清華大學

Tsinghua University

HUAWEI

Ultra-Fast Bloom Filters using SIMD Techniques

Jianyuan Lu*, Ying Wan*, Yang Li*, Chuwen Zhang*,
Huichen Dai*, Yi Wang †, Gong Zhang †, Bin Liu*

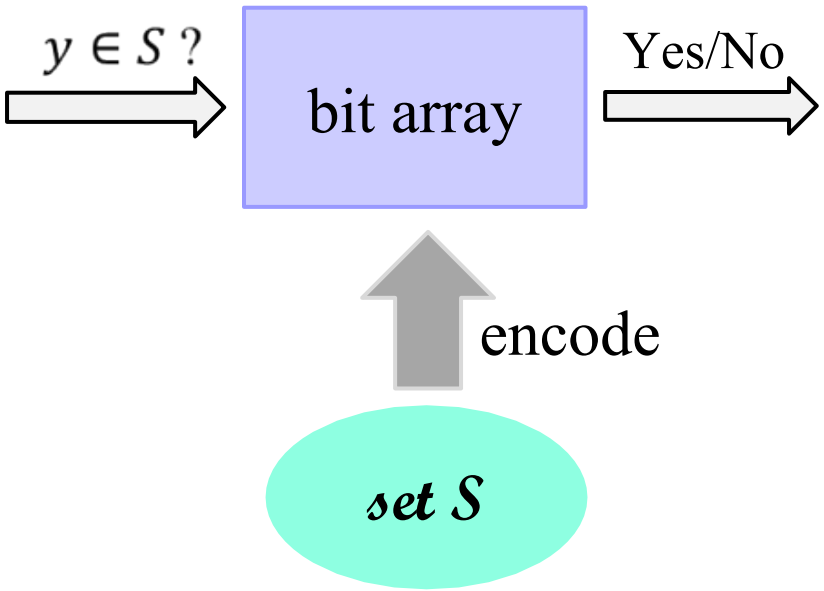
* Tsinghua University, China
†Huawei Future Network Theory Lab, Hong Kong

Outline

1. **Background and Motivation**
2. **Our Solutions**
3. **Simulation Results**
4. **Conclusions**

What is Bloom Filter?

- A Bloom filter encodes a large set $S = \{x_1, x_2, \ldots x_n\}$ to a small bit array.
  - space-efficient randomized data structure
- A Bloom filter is used to check whether an element y belongs to the set S or not.



Bloom Filter's Applications

- Due to simplicity and efficiency, Bloom filters (and their variants) have been applied in a wide range of network applications.
  - Routing table lookup
  - Packet classification
  - Per-flow measurement
  - Deep packet inspection
  - etc.

The Challenges

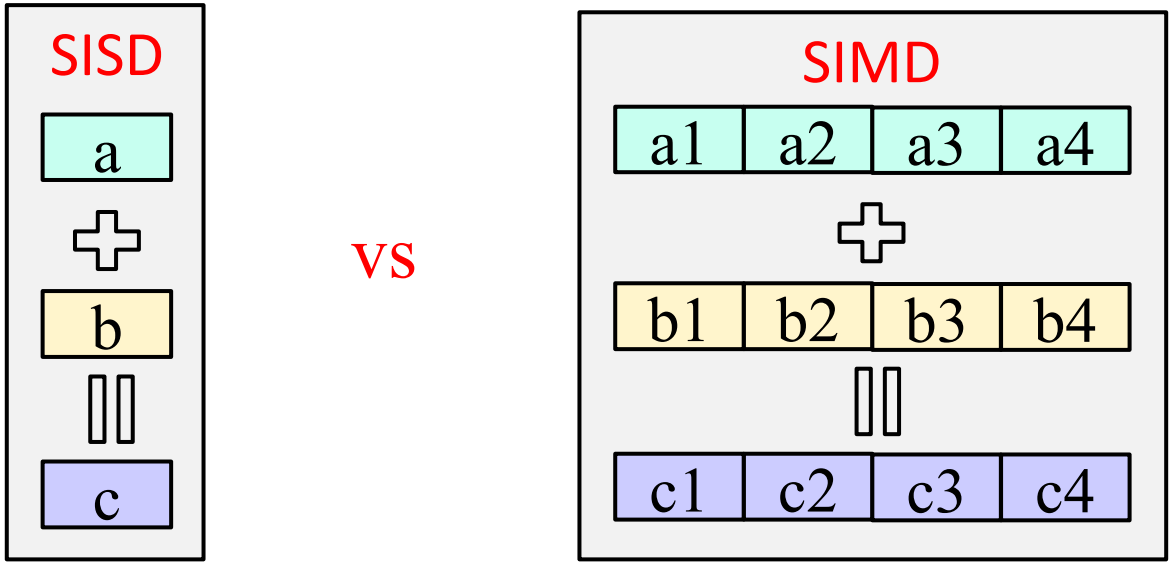
- Fast Network Link Speed
  - The 40GE and 100GE ports for routers' line-cards have already been commercialized and deployed
  - Cisco CRS-X and Huawei NE9000 both support 400 Gbps linecards
- Leaving a very limited processing delay
  - 10GE port needs to achieve 15Mpps throughput
  - Processing a packet within 300 clock cycles with the state-of-the-art 4GHz CPU
- The Bloom filters need to keep pace

The Challenges

- Bloom filters become the performance bottleneck
- For example, one membership query needs at least 552 clock cycles using a common configurations below.
  - k=10 hash functions
  - Key length for hash function input is 8 bytes
  - All hash functions employ CRC32
- Recall that every 300 clock cycles, a 10GE port has to process a packet.

Idea

- Speedup the Bloom filter query using SIMD techniques
  - SIMD instructions can operate multiple operands in parallel, compared to traditional SISD instructions.
  - SIMD instructions are widely supported by general CPUs and embedded CPUs. Take Intel CPU as an example. It supports MMX, SSE, AVX, FMA, KNC, SVML etc, SIMD instruction sets

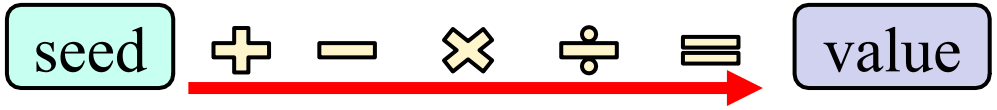


Solutions

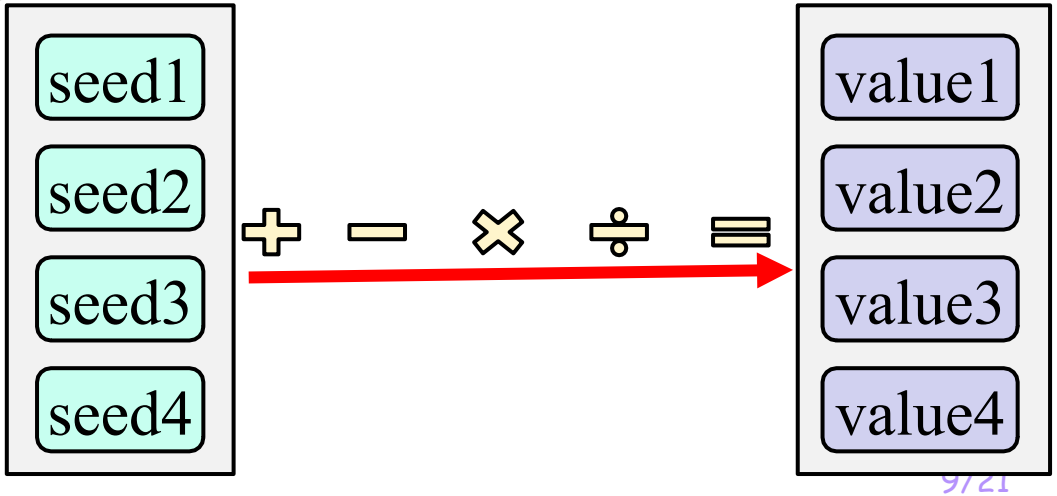
- We propose Ultra-Fast Bloom Filters (UFBF), a parallel Bloom filter framework accelerated by SIMD.
- The UFBF has three optimizations
  - Parallel hash computation
  - Parallel bit-test process
  - Improving cache efficiency

Parallel Hash Computation

- Traditional hash computation
  - 1 initial seed encounters a sequence of arithmetic and logic operations, producing 1 hash value.
  - Using SISD instructions



- Parallel hash computation
  - k initial seed encounters a same sequence of arithmetic and logic operations, producing k hash values.
  - Using SIMD instructions



Parallel Hash Computation

Initialization

Load k seeds to an SIMD register

Implementing the rewrite parallel hash algorithm

Store the k hash values to memory

Algorithm 1: The hash computation algorithm in UFBF

```
1 seeds[p] ← [seed_1, seed_2, ..., seed_p]
2 hashVals[p] ← [0, 0, ..., 0]
3 vr_seeds ← v_load(seeds)
  /* load the p seeds to an SIMD register   */
4 vr_val ← v_hashFunc(vr_seeds)
  /* implement the SIMD hash function which
     takes p seeds and compute in parallel  */
5 hashVals ← v_store(vr_val)
  /* store the p hash values to memory      */
```

Rewrite rules：

1.broadcast the data， v_broadcast

2. Run the corresponding SIMD instruction， v_OP

Algorithm 2: The main change from a traditional hash function to its SIMD-version

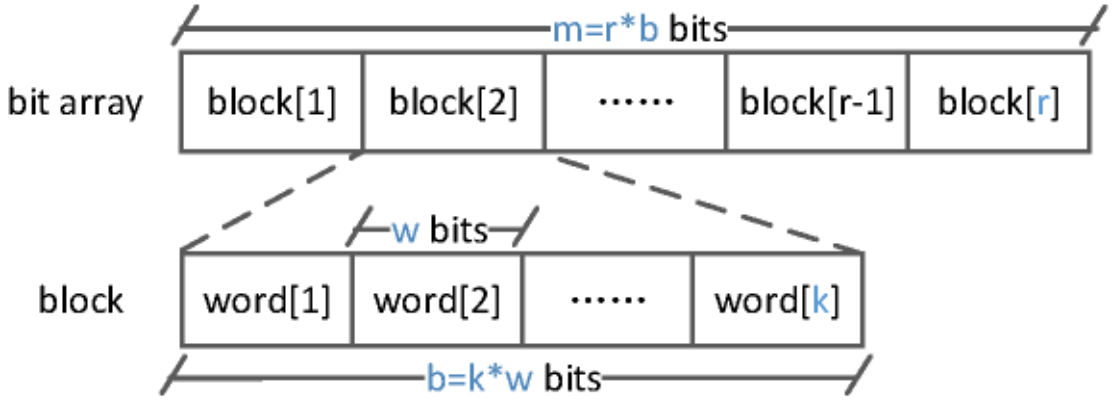
```
1 val ← val OP a
  /* OP is a general arithmetic operation, val
     stores the intermediate hash value     */
        ⇓
1 vr_a ← v_broadcast(a)
  /* v_broadcast copy p copies of a to vr_a */
2 vr_val ← v_OP(vr_val, vr_a)
  /* v_OP is the SIMD-version of OP, vr_val
     stores the intermediate p hash values  */
```

Parallel Bit-test Process

- Traditional Bloom filters employ sequential bit-test
  - $\mathrm{B}[h_1(y)]$, $\mathrm{B}[h_2(y)]$, …, $\mathrm{B}[h_k(y)]$

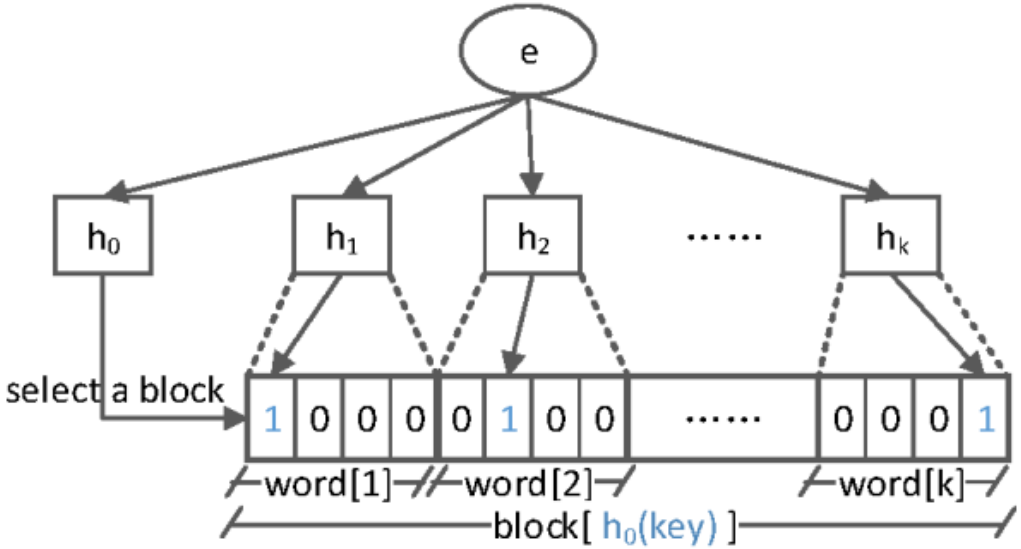
- How to achieve parallel bit-test?
  - $\mathrm{B}[h_1(y)]$
  - $\mathrm{B}[h_2(y)]$
  - …
  - $\mathrm{B}[h_k(y)]$

Parallel Bit-test Process



- Bit array is composed of r blocks
  - The blocks have the same size, smaller than or equal to the cache-line size.
- Each block is composed of k words
  - word means the bit width of traditional register, e.g., 32-bit or 64-bit
  - The k neighboring words bring in parallelism for membership

Parallel Bit-test Process



- How to insert an element e ?
  - First, use one hash function selects a block from bit array
  - Then, use k hash functions select one bit in each word and set
  - That is to say, element e is encoded into k words in one randomly selected block.

Parallel Bit-test Process

Use the parallel hash computation algorithm to compute the hash values

Use SIMD instructions to parallel shift bits

Use SIMD instructions to parallel test bits

Return positive or negative according to the test result

Algorithm 3: The membership check algorithm in UFBF

```
1  Function membershipCheck(element e)
2      loc ← compute the block index of e
3      vr_val ← compute k hash values using Algorithm 1
4      vr_a ← v_broadcast(1)
5      vr_a ← v_shiftLeft(vr_a, vr_val)
       /* v_shiftLeft shifts k words in vr_a
          left in parallel by the amount
          specified by vr_val                    */
6      vr_b ← v_load(&block[loc])
7      vr_b ← v_not(vr_b)
       /* v_not is bitwise NOT operation         */
8      v_test(vr_a, vr_b)
       /* v_test is bitwise AND operation        */
9      if zero-flag is set then
10         return positive
11     end
12     return negative
13 end
```

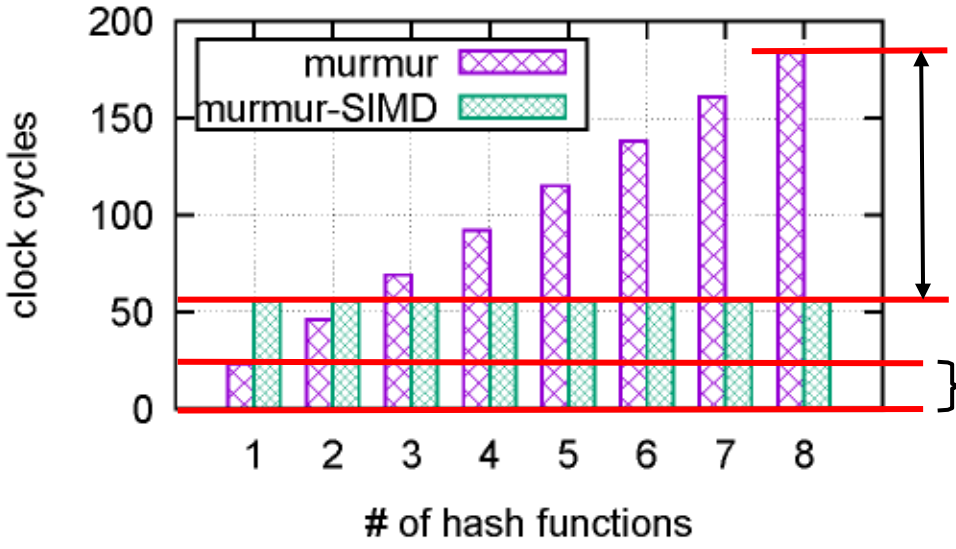
Improving Cache Efficiency

- Theorem 1
  - Suppose the cache-line size is L. If the block size satisfies $b|L$ and the bit array is L-aligned, at most one cache miss would occur in one membership check.
- Corollary 1.
  - Suppose the cache-line size L is a power of 2. Then we can conclude that if $b|L$, b is a power of 2, and the bit array is L-aligned, at most one cache miss would occur in one membership check.
- Corollary 2
  - Suppose w is a power of 2. Then we can conclude that if $\mathrm{k} = \frac{b}{w} \leq \frac{L}{w}$ is a power of 2, and the bit array is L-aligned, at most one cache miss would occur in one membership check.

Simulation Results

- Experiment Setup
  - Intel i7-4790 CPU, L1 cache (L1 D-Cache is 32 KBytes, L1 I-Cache is 32 KBytes), L2 cache (256 KBytes), L3 Cache (8 MBytes)
  - AVX, AVX2 SIMD instruction sets
  - Real-world Internet traces, obtained from CAIDA

Simulation Results

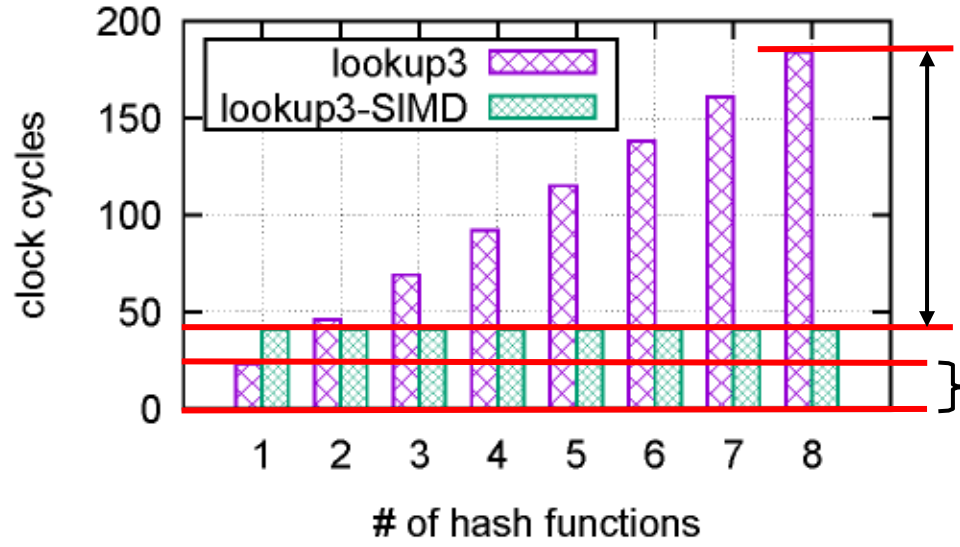


Saving more hash computation time when hash function number increases

2.43 times

MurmurHash

23 clock cycles



Saving more hash computation time when hash function number increases

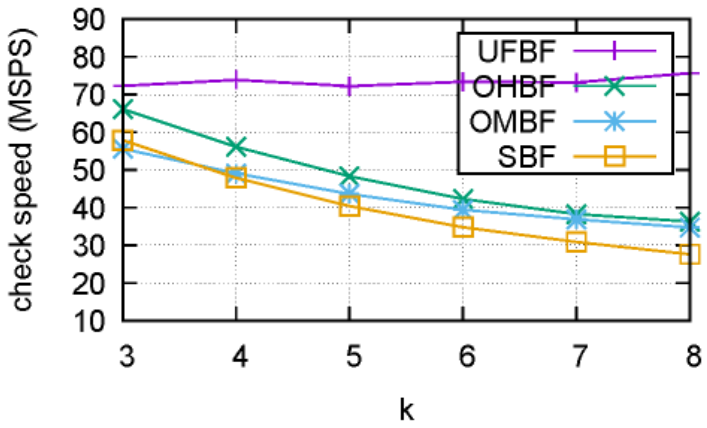
1.78 times

Lookup3

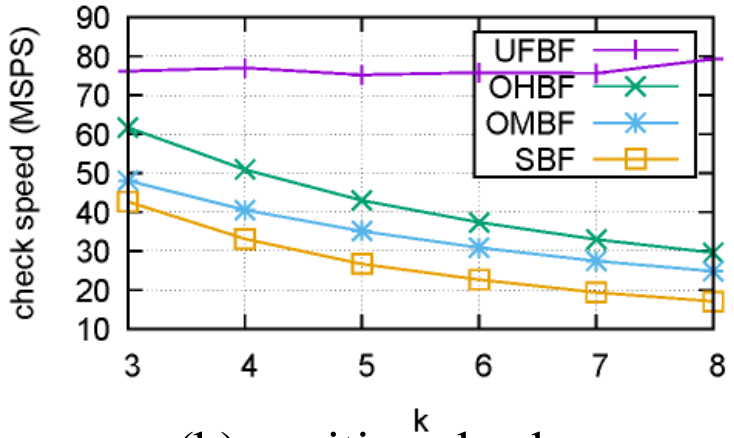
23 clock cycles

Simulation Results

- When the hash function number k varies, the membership check speed comparison of four Bloom filter variants.
  - UFBF is fastest over other Bloom filter variants.
  - In negative check and positive check, the membership check speed of UFBF is the same.
  - The small jitter of UFBF results from different cache efficiency when k varies, better cache efficiency when k is a power of 2.



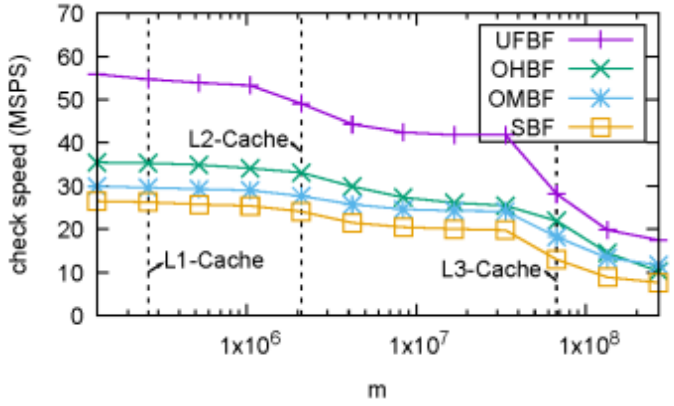
(a) negative check



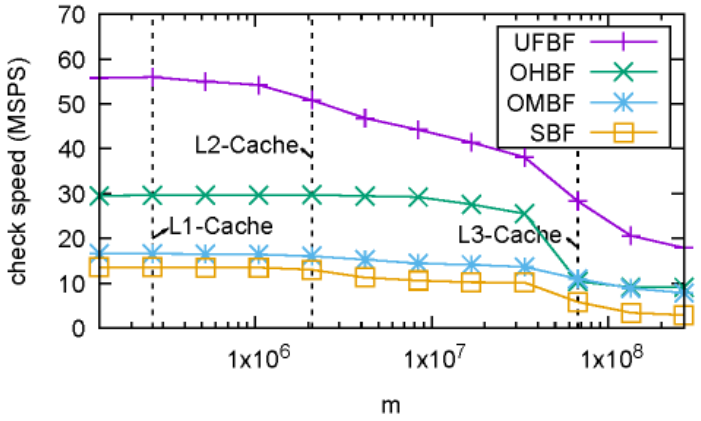
(b) positive check

Simulation Results

- When Bloom filter size m varies, the membership check speed comparison of four Bloom filter variants.
  - Three level CPU caches(L1, L2, L3) are marked out.
  - We conclude that， whether the Bloom filter size m is larger than , or smaller than, the CPU cache size, the membership check speed of UFBF is the fastest.



(a) negative check



(b) positive check

Conclusions

- We propose Ultra-Fast Bloom Filter(UFBF) which employs SIMD parallel techniques to accelerate membership check
- The UFBF has three optimizations over standard Bloom filter
  - Parallel hash computation
  - Parallel bit-test process
  - Improving cache efficiency
- Simulation studies show that, due to parallel computing and higher cache efficiency, UFBF improves the membership check speed 2~3 times over state-of-the-art Bloom filter variants

Q&A

Thank You!

Backup problem 1

- What's the change of false positive rate of the UFBF
- Answer
  - The UFBF has a little higher false positive probability than standard Bloom filter, it has been formally analyzed in the paper. Actually, the UFBF sacrifices the false positive rate a little for vast lookup performance improvement.

Backup problem 2

- How do you realize the parallel algorithms using SIMD instructions? Using compile languages or C programming language?
- Answer
  - The intel has a guide to call their SIMD instructions using C/C++ programming language. In our algorithm, most of the SIMD instructions are called by C programming language, however, there are a few sentences are called by compile language, because intel dose not provide the corresponding interface for C programming language.
  - We can share the code of our algorithm after the IWQoS conference if someone has interests.