Leveraging SIMD parallelism for accelerating network applications

Hejing Li
KAIST
hajeongee@gmail.com

Juhyeng Han
KAIST
sparkly9399@gmail.com

Dongsu Han
KAIST
dongsuh@ee.kaist.ac.kr

ABSTRACT

Software packet processing frameworks act as critical components in modern network architecture, as their performance has a vital impact on the quality of the network services. Motivated by the increasing number and capability for advanced vector instructions in recent mainstream CPUs, this paper explores a new parallel processing design and implementation of data structures and algorithms that are frequently used for building network applications. In particular, we propose effective SIMD optimization techniques for the bloom filter and Open vSwitch megaflow cache. Our design reduces memory access latency via careful prefetching and a new design that meets the needs of fast data consuming instructions. Our evaluation shows performance improvements up to 162% in bloom filter and 48% in Open vSwitch compared to their scalar version.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Computing methodologies** → **Parallel computing methodologies**.

ACM Reference Format:
Hejing Li, Juhyeng Han, and Dongsu Han. 2020. Leveraging SIMD parallelism for accelerating network applications. In *4th Asia-Pacific Workshop on Networking (APNet '20), August 3–4, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3411029.3411033

1 INTRODUCTION

Hash tables are fast data structures for manipulating tables with a large number of entries. Due to simplicity and efficiency, hash tables and their variants act as fundamental components of several network applications, such as tuple space search algorithm in virtual switches [21, 25] and hash-based pattern matching in IDS applications [8, 24].

In a high-speed network environment equipped with 10/40/100 GbE network cards, such software packet classification applications face over 14 millions of minimum-sized packets per second. Excessive hash function calls during the packet classification process gives heavy computational load to the CPU and has significant impact on the application's performance.

Meanwhile, architectural advancements have been made into the recent mainstream CPUs. Not only with respect to the number of cores and clock frequency, but they also adopt higher single-instruction-multiple-data (SIMD) capabilities including wider SIMD registers and more advanced vector instructions [12, 13]. SIMD technology has been evolving from the early years and is showing no signs of stopping. In Intel CPUs, for example, the first generation SIMD technology (MMX) only uses 64-bit registers and provides only integer operations. Since then SIMD instructions have been extended to SSE (128-bit), AVX/AVX2 (256-bit) and most recently, AVX-512 which supports 512-bit register operation. In addition to the extensions in register size, more practical features such as floating-point number operations, non-contiguous memory load (gather), and store (scatter) operations are added to the latest versions of the ISA [12, 13].

Effectively utilizing SIMD capabilities can drastically improve the performance of the tasks running on CPUs. Various vectorized designs are proposed to accelerate database operations [22, 23, 28, 29], multimedia processing [6] as well as other applications that have data parallelism [18, 20]. However, converting a scalar code having its control flow into a vectorized code is not trivial for the data packed into a vector are executed with the same instruction no matter which branches are taken for the data. Thus conditional branches should be substituted by the data flow, which means a series of arithmetic operations with specific operand for each data in different branch. Moreover, parallel operations often come at the expense of increased memory access latency. To be specific, since a data object resides in a contiguous memory block, a scalar code benefits from the spatial locality. However, loading data for parallel processing often involves several discontiguous memory access. One cache miss among the access would postpone the data being loaded to the vector register and thus degrade performance.

We argue that SIMD vectorization is more than a parallel algorithm, but proper memory access latency hiding and corresponding data structure conversion. The array of structure form of data in scalar code leads to discontiguous memory access for the vectorized code to load several data on the vector register. Instead, form of the structure of array that locates the same field data in each array is preferred for the reason of that the instruction to load a contiguous memory block is much cheaper than from discontiguous memory addresses [13].

APNet '20, August 3–4, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-8876-4/20/08…$15.00
https://doi.org/10.1145/3411029.3411033

In this paper we show two use cases of SIMD acceleration for vectorized bloom filter [23] and Open vSwitch megaflow cache [21]. The algorithm of the existing vectorized bloom filter [23] is designed for 256-bit ISA, which can process 8 32-bit keys. To assign an operand for each lane by the boolean result of the conditional branch, it refers to a table having 2^8 entries and each entry contains a set of operands for the vector. The table size grows exponentially to 2^{16} entries for the naïve extension to the 512-bit ISA, which prevents its performance to scale with the size of the registers used for the algorithm. To extend the algorithm to the 512-bit ISA without penalty, we exploit the new instructions in the state of the art SIMD technology, AVX-512, and adopted careful data prefetching and loop unrolling optimization techniques to hide the memory access latency. Moreover, we propose a new data structure for the packet batch representation in Open vSwitch megaflow cache to avoid discontiguous memory accesses.

Our evaluation shows bloom filter lookup throughput for the synthesized inputs outperforms existing vectorized bloom filter for up to 162%. The end-to-end throughput of OvS with the microflow cache disabled, vectorized Open vSwitch outperforms its scalar implementation by up to 48%.

In addition, we show that SIMD aware data structure is mandatory for Open vSwitch acceleration by comparing its performance on the vectorized implementation without a new data structure.

2 OPPORTUNITY AND CHALLENGES

To improve the performance of network applications, many previous studies explore the design space of parallelism relying on the massively-parallel processing power of GPU [10, 14, 16, 26]. Some research reports that other hardware accelerators such as integrated GPU (APU) also help improve the performance [9]. Although these approaches are effective, the involvement of hardware accelerators incurs extra expense for the devices and energy consumption.

In contrast, SIMD technology is widely adopted on mainstream CPUs. Recent Intel CPUs support 512-bit instruction set which can pack sixteen 32-bit integers within a vector. Although SIMD parallelization degree is less aggressive than GPUs, it has benefited from other aspects. First, it is cost-effective because it does not incur the cost of buying and maintaining new hardware. Second, data is processed on CPU, thus, it has no additional latency caused by transferring data to/from the accelerators. Previous work reports it takes $\sim 15\mu$s to send one-byte data to and from a GPU [19], which is intolerable for some latency-critical applications [15]. Without the data transferring latency overhead, SIMD technology in CPU is more attractive to carry out parallelization.

In leveraging SIMD, two considerations must be made to adopt the original program to use vector instructions. We present the two and demonstrate their use through two case studies in this paper.

Control flow to data flow conversion: SIMD vectorization requires an algorithmic change in the original program, which is different from vectorization in GPU. In GPU programming, a code named *kernel* is executed on a streaming multiprocessor. It follows SIMT (single instruction multiple threads) computation model in which the cores in a streaming multiprocessor shares the same program counter and executes the same instruction. The control divergence is managed by the hardware by turning the threads on and off. Thus, a *kernel* code can be written in the same way as scalar code executed in CPU.

On the contrary, a SIMD program should handle the control divergence of each data in a vector in the program. The conditional branches turn out to be a comparison operation that generates the bitmask value as the result which indicates the lanes that taking the branch or not by the bit set. Then different branch operations are done selectively to the lanes corresponding to the bitmask. Other lanes of data those are in the same vector but do not require the operation cause waste of vector computation. Converting the control flow to a data flow while maintaining high utilization of the vector is not straightforward.

Memory access latency hiding: A 512-bit SIMD instruction can consume sixteen 32-bit data at once. Feeding multiple data quickly enough so as to prevent the instruction from stalling is crucial for high performance. A scalar code accesses one data structure at once and once the data structure is loaded on the cache, it can take advantage of the spatial locality. However, for loading the data onto a vector register, several addresses spread over the array of structures are accessed. The instruction cannot retire until all the data is fetched from the memory. The possibility of a vector operation to benefit from the spatial locality is far less than the scalar code. Because of that, naïvely change the branches in the algorithm to data flow with the existing array of structure form of data might slow down the program. Thus, memory access latency hiding [15] becomes much more important and must be considered along with the vectorization algorithm to effectively utilize SIMD instructions.

3 DESIGN AND IMPLEMENTATION

3.1 Bloom Filter

Bloom filter is a probabilistic data structure designed to test whether an element is a member of a set. Multiple hash functions are used to index each element. To insert an element to the set, it sets each index bits to one in the bloom filter, which is a fixed-length bit array. To query an element for the membership, compute the hash values of the key, and test all the bits in the bloom filter. If all the bits are set, then the element is highly likely to belong to the set. Else, if there exist a bit tested is zero, the element is definitely not in the set. The simplicity and space-efficient characteristics make bloom filter and its variants widely used in network applications [3, 5] and database systems [8].

We start by giving a brief introduction to the scalar bloom filter algorithm and then describe our implementation details step by step. The hash function used in the bloom filter is MurmurHash3 which is well known as a fast and uniform algorithm. We input the key with the different seed values to get several independent hash values with one single hash algorithm. The keys, which in this case 32-bit integer values are

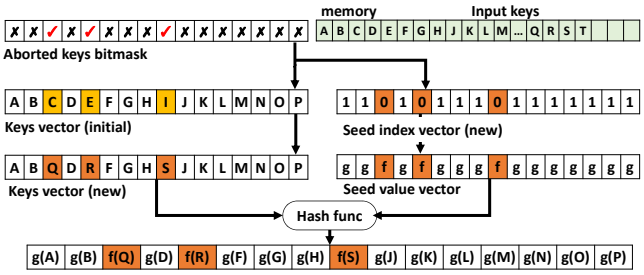

Figure 1: AVX-512 vectorized bloom filter overview

prepared as an input array. In the scalar implementation of the bloom filter, one key is processed at a time. The corresponding bits at the offsets dependent on the hash values are tested in order. Once a bit is not set, the key is discarded and the next key is processed. For the lookup succeeded keys, the keys and their payloads are stored to the output array.

Vectorization Algorithm: To re-write the bloom filter into a vectorized program, the control flow should be converted to the data flow. The data flow in our program referred to previous vectorized bloom filter [23] can be divided into three parts, which are hashing part, gather and test part, vector update part. We describe each component below.

Sixteen keys are loaded on the 512-bit vector at the starting of the iteration. Each key starts to compute the first hash value with the hash seed index for each key points to the first seed value. The hash seed values are loaded according to the index values and input to the vectorized MurmurHash3 function together with the keys vector. The vectorized hash function outputs sixteen 32-bit hash values in a 512 vector as the result.

The 32-bit filter words containing the bits to be tested are gathered from the memory. The offset of the word is derived from the hash value divided by 32. The bit is tested for each key by AND operation with a mask vector that only the tested bit is set in each lane. The test result will return a 16-bit bitmask that indicates the test failed keys' lane. Then at the last stage, the hash seed index vector will be updated to point to the next seed value of each lane. And for those keys that already reached the last seed value, means all the bits for the key are tested and no anyone bit is zero. Those keys will be regarded as to be pass and be stored in the output array.

In the next iteration, the keys that failed or passed all the hash functions will be removed from the keys vector, and new keys will be loaded on those lanes. In the prior work [23] it permutes the keys vector to separate the keys should remain and those should be removed. A set of permutation indices is indicating each lane's new position after the permutation. Those sets of indices are stored in a table and accessed using the 8-bit test result as the offset to the set. We utilize the latest AVX-512 instruction to remove the permuting process. The key instruction is *expandload* as shown in Figure 2 which loads contiguous values in memory into specific lanes in the vector that indicated by the bitmask.

Figure 1 shows a possible situation in the second iteration. The aborted keys bitmask tells the positions of keys that failed

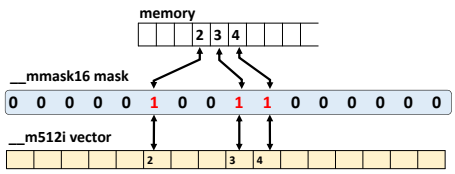

Figure 2: Expand load and compress store intrinsic fucntion operations

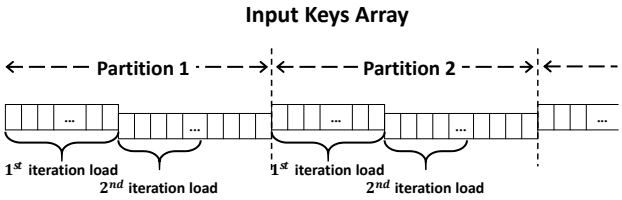

Figure 3: Multi-way loop unrolling

or succeed in all the tests in the previous iteration. Those lanes will be overwritten by the new keys in the input array. The loop iterates until remained keys are less than aborted keys in previous iteration. And the rest keys are processed serially.

Vector Loop Unrolling: Modern processors feature advanced out-of-order execution to avoid pipeline stalls. Loop unrolling increases the number of independent instructions in the loop body and it leads to increasing opportunities to benefit from this feature and improving the performance. The previous work [23] implemented loop unrolling with a maximum factor of 2, which processes two vectors of keys in the loop body.

We extend the prior work [23] by introducing multi-way loop unrolling implementation. It divides the input keys array into the number equal to the unrolling factor U. In the loop body, the number of U vectors load input keys from the start of each partition as shown in Figure 3. In the next iteration, each vector loads new keys to be processed in its own partition. The for-loop ends when one of the vectors among those reaches the end of its partition, and the rest of the keys in the input array will be handled serially. In this way, the unrolling factor can be adjusted to any number to get the best performance.

Bloom Filter Prefetch: Because filter word gathering operation is random access to the memory, it causes frequent cache misses and significant performance degradation when the bloom filter size exceeds the size of the caches. Several processing vectors in the loop body gives further chances for the acceleration. That is to adopt memory access latency hiding technique, prefetch the filter words in the first hash value vector and then proceed to compute the hash value of the second keys vector and prefetch the filter words according to the second hash value vector and so on. When all the vectors in the loop body finish the hash computing and prefetching, then the loop body goes on for the remaining procedure of gathering and testing. Several processing vectors existing in the loop body enabled hash computing and memory accessing interleaving such that improves the lookup performance with a large size of bloom filters.

Functions	CPU%
netdev_flow_key_hash_in_mask	48.67%
cmap_find_batch	18.48%
dpcls_rule_matches_key	6.54%
total	73.69%

Table 1: OVS profiling data shows megaflow cache lookup is a bottleneck. Tested with the number of average traversed subtables set to 30 per packet, each subtable contains 2000 rules

3.2 Open vSwitch

The performance of the OvS packet classification is critical to the quality of network services. Megaflow cache as a flow cache module in the packet classification pipeline should handle most of the packets when it faces a large number of flows. The profiling data for stress testing megaflow cache in Table 1 depicts that the major bottleneck in megaflow cache lookup is computing the hash values for the packets. Our goal is to vectorize this with SIMD instructions.

Existing Data Structure and Algorithm: OvS megaflow cache adopts tuple space search algorithm to perform packet classification. A tuple or subtable means a collection of classification rules that matches the same combination of fields with the same wildcard. For each subtable, the wildcard mask is applied to all the packets that remaining in the batch. The only fields that the subtable cares are ANDed with the mask value and composed into the hash key for computing the signature.

The data structure used in this algorithm is *miniflow*, which is a compact data structure to represent the packet header information. OvS defines a huge *flow* data structure in which enumerates all the header fields used in all kinds of protocols that OvS supports. Because of its wide coverage, the size of it is fairly large and sparse to represent a packet header with it. *miniflow* is proposed to minimize the packet batch memory footprint and the number of accessed cache lines. It has a bitmap of which each bit represents each *uint_64* value in *struct flow*. A zero bit indicates the corresponding 64-bit value in *struct flow* is zero and a 1-bit that may be nonzero value. The nonzero 64-bit values follow the bitmap.

In addition to the packets, a subtable mask is also represented as miniflow. The values of it are the mask that 1-bits set on the bits to match. The process of subtable lookup by miniflows of packets and subtable mask is like following. The header fields of the packet that the subtable miniflow bitmap indicates are extracted. It starts from the rightmost bit in mask bitmap, compares the bit of packet bitmap that at the same position. If the bit in packet bitmap is zero, the extracted value is zero. Else, if the bit in packet bitmap is one, the packet value is retrieved from its value array. The values index is the same as the number of 1-bits at the right side of the current rightmost mask bit. Subtable mask values are applied to the extracted packet values to wipe the don't care bits out and fed to the hash function.

Limitation of Naïve SIMD Implementation: The approach to naïvely convert the control flow to the data flow with existing data structure and manipulation algorithm to process several packets in one vector degrades performance in practice. Although the operations comprising the most of this process such as arithmetic operations(plus and minus), bitwise operations(AND) can be parallelized with SIMD vector operations.

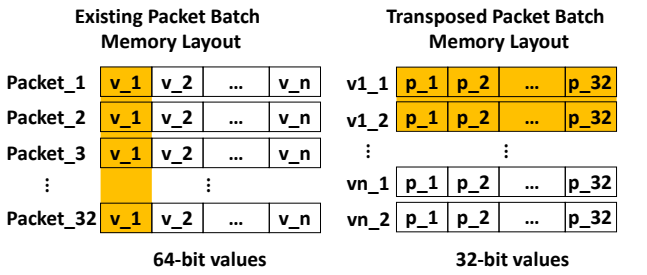

Figure 4: Existing and new data structure memory layout

The fundamental problem is the array of structure form of packet batch data structure as shown in Figure 4. Loading multiple packet data requires to access multiple discontiguous memory addresses and the possibility of all the data accessing hit the cache is far lower than that of accessing single data. As a result, the overhead of loading data to the SIMD vector offsets the benefits from the vectorized hash computing.

To remove the memory access overhead in the vectorized program, we propose a new data structure which is the transposed form of the existing one and corresponding vectorized algorithm to handle it.

Transposed Data Structure: To retrieve the packet data from one contiguous memory block, the form of structure of array is preferred. We come up with the transposed miniflow data structure as shown in Figure 4. The first 64-bit values of the packets in the batch(maximum 32 packets) are arranged into two arrays. We divide a 64-bit value into a higher 32-bit value and a lower 32-bit value and pack all lower 32-bit values from 32 packets in one array, all higher 32-bit values pack into the other array. The reason for truncation is that internal of the hash function on 64-bit value is actually done by hashing lower 32-bit value first and higher 32-bit value followed. By separating two 32-bit values into different arrays, we can directly load lower/higher 32-bit values instead of loading a whole 64-bit value and then do extra vector operation to separate them. The transposed miniflow is generated in addition to the normal miniflow. When the packet header is parsed and the value is stored in the miniflow, we copy the value to the transposed miniflow structure.

Vectorized Algorithm: With the transposed data structure, loading the packet value does not access multiple memory addresses, instead, loading from a contiguous block. A 512-bit vector contains 16 32-bit values is passed to the vectorized hash function and 16 hash values are produced in one iteration. SIMD instructions are effective when the vector is full of data, yet they result in lower throughput per cycle compared to scalar instructions. As a result, processing a few data using vector register costs more time than serial processing. Thus, we set the threshold for vector processing such that we execute SIMD code only when the number of packets remaining in the batch is larger than the threshold. We conduct micro-benchmark to determine the threshold which is 8 in our case.

In the vectorization algorithm, we assume that the network traffic has a majority type of packets such as TCP, UDP, or any tunneling protocols. The packets in the same protocol will have the same miniflow bitmap, and it ensures the values in an array of the transposed miniflow are the data of the same header field. For those exceptional packets having different miniflow bitmap with the majority, different field values will be stored in the array. Taking account of the fact that an array in the transposed miniflow structure is deemed to be all the same field values and be retrieved for computing the signature, an exceptional packet is getting the wrong signature value for it. It results in the megaflow cache miss and the packet will be passed to costly OpenFlow table lookup which is an undesired case. To prevent the exceptional packets to miss the megaflow cache, we mark the exceptional packets in the batch before the megaflow cache lookup. For each subtable, when the number of packets is larger than the threshold, we first process the batch of packets with AVX-512 vectorized code and do the extra serial computing only for the exceptional packets.

4 EVALUATION

4.1 Bloom Filter

We evaluate the performance of AVX-512 vectorized bloom filter which we call HPBF(highly vectorized bloom filter). From the evaluation we emphasize that naïvely adopting SIMD instructions is leading to suboptimal performance than it actually could achieve. We show the impact of data flow conversion using new instructions and memory access latency hiding techniques give to the performance in the case study.

The experiment for the bloom filter lookup is executed on the platform with Intel Xeon Silver 4114 at 2.2GHz. The CPU has 32KB L1 data cache, 1MB L2 cache and 10M L3 cache on-chip. We randomly generate 1M keys following the uniform distribution. We measure the lookup throughput in millions of operations per second. The number of elements in the filter is set to 1/10 of the filter bits and 3 hash functions to get the moderate false positive rate. Error bars show a 95% confidence interval.

First we show the performance improvement of HPBF which extends the vector size from 256 to 512 as well as adopts *expand load/compress store* instructions into data flow instead of the permutation table. We only use a single core in the CPU for all the experiments. Figure 5 shows HPBF outperforms AVX/AVX2(256-bit) baseline up to 94%. The varying size of bloom filters that don't exceed the capacity of L1 cache, L2 cache, L3 cache, as well as the one that exceeds all level of caches are tested. The throughput and the speedup rate are decreased as the size of the bloom filter increases as memory accessing dominants the performance. The naïve AVX-512 implementation, which only extended the size of the vectors while retaining the same algorithm has little improvement due to an exponentially enlarged permutation table.

Figure 6 shows the loop unrolling optimization affects the performance upon the parallelized algorithm. The baseline is 256-bit vectors with the factor of 2 loop unrolling. Loop unrolling optimization has a drastic improvement on the throughput up to 162% compared to the baseline. The throughput grows as the loop unrolling factor increases until a certain number, then degrades with more loop unrolling. That is because aggressive loop unrolling might incur register threshing. We also find that the loop unrolling factor at the peak of the performance increases with the growing filter size.

Figure 7 shows the impact of prefetching upon loop unrolling when the size of the bloom filter is large. Loop unrolling hides the memory access latency relying on the advanced out-of-order execution capabilities in modern CPUs. We find that prefetching can further optimize the memory access latency. It outperforms baseline which is using 256-bit with factor of 2 loop unrolling up to 65% and the loop unrolling implementation without prefetching 25%.

4.2 Open vSwitch

We use two servers with Intel Xeon Gold 6128 CPU at 3.40 GHz, one for running the OvS, and the other one for running the software packet generator MoonGen[7]. Two machines are connected by dual-port 10 Gbps NICs. We generate minimum-size 64-byte TCP packets with line-rate using MoonGen packet generator. We run OvS v2.11.1 with DPDK with the PMD thread pinned to an isolated core. OvS-DPDK bridge is set up with the rules to match all the packets and switch the packets to the other port. We turn off the Exact Match Cache(EMC) to benchmark the performance of standalone megaflow cache performance. It can be regarded as the worst case performance with random and a large number of short lived connections that many packets miss the EMC.

Figure 8 shows the time cost for producing signature values for one packet batch in a subtable. For serial implementation, the time cost is proportional to the number of packets in the batch that have not yet found a matching rule. However, AVX-512 vectorized megaflow runs two iterations no matter how many packets left. The reason is AVX-512 processes 16 consecutive packets in the batch. It doesn't pick the remained packets to pack the vector, since record the packets' indices in the batch and gather the data from discontiguous memory cost extra processing time. To make sure all positions in the batch are processed, we run two iterations either the number of packets is less or larger than 16. Thus, AVX-512 vectorized megaflow cache has the same processing time across the x-axis. We also compare with the performance of CRC hash algorithm which is an alternative hash function in OvS using CRC intrinsic functions in SSE ISA. We find that when the number of packets in the batch is less than 8, serial processing with CRC is faster than AVX-512. So we set the threshold value to 8 according to this result.

We evaluate the end-to-end throughput of the vectorized OvS with varying numbers of subtable traversal per packet. Figure 9 shows AVX-512 implementation with transposed data structure outperforms all the serial implementation when the number of subtables is larger than 5. And the speedup rate

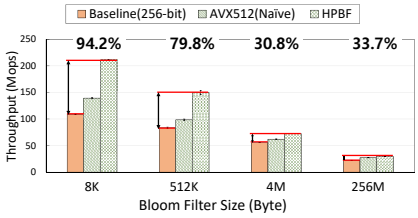

Figure 5: Bloom filter throughput with varying SIMD vector size implementation

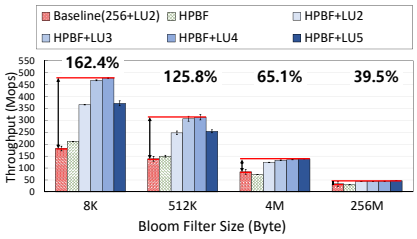

Figure 6: Bloom filter throughput with varying number of loop unrolling

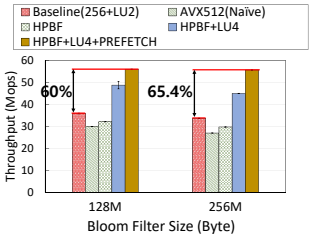

Figure 7: Bloom filter throughput of large size

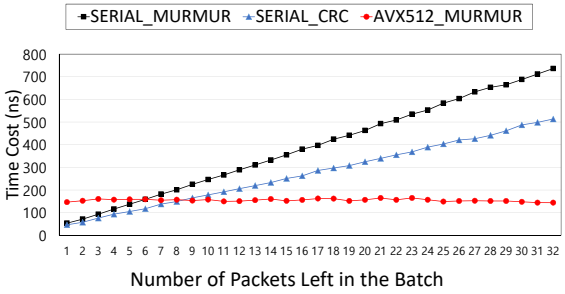

Figure 8: Time cost for computing signature values for a batch of packets with varying number of packets left

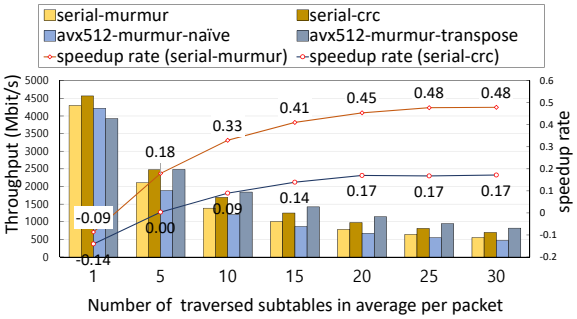

Figure 9: End-to-end throughput for TCP packet traffic

keeps growing up to 17% compared to CRC and 48% compared to murmur hash function.

5 RELATED WORK AND FUTURE WORK

SIMD parallelism has been studied for several database systems. Zhou et.al. [29] proposed vectorized database operations including sequential scans, aggregation, index operations etc. Cargri et.al explored SIMD implementations of sort-merge join and radix-hash join [1]. There are some parallel sorting algorithms for SIMD processors [2, 11]. Harald Lang et.al presented the algorithm that addresses control flow divergence for the data-parallel query pipelines [17]. Orestis and Kenneth proposed vectorized bloom filter [23] to efficiently utilize vector lanes.

SIMD capabilities are also adopted in various network applications and frameworks. DPDK [4] has vector PMD as well as hash table library which optimizes packet I/O and comparison operations using SSE/AVX instructions. VPP [27] also makes use of SIMD instructions for its processing. However, neither of them is adopting SIMD at the code with control flow.

In the future, we will adapt our bloom filter design to the real world network applications and also apply vectorized packet classification to other network applications.

6 CONCLUSION

In this paper, we proposed the effective ways to leveraging SIMD parallelism to improve the performance of the data structure and algorithm used in network applications. Not only the conversion of the algorithm from control flow to the data flow but also memory access latency hiding and SIMD aware data structure design should be considered. We showed the impact of those by two case studies which are AVX-512 vectorized bloom filter and the vectorized OvS megaflow cache. We show that the performance of the bloom filter lookup improved up to 162% and 48% for OvS.

ACKNOWLEDGMENTS

This work is supported by Institute of Civil Military Technology Cooperation Center (ICMTC) funded by the Korea government (MOTIE & DAPA) [18-CM-SW09]; and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R-20160222-002755, Cloud based Security Intelligence Technology Development for the Customized Security Service Provisioning)

REFERENCES

[1] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[2] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1313–1324.

[3] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, Kyoungsoo Park, and Dongsu Han. 2016. {DFC}: Accelerating String Pattern Matching for Network Applications. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 551–565.

[4] Data Plane Development Kit [n.d.]. Data Plane Development Kit. https://www.dpdk.org/.

[5] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. 2003. Deep packet inspection using parallel bloom filters. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.* IEEE, 44–51.

[6] Keith Diefendorff, Pradeep K Dubey, Ron Hochsprung, and HASH Scale. 2000. Altivec extension to PowerPC accelerates media processing. *IEEE Micro* 20, 2 (2000), 85–95.

[7] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*. ACM, 275–287.
[8] Shahabeddin Geravand and Mahmood Ahmadi. 2013. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks* 57, 18 (2013), 4047–4064.
[9] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing {GPU} as Packet Processing Accelerator. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 83–96.
[10] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2011. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 195–206.
[11] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. 2007. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. IEEE, 189–198.
[12] Intel 64 and IA-32 Architectures Software Developer's Manual [n.d.]. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf.
[13] Intel Intrinsics Guide [n.d.]. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide.
[14] Keon Jang, Sangjin Han, Seungyeop Han, Sue B Moon, and KyoungSoo Park. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors.. In *NSDI*. 1–14.
[15] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G Andersen. 2015. Raising the bar for using GPUs in software packet processing. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 409–423.
[16] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (network balancing act): A high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 22.
[17] Harald Lang, Linnea Passing, Andreas Kipf, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2020. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. *The VLDB Journal* 29, 2 (2020), 757–774.
[18] Samuel Larsen, Rodric Rabbah, and Saman Amarasinghe. 2005. Exploiting vector parallelism in software pipelined loops. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 11–pp.
[19] Daniel Lustig and Margaret Martonosi. 2013. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 354–365.
[20] Gaurav Mitra, Beau Johnston, Alistair P Rendell, Eric McCreath, and Jun Zhou. 2013. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1107–1116.
[21] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The design and implementation of open vswitch. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 117–130.
[22] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1493–1508.
[23] Orestis Polychroniou and Kenneth A Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*. ACM, 6.
[24] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. Fast hash table lookup using extended bloom filter: an aid to network processing. In *ACM SIGCOMM Computer Communication Review*, Vol. 35. ACM, 181–192.
[25] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet classification using tuple space search. In *ACM SIGCOMM Computer Communication Review*, Vol. 29. ACM, 135–146.
[26] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. {GASPP}: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 321–332.
[27] VPP [n.d.]. The Vector Packet Processor (VPP). https://fd.io/docs/vpp/master/whatisvpp/index.html.
[28] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment* 2, 1 (2009), 385–394.
[29] Jingren Zhou and Kenneth A Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 145–156.