DIAL YOUR NETWORKING CODE UP TO

Bruce Richardson, Harry van Haaren
What do we mean by “Vectorization”?

• SIMD – single instruction multiple data
  • Process data vector in parallel
• For Intel Architecture systems:
  • Intel® Streaming SIMD Extensions (Intel® SSE)
  • Intel® Advanced Vector Extensions (Intel® AVX ... Intel® AVX-512)
Why Vectorize?

• **Work with multiple packets in parallel**
  • Process the descriptor flags for 4 packets at a time
  • Lookup 8 hashes in a table simultaneously
• **Work with up to 64-bytes of a packet at a time**
  • One load instruction vs 8
  • Compare 3 protocol headers simultaneously
• **Take advantage of new instructions/capabilities**
  • Byte shuffling
  • Masking operations

Do more work with fewer instructions!
Talk Outline

Three examples of vectorization of packet processing workloads

1. OVS Packet Parsing – “Miniflow Extract”

2. DPDK Poll-Mode-Driver

3. OVS Wildcard Rule Classifier (DPCLS)
VECTORIZED PACKET PARSING

Case study with OVS Miniflow Extract
OVS Parses Packet Headers into a “Miniflow”

Bits
- “What does each block represent”
- 128 bits total
- Two \texttt{uint64\_t}

Blocks
- “Value of what bit says I am”
- A \texttt{uint64\_t} per block
- Block count == Number of bits set
Scalar miniflow_extract()

Parsing in OVS Today

1. Load and Store MAC, type
2. Load & Branch Ether Type
3. Load & Store IP src, dst, proto, ttl
4. Load & Branch on IP proto
5. Load & Store UDP sport, dport, len, ...
Vectorized miniflow_extract()

Optimized with SIMD

1. Load 64 bytes of headers
2. Apply packet header bitmask
3. Compare with known protocol
4. Shuffle Packet to Miniflow
5. Store Miniflow!
PACKET I/O

Vectorized NIC Descriptor Processing
RX Path – Shuffling Data

MBuf Fields (Extract only!!)

union {
  uint32_t packet_type;
  ...
}
uint32_t pkt_len
uint16_t data_len
uint16_t vlan_tci
union {
  uint32_t rss
  ...
}

Descriptor Fields (ixgbe)

uint16_t packet_type
uint16_t rsc_hdr_len
uint32_t rss_filter_id
uint32_t status_error
uint16_t pkt_len
uint16_t vlan_tci
RX Path – Shuffling Data

MBuf Fields (Extract only!!)

```c
union {
    uint32_t packet_type;
    ...
}
uint32_t pkt_len
uint16_t data_len
uint16_t vlan_tci
union {
    uint32_t rss
    ...
}
```

Descriptor Fields (ixgbe)

```c
uint16_t packet_type
uint16_t rss_hdr_len
uint32_t rss_filter_id
uint32_t status_error
uint16_t pkt_len
uint16_t vlan_tci
```

- Unused
- Used elsewhere
RX Path – Shuffling Data

MBuf Fields (Extract only!!)

```c
union {
    uint32_t packet_type;
    ...
}
uint32_t pkt_len
uint16_t data_len
uint16_t vlan_tci
union {
    uint32_t rss
    ...
}
```

Descriptor Fields (ixgbe)

```c
uint16_t packet_type
uint16_t rsc_hdr_len
uint32_t rss_filter_id
uint32_t status_error
uint16_t pkt_len
uint16_t vlan_tci
```

- Needs processing
- Unused
- Used elsewhere
RX Path – Shuffling Data

```
shuf_msk = _mm_set_epi8(
    7, 6, 5, 4, 15, 14, 13, 12, 0xFF, 0xFF, 13, 12, 0xFF, 0xFF, 0xFF, 0xFF
);
```
RX Path – Shuffling Data

```
shuf_msk = _mm_set_epi8(
    7, 6, 5, 4, 15, 14, 13, 12, 0xFF, 0xFF, 13, 12, 0xFF, 0xFF, 0xFF, 0xFF
);
```
RX Path – Shuffling Data

```
shuf_msk = __mm_set_epi8(
    7, 6, 5, 4, 15, 14, 13, 12, 0xFF, 0xFF,
    13, 12, 0xFF, 0xFF, 0xFF, 0xFF
);
```
shuf_msk = _mm_set_epi8(0xFF, 0xFF, 13, 12, 0xFF, 0xFF, 0xFF, 0xFF);
RX Path – Shuffling Data

shuf_msk = _mm_set_epi8(13, 12, 0xFF, 0xFF, 13, 12, 0xFF, 0xFF, 0xFF, 0xFF);
RX Path – Shuffling Data

```
shuf_msk = _mm_set_epi8(
    15, 14, 13, 12, 0xFF, 0xFF, 13, 12, 0xFF, 0xFF, 0xFF, 0xFF
);
```
RX Path – Shuffling Data

shuf_msk = mm_set_epi8(
    7, 6, 5, 4, 15, 14, 13, 12, 0xFF, 0xFF, 13, 12, 0xFF, 0xFF, 0xFF, 0xFF
);
RX Path – Shuffling Data

379  /* mask to shuffle from desc. to mbuf */
380       shuf_msk = _mm_set_epi8(
381           7, 6, 5, 4,  /* octet 4~7, 32bits rss */
382           15, 14,  /* octet 14~15, low 16 bits vlan_macip */
383           13, 12,  /* octet 12~13, 16 bits data_len */
384           0xFF, 0xFF,  /* skip high 16 bits pkt_len, zero out */
385           13, 12,  /* octet 12~13, low 16 bits pkt_len */
386           0xFF, 0xFF,  /* skip 32 bit pkt_type */
387           0xFF, 0xFF
388         );
RX Path – Shuffling Data

...  
452  descs[1] = _mm_loadu_si128((__m128i *)(rxdp + 1));
453  rte_compiler_barrier();
454  descs[0] = _mm_loadu_si128((__m128i *)(rxdp));
...
476  pkt_mb2 = _mm_shuffle_epi8(descs[1], shuf_msk);
477  pkt_mb1 = _mm_shuffle_epi8(descs[0], shuf_msk);
...
537  _mm_storeu_si128((void *)&rx_pkts[pos+1]->rx_descriptor_fields1,
538       pkt_mb2);
539  _mm_storeu_si128((void *)&rx_pkts[pos]->rx_descriptor_fields1,
540       pkt_mb1);
RX Path – Merging Data for Parallelism

Descriptor Struct

- uint16_t packet_type
- uint16_t rsc_hdr_len
- uint32_t rss_filter_id
- uint32_t status_error
- uint16_t pkt_len
- uint16_t vlan_tci

We want to be able to process the status of multiple packets in parallel
RX Path – Merging Data for Parallelism

desc3 = _mm_load_si128();  desc2 = _mm_load_si128();  desc1 = _mm_load_si128();  desc0 = _mm_load_si128();

sterr_tmp2 = _mm_unpackhi_epi32(descs[3], descs[2]);  sterr_tmp1 = _mm_unpackhi_epi32(descs[1], descs[0]);

staterr = _mm_unpacklo_epi32(sterr_tmp1, sterr_tmp2)
RX Path – Using the Merged Data

Counting Packets using “DD” Bit in “staterr”

533  staterr = _mm_and_si128(staterr, dd_check);  /* mask unwanted bits */
534  staterr = _mm_packs_epi32(staterr, zero);    /* pack 32-bit to 16 bit values */
      /* (we go from 128-bits to 64 bits data) */
...
545  var = __builtin_popcountll(_mm_cvtsi128_si64(staterr));  /* count the bits set */
546  nb_pkts_recd += var;                                     /* update our stats */
547  if (likely(var != RTE_IXGBE_DESCS_PER_LOOP))    /* not enough packets – we’re done */
548       break;
RULE MATCHING

Vectorizing the OVS Wildcard Engine
Scalar Wildcard Matching

Scalar Lookup

1. Loop to find Table IP block
2. Loop find Table UDP block
3. Mask Packet By Table
4. Hash resulting data
Compute-based Scalar Wildcard Matching

**Compute – Don't Loop**

1) Generate bitmasks
2) Use Popcount (Bitmask to Index)
3) Hash resulting data
4) Loop for each block

**Packet Miniflow**

| 0 | - | - | - | - | - | - |

**Diagram:**

- Bits[0]
- Gen. Mask
- &
- Bitmask
- popcount
- Index[]
- Load Index
- Data

**Flow:**

- HASH DATA
- Hash Value
AVX-512 SIMD Wildcard Matching

AVX-512 Compute – No Loops!

1) Generate bitmasks

2) Use Vector Popcount

3) SIMD Loop Unrolling
   - 1 Loop iteration/SIMD “lane”
   - k-masks to disable lanes

4) Hash resulting data
AVX-512 SIMD Wildcard Matching

AVX-512 K-masks!

- “Switch off” lanes
- Zero data in off lanes
- Blend data off lanes

Compared to SSE / AVX

- No more explicit blends!
- Easy to manage per-lane ops
Benefits of Vectorization

• Larger Loads & Stores
  • Fewer instructions to get data in and out of the core

• Increased Compute per Instruction
  • Work on more bigger blocks of data
  • Work on multiple blocks of data

• Novel Instructions
  • Shuffle data
  • Mask operations

Fewer instructions for same amount of work
TX Path – bigger stores

- Tx function's job - Take an array of mbufs and turn them into these:

8.4.2.1.1  Transmit Data Descriptor

<table>
<thead>
<tr>
<th>Quad Word</th>
<th>6</th>
<th>3</th>
<th>0</th>
<th>0</th>
<th>L2 Tag 1</th>
<th>Tx Buffer Size</th>
<th>Offset</th>
<th>rsv</th>
<th>CMD</th>
<th>DTYP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>33</td>
<td></td>
<td>43</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td>65</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>33</td>
<td></td>
<td></td>
<td>43</td>
</tr>
</tbody>
</table>
static inline void vtx1(volatile struct i40e_tx_desc *txdp,
struct rte_mbuf *pkt, uint64_t flags)
{
    uint64_t high_qw = (I40E_TX_DESC_DTYPE_DATA |
    ((uint64_t)flags << I40E_TXD_QW1_CMD_SHIFT) |
    ((uint64_t)pkt->data_len << I40E_TXD_QW1_TX_BUF_SZ_SHIFT));

    __m128i descriptor = _mm_set_epi64x(high_qw, pkt->buf_iova + pkt->data_off);

    _mm_store_si128((__m128i *)txdp, descriptor);
}

3 Pieces of Data Per Packet
One store per packet
for (; nb_pkts > 3; txdp += 4, pkt += 4, nb_pkts -= 4) {
    uint64_t hi_qw3 = hi_qw_tmpl | ((uint64_t)pkt[3] - data_len << I40E_TXD_QW1_TX_BUF_SZ_SHIFT);
    uint64_t hi_qw2 = hi_qw_tmpl | ((uint64_t)pkt[2] - data_len << I40E_TXD_QW1_TX_BUF_SZ_SHIFT);
    uint64_t hi_qw1 = hi_qw_tmpl | ((uint64_t)pkt[1] - data_len << I40E_TXD_QW1_TX_BUF_SZ_SHIFT);
    uint64_t hi_qw0 = hi_qw_tmpl | ((uint64_t)pkt[0] - data_len << I40E_TXD_QW1_TX_BUF_SZ_SHIFT);

    __m256i desc2_3 = _mm256_set_epi64x(
    __m256i desc0_1 = _mm256_set_epi64x(
        hi_qw1, pkt[1] - buf_physaddr + pkt[1] - data_off,
        hi_qw0, pkt[0] - buf_physaddr + pkt[0] - data_off);

    _mm256_store_si256((void *) (txdp + 2), desc2_3);
    _mm256_store_si256((void *) txdp, desc0_1);
}

Loop unroll to do 4 at a time

One store for every 2 packets
TX Path – Bigger Stores – AVX512

With AVX-512 can go further and have 4 descriptors per write

However, increased number of instructions also allows more to be done using vector rather than scalar code:

- Use load and expand to set up addresses
- Use gather rather than scalar inserts for the lengths
- Use blend to merge in arrays of constants rather than individual values
- Use kmasks to work on some parts of the data only