest bit is not set in the corresponding element).

Operation

```

FOR j := 0 to 7
  i := j*32
  IF mask[i+31]
    dst[i+31:i] := MEM[mem_addr+i+31:mem_addr+i]
  ELSE
    dst[i+31:i] := 0
  FI
ENDFOR
dst[MAX:256] := 0

```

Performance

Architecture	Latency	Throughput (CPI)
Skylake	11	1
Broadwell	8	2
Haswell	8	2

__m256i _mm256_maskload_epi64 (__int64 const* mem_addr, __m256i mask)
vpmaskmovq

__m256d _mm256_maskload_pd (double const * mem_addr, __m256i mask)
vmaskmovpd

__m256 _mm256_maskload_ps (float const * mem_addr, __m256i mask)
vmaskmovps

图 6: AVX2 mask load 相关函数

<code>__m128i _mm_mask_loadu_epi8 (__m128i src, __mmask16 k, void const* mem_addr)</code>	<code>vmovdqu8</code>
<code>__m128i _mm_maskz_loadu_epi8 (__mmask16 k, void const* mem_addr)</code>	<code>vmovdqu8</code>
<code>__m256i _mm256_mask_loadu_epi8 (__m256i src, __mmask32 k, void const* mem_addr)</code>	<code>vmovdqu8</code>
<code>__m256i _mm256_maskz_loadu_epi8 (__mmask32 k, void const* mem_addr)</code>	<code>vmovdqu8</code>
Synopsis <code>__m256i _mm256_maskz_loadu_epi8 (__mmask32 k, void const* mem_addr)</code> <code>#include <immintrin.h></code> Instruction: <code>vmovdqu8</code> CPUID Flags: <code>AVX512VL + AVX512BW</code>	
Description Load packed 8-bit integers from memory into <code>dst</code> using zeromask <code>k</code> (elements are zeroed out when the corresponding mask bit is not set). <code>mem_addr</code> does not need to be aligned on any particular boundary.	
Operation <pre> FOR j := 0 to 31 i := j*8 IF k[j] dst[i+7:i] := MEM[mem_addr+i+7:mem_addr+i] ELSE dst[i+7:i] := 0 FI ENDFOR dst[MAX:256] := 0 </pre>	
<code>__m512i _mm512_mask_loadu_epi8 (__m512i src, __mmask64 k, void const* mem_addr)</code>	<code>vmovdqu8</code>
<code>__m512i _mm512_maskz_loadu_epi8 (__mmask64 k, void const* mem_addr)</code>	<code>vmovdqu8</code>

图 7: AVX-512 mask load 相关函数

3.4 当前掩码处理方式

3.4.1 取数据加掩码方式

```

1  if (*pflag1 == 2)
2  {
3      vllr1 = VECTOR_LOAD(*p_indice_nod1);
4      vllr2 = VECTOR_LOAD(*p_indice_nod_pre1);
5      mask1 = VECTOR_LOAD(p_mask1);
6      mask2 = VECTOR_LOAD(p_maskpre1);
7      vllrm1 = VECTOR_AND(vllr1, mask1);
8      vllrm2 = VECTOR_AND(vllr2, mask2);
9      vllr = VECTOR_OR(vllrm1, vllrm2);
10
11     p_mask1++;
12     p_maskpre1++;
13     p_indice_nod_pre1++;
14 }
15 else
16     vllr = VECTOR_LOAD(*p_indice_nod1);

```

3.4.2 存数据加掩码方式

```

1  if (*pflag2 == 2)
2  {
3      vllr1 = VECTOR_LOAD(*p_indice_nod2);
4      vllr2 = VECTOR_LOAD(*p_indice_nod_pre2);
5      mask1 = VECTOR_LOAD(p_mask2);
6      mask2 = VECTOR_LOAD(p_maskpre2);
7      vllrm1 = VECTOR_AND(mask1, v2llr);
8      vllrm2 = VECTOR_AND(mask2, v2llr);

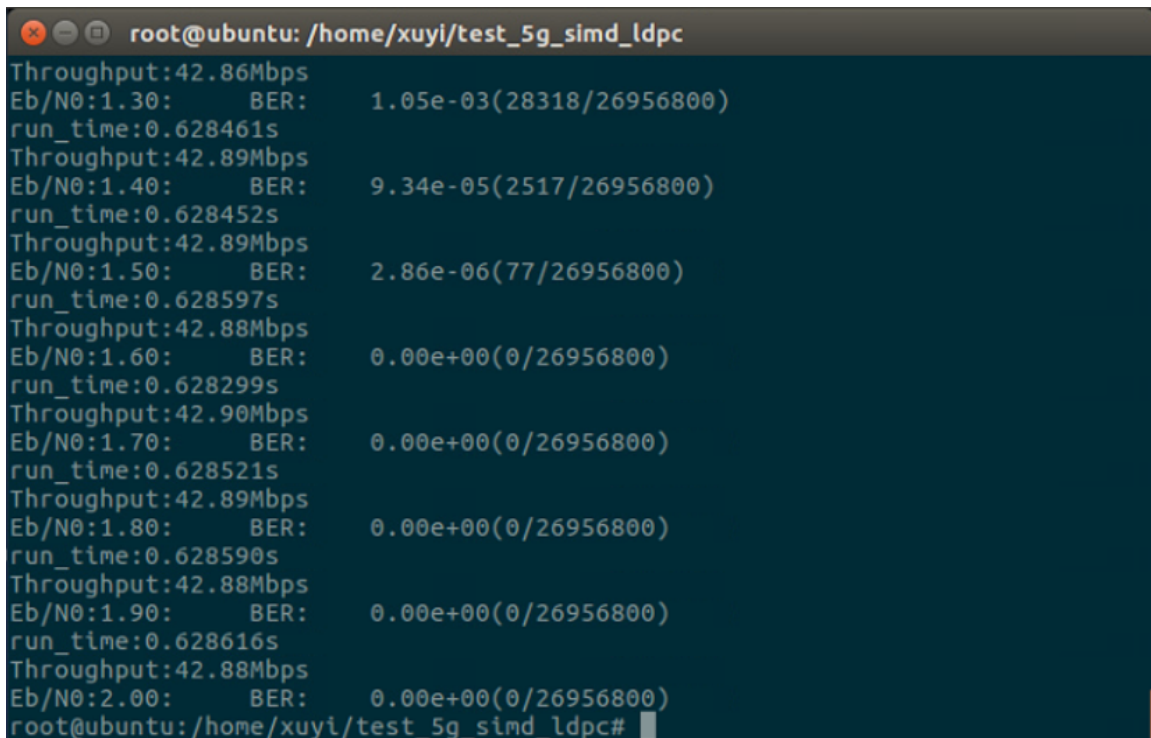
```

```

9      vllro1 = VECTOR_ANDNOT(mask1, vllr1);
10     vllro2 = VECTOR_ANDNOT(mask2, vllr2);
11     vllr1 = VECTOR_OR(vllrm1, vllro1);
12     vllr2 = VECTOR_OR(vllrm2, vllro2);
13     VECTOR_STORE(*p_indice_nod2, vllr1);
14     VECTOR_STORE(*p_indice_nod_pre2, vllr2);
15
16     p_mask2++;
17     p_maskpre2++;
18     p_indice_nod_pre2++;
19 }
20 else if (*pflag2 == 1)
21 {
22     vllrm1 = VECTOR_AND(endmask, v2llr);
23     vllr = VECTOR_LOAD(*p_indice_nod2);
24     vllro1 = VECTOR_ANDNOT(endmask, vllr);
25     vllr = VECTOR_OR(vllrm1, vllro1);
26     VECTOR_STORE(*p_indice_nod2, vllr);
27 }
28 else
29     VECTOR_STORE(*p_indice_nod2, v2llr);

```

3.5 系统测试



```

root@ubuntu: /home/xuyi/test_5g_simd_ldpc
Throughput:42.86Mbps
Eb/N0:1.30:    BER:    1.05e-03(28318/26956800)
run_time:0.628461s
Throughput:42.89Mbps
Eb/N0:1.40:    BER:    9.34e-05(2517/26956800)
run_time:0.628452s
Throughput:42.89Mbps
Eb/N0:1.50:    BER:    2.86e-06(77/26956800)
run_time:0.628597s
Throughput:42.88Mbps
Eb/N0:1.60:    BER:    0.00e+00(0/26956800)
run_time:0.628299s
Throughput:42.90Mbps
Eb/N0:1.70:    BER:    0.00e+00(0/26956800)
run_time:0.628521s
Throughput:42.89Mbps
Eb/N0:1.80:    BER:    0.00e+00(0/26956800)
run_time:0.628590s
Throughput:42.88Mbps
Eb/N0:1.90:    BER:    0.00e+00(0/26956800)
run_time:0.628616s
Throughput:42.88Mbps
Eb/N0:2.00:    BER:    0.00e+00(0/26956800)
root@ubuntu: /home/xuyi/test_5g_simd_ldpc#

```

图 8: 运行结果

表 1: High-Throughput 和 Low-Latency 系统性能对比 (Intel® Xeon® CPU E7-8867 V4 @2.40GHz)

scheduling	High-Throughput	Low-Latency
Throughput	59.19Mbps	43.02Mbps
Latency(K=8448)	4.5543ms	0.1958ms

4 改写仿真报告