VATE: a trade-off between memory and preserving time for high accuracy cardinalities estimation under sliding time window

Jie Xu\textsuperscript{a,}\textsuperscript{*}, Wei Ding\textsuperscript{b}, Xiaoyan Hu\textsuperscript{b}

\textsuperscript{a}School of Computer Science and Engineering, South East University, Nanjing, China
\textsuperscript{b}School of Cyber Science and Engineering, South East University, Nanjing, China

Abstract

Host cardinality is one of the important attributes in the field of network research. The cardinality estimation under sliding time window has become a research hotspot in recent years because of its high accuracy and small delay. This kind of algorithms preserve the time information of sliding time window by introducing more powerful counters. The more counters used in these algorithms, the higher the estimation accuracy of these algorithms. However, the available number of sliding counters is limited due to their large memory footprint or long state-maintenance time. To solve this problem, a new sliding counter, asynchronous timestamp (AT), is designed in this paper which has the advantages of less memory consumption and low state-maintenance time. AT can replace counters in existing algorithms. On the same device, more AT can be used to achieve higher accuracy. Based on AT, this paper designs a new multi-hosts cardinalities estimation algorithm VATE. VATE is also a parallel algorithm that can be deployed on GPU. With the parallel processing capability of GPU, VATE can estimate cardinalities of hosts in a 40 Gb/s high-speed network in real time at the time granularity of 1 second.

1. Introduction

Measuring the attributes of the core network, such as traffic size, packet number, host cardinality and so on, is the basis of network management and research. This paper mainly studies how to estimate the cardinality of host under sliding time window. Cardinality refers to the number of distinct elements in a data stream over a period of time. It is an important attribute in network management and research, and plays an important role in many network applications, such as DDoS attack detection\textsuperscript{[1]}\textsuperscript{[2]} and network scanning\textsuperscript{[3]}. In the network domain, cardinality can be flow cardinality (the number of distinct flows in a period\textsuperscript{[4]}), host cardinality (the number of other hosts communicating with it\textsuperscript{[5]}\textsuperscript{[6]}). The cardinality estimation algorithm can be applied to all these problems. In this paper, the estimation of host cardinality is taken as the research object, and the algorithm studied in this paper can also be applied to the estimation of flow cardinality.

Suppose there is a core network ANet, which is managed by some organisation, institute or ISP (Internet Service Provider). ANet communicates with other networks through a group of edge routers ER. Use BNet to represent other networks that communicate with ANet. For a host aip in ANet, its cardinality refers to the number of hosts in BNet that communicate with it through ER in a time window. We call these hosts that communicate with aip in BNet the opposite hosts of aip. They communicate with aip in a certain time window, that is, they send packets to aip or receive packets from aip. In this paper, the task of host cardinality estimation is to estimate the cardinality of each aip in ANet by scanning all packets passing through the ER.

The time window can be a discrete time window or a sliding time window, as shown in Figure 1. Divide the network traffic by successive time slices. The size of the time slice can be set to 1 second, 1 minute, 5 minutes or other lengths suitable for different applications.

The time window moves forward one time slice at

\textsuperscript{*}Corresponding author

Email addresses: xujieip@163.com (Jie Xu),
ding@carnation.njnet.edu.cn (Wei Ding),
huxiaoyan@carnation.njnet.edu.cn (Xiaoyan Hu)
a time. A sliding time window can contain up to $k$ consecutive time slices, while a discrete time window has only one time slice. Let $W(t, k')$ denote a time window consisting of consecutive $k'$ time slices starting from time slice $t$, where $k'$ is a positive integer less than or equal to $k$. The size of the time window is the length of time it covers. For a sliding time window containing $k'$ time slices, its size is the sum of the lengths of $k'$ time slices. For discrete time windows, the size is the length of a time slice. It is assumed that the cardinality estimation is performed only at the end of each time slice, that is, at the end of the time slice $t + k' - 1$, the cardinalities of the hosts in $W(t, k')$ is estimated.

Finer-grained time slice makes the cardinality obtained under sliding time window more accurate and timely than that obtained under discrete time window. But the calculation of cardinality under sliding time window is much more complicated than that under discrete time window, because when sliding time window moves forward, it needs to keep the state of opposite hosts in the previous time slices. The state of opposite host is defined as follows:

**Definition 1** (Active opposite hosts). Let $OP(aip, t, k')$ represents $aip$'s set of opposite hosts in the time window $W(t, k')$. If $bip$ belongs to $OP(aip, t, k')$, it is said that $bip$ is active for $aip$ in time window $W(t, k')$.

The cardinality estimation under sliding time window must preserve the active opposite hosts in the previous $k - 1$ time slices. In the cardinal estimation algorithm, $aip$ may contain multiple counters and each counter may correspond to various opposite hosts. The state of the counter is determined by its opposite hosts.

**Definition 2** (Active counter). In the time window $W(t, k')$, when one of the opposite hosts corresponding to a counter is active, the counter is active; if all the opposite hosts corresponding to it are inactive, the counter is inactive.

Timestamp (TS) is the earliest counter for sliding time window. TS recorded the latest appearance time of its corresponding opposite hosts. At a time point, the TS is inactive if the difference between the current time and TS is greater than the size of the time window. The state of TS can be obtained at any time. Because the sliding time window keeps moving forward, TS must be large enough, such as 32 or 64 bits. High memory requirements limit the number of timestamps.

For each counter, at the end of a time slice, we only want to determine whether it is active or not. In other words, we are interested in whether the latest opposite host appears in the current time window. Since the length of each time slice is invariant and a sliding time window contains $k$ time slices at most, $\log_2(k + 1)$ bits are theoretically sufficient to preserve the counter’s state. Distance Recorder (DR) is proposed under this idea. Each DR occupies $\lceil \log_2(k + 1) \rceil$ bits, where the $\lceil x \rceil$ function represents the smallest integer no smaller than $x$. Each bit of DR is set to 1 at the beginning. The DR is set to 0 if the corresponding host of the DR appears in the current time slice. Each time the time window slides, the state of DR needs to be updated, that is, the value of DR plus 1. At the end of time slice $t$, if the value of DR is less than $k'$, the DR is active in $W(t - k' + 1, k')$. Unlike TS, DR uses only $\lceil \log_2(k + 1) \rceil$ bits. But the state of DR needs to be maintained at the end of each time slice. When the number of DRs is large, maintaining the state of each DR is a heavy task. The state-maintenance time of DR limits the number of DRs available.

Both TS and DR have their strengths and weaknesses. Can we use less memory and less maintenance time to determine the state of the counter? If we want to save memory, we must regularly maintain the state of each counter and mark inactive ones; if we want fewer state-maintenance operations, we can reduce the number of counters whose state need to be maintained in each time slice. Inspired by this idea, we propose a new counter, asynchronous timestamp (AT). AT is a tradeoff between
memory consumption and state-maintenance time. It contains $\text{ceil}(\log_2(2 \times k + 1))$ bits. Although $AT$ has one bit more than $DR$, $AT$ only needs to maintain its state once every $k$ time slices. In other words, $AT$ reduces the time complexity of state-maintenance to $O(1/k)$ at the cost of an extra bit. Based on $AT$, an efficient parallel algorithm, virtual asynchronous timestamp estimator ($VATE$), is proposed to estimate the cardinalities under sliding time windows. The main contributions of this paper are as follows.

- A new counter, asynchronous timestamp ($AT$), is proposed. $AT$ only occupies $\text{ceil}(\log_2(2 \times k + 1))$ bits and can maintain its state once every $k$ time slices. $AT$ can determine its state at the end of each time slice and is suitable for running in the parallel environment.

- A new algorithm for estimating the cardinalities under sliding time window is proposed, which is called virtual asynchronous timestamp estimation algorithm $VATE$. $VATE$ uses a fixed number of $AT$s to estimate cardinalities. Because a single $AT$ takes up less memory and has low state-maintenance time, $VATE$ can contain more $AT$s to achieve higher estimation accuracy than other algorithms.

- The $VATE$ algorithm is deployed in GPU to acquire real-time cardinalities estimation in high-speed network under sliding time windows.

- Using real high-speed network data, we compare the accuracy of $VATE$ under different parameters and the running time on different GPU.

This article is arranged as follows. In the next section, we will introduce existing works of cardinality estimation under sliding time window. In Section 3, we will describe how $AT$ works and why it can use less memory and less maintenance time to determine its own state. Section IV presents an $AT$-based algorithm for estimating cardinalities under sliding time window, $VATE$. It also introduces how to deploy $VATE$ on GPU and realize the cardinalities estimation in parallel. In Section 5, the experiments of real high-speed network traffic are shown. In section 6, we summarize this paper.

2. Background & Related works

2.1. Cardinality estimation

For a host $aip$ in $ANet$, let $Pkt(aip,t,1)$ represent the set of all packets that communicate with $aip$ through $ER$ in the time window $W(t,1)$. From each packet in $Pkt(aip,t,1)$, an IP address pair like $<aip,bip>$ can be extracted, in which $bip$ is the opposite host of $aip$ in the packet. Let $IPair(aip,t,1)$ represent the stream of IP address pairs extracted from $Pkt(aip,t,1)$. Because an opposite host $bip$ may send or receive multiple packets in a time window, the same IP address pair may appear many times in $IPair(aip,t,1)$. The cardinality of $aip$ is calculated by scanning $IPair(aip,t,1)$ to get the number of distinct IP address pairs, namely $|OP(aip,t,1)|$.

In recent years, many excellent cardinality estimation algorithms have been proposed. These algorithms mostly use a vector containing $g$ counters to estimate the cardinality of a host. These algorithms differ in what is saved in each counter, how to update the counter, and how to estimate the cardinality from the counter vector.

Flajolet et al. [10] proposed a cardinality estimation algorithm called PCSA. The counter used by PCSA is a bitmap containing 32 bits. Each host is randomly mapped to a counter, and its lowest significant bit is stored in the counter. When all IP address pairs are scanned, the cardinality is estimated based on the values of all counters.

The task of each counter in PCSA is to record the lowest significant bit of opposite hosts. For IPv4 addresses, the maximum lowest significant bit is 32, which can be expressed in 5 bits. But PCSA uses 32 bits, leaving much room for improvement. For this reason, Philippe et al. [11] proposed LogLog algorithm. Unlike PCSA, each counter in LogLog records the position of the “1” bit on the leftmost side of opposite hosts. LogLog estimates the cardinality based on the geometric mean of all counters. Many algorithms are derived from LogLog. Flajolet et al. [12] found that when using the harmonic mean of all counters, the accuracy of the estimation results would be improved. Based on this idea, the HyperLogLog algorithm is proposed. MinCount [13] is another algorithm similar to LogLog. But it first hashes each host evenly to a real number between $[0,1]$, and each counter stores the minimum hash value it has seen. The size of each counter can be adjusted according to different accuracy.
Although the above algorithms are effective for large cardinality, their accuracy is limited. Whang et al [14] proposed a high-precision cardinality estimation algorithm based on the maximum likelihood principle, linear estimator LE. A counter in LE has only one bit, and all counters are initialised to zero at the beginning. Each opposite host is mapped to a counter in LE by a random hash function[15]. When an opposite host appears, its corresponding counter will be set to 1. At the end of a time window, LE estimates the cardinality of $a_{ip}$ according to the following formula, where $g_0$ is the number of ‘0’ bits.

$$|OP(a_{ip}, t, k)| = -g_0 \cdot \ln\left(\frac{g_0}{g}\right)$$  \hspace{1cm} (1)

The accuracy of the cardinality estimation algorithm is evaluated by standard error[16]. Let $n$ denote the cardinality of $a_{ip}$ and $n'$ denote the estimated value obtained by an algorithm. The standard deviation of the estimation algorithm is the standard deviation of $n/n'$, which is recorded as $\sigma$. Table 1 shows the accuracy and memory consumption of different algorithms when $\sigma = 1$ and $n = 5000$.

From Table 1, we can see that LE uses the smallest memory to achieve the same accuracy as other algorithms. In this paper, based on the principle of LE, we design a cardinality estimation algorithm under the sliding time window.

### 2.2. Sliding time window and distinct time window

As shown in Figure 1, discrete time window and sliding time window are two commonly used time windows in cardinality estimation. The sliding time window moves forward one time slice at a time. Therefore, two adjacent sliding time windows containing $k$ time slices have $k - 1$ same time slices. In Figure 1, the size of the time slice is set to 1 second for the sliding time window and 300 seconds for the discrete time window. Sliding time window and discrete time window have the same length, but the sliding time window contains 300 time slices. The granularity of the time slice is smaller than that of the discrete time window.

The cardinality estimation under the discrete time window is relatively simple because it does not need to save the state of the previous time slices. However, the estimation results of cardinalities are influenced by the time window boundary, and the estimation delay is significant. Compared with the discrete time window, the cardinality calculation under the sliding time window has higher accuracy because the sliding time window can monitor network traffic[17] on a finer-grained time slice.

There are many algorithms[18][17][19] to solve this problem by using better-functioning counters. Fusy et al [18] improved MinCount by using a data structure called List of Future Possible Minima(LFPMin), and proposed a Sliding MinCount. Sliding MinCount replaces each counter in the MinCount with LFPMin so that it can save the smallest packet in the current time window. But the accuracy of Sliding MinCount depends on MinCount, and the length of LFPMin increases with the number of IP address pairs. Inspired by LFPMin, Chabchoub et al [17] proposed a data structure called List of Future Possible Maxima(LFPMax) to replace each counter in HyperLogLog, and proposed Sliding HyperLogLog. The accuracy of Sliding HyperLogLog depends on HyperLogLog, and the length of LFPMax increases with the number of IP address pairs. The size of LFPMin and LFPMax is not fixed. When a time window contains $k$ time slices, LFPMin and LFPMax can contain $k$ elements at most or only one element at least. The sizes of LFPMin and LFPMax need to be adjusted dynamically at runtime or reserved the maximum space before running. Dynamically adjusting the size of memory will increase the operating burden of the system, while reserving the maximum required memory space will waste a lot of memory.

Because LE algorithm has high estimation accuracy and simple operation, many algorithms are designed or improved on the basis of LE. Kim et al [19] replaces bit vectors in LE with time stamp vectors, and proposes a cardinality estimation algorithm under sliding time window based on timestamp vectors, TSV. Each timestamp contains 64 bits. TSV can estimate the cardinality under any time window of any size. But in practice, we don’t need to query the cardinality of the host in this way. For windows with $k$ time slices, the size of each counter can be as small as only $\log_2(k + 1)$ bits, such as DR proposed by Jie et al[9].

| Alg       | StdErr($\sigma = 1$) | Mem (bit) | Tau (Mbit) |
|-----------|----------------------|-----------|------------|
| MinCount  | $1/2^{d_0}$          | 32 bit    | 10000 bits |
| PCSA      | $1/2^{d_0}$          | 32 bit    | 10000 bits |
| LogLog    | $1/2^{d_0}$          | 0 bit     | 64000 bits |
| HyperLogLog| $1/2^{d_0}$         | 0 bit     | 50000 bits |
| Linear Estimator | $1/2^{d_0}$         | 1 bit     | 5000 bits  |
A DR only occupies $\log_2(k + 1)$ bits to record its own state of activity accurately. DR can replace each counter in the ACE [21] to estimate cardinalities under sliding time window and we denote this new algorithms as VDRE. But when the time window slides, the values of each DR need to be updated to maintain their state. When a large number of counters are included, the state-maintenance operation of DR takes a lot of time and becomes the bottleneck of real-time running.

In order to improve the speed, Jingsong et al proposed CVS [22] algorithm and LRU-Sketch [23] algorithm. CVS adopts the strategy of random updating, and randomly selects a part of the counter to update each time. However, the method of random updating will increase the error of the estimation results. The LRU-Sketch algorithm uses the principle of memory page replacement to delete inactive counters. But the LRU-Sketch algorithm runs in a special sliding time window, that is, there is at most one packet in a time slice.

In this paper, an algorithm called virtual asynchronous timestamp vector (VATE) is proposed to estimate cardinalities under the sliding time window. VATE reduces memory occupancy and state-maintenance time at the same time. Table 2 compares the differences between these algorithms. All these sliding cardinality estimation algorithms are based on some classical estimation algorithms in the “basic algorithm” column. “Memory Unit” column is the number of bits required by each counter. The column “PT” represents the time complexity of the state-maintenance operation of a counter in each time slice.

In Table 2, $n$ denotes the number of distinct IP address pairs in a time window, $g$ denotes the number of counters, and $k$ indicates the max number of time slices in a window. Comparing with LFP-Max and TSV, VATE reduces the memory requirements of each counter. Although the counter used in VATE contains one bit more than that in CDV, the state-maintenance time of VATE is decreased greatly. VATE is also a multi-cardinalities estimation algorithm, that is, it can simultaneously estimate multiple hosts’ cardinalities.

2.3. Multi cardinalities estimation

In the core network, there are a large number of hosts. The direct way to get all these hosts’ cardinalities is to assign an estimator to each of them. But it wastes a lot of memory. Some efficient algorithms use a fixed number of counters to calculate cardinalities of all hosts. These algorithms can be divided into two categories: the algorithm based on estimator matrix and the algorithm based on virtual estimator.

The estimator-matrix based algorithm estimates cardinalities of all hosts using the estimator matrix of the u-row and v-column. For each host, an estimator corresponds to it in each row, i.e. a host cardinality is estimated by using u estimators at the same time. For example, Wang et al designed DCDS [24] based on Chinese remainder theorem; Liu et al designed VBF [25] based on Bloom filter principle.

The algorithm based on virtual estimator assigns a logical estimator to each host. The counter of each logical estimator is randomly selected from a counter pool, as shown in Figure 2.

The virtual estimator can be LE [26] [27], HyperLogLog [16] and so on. Every counter in the virtual counter vector is relative to a physical counter in the pool. A physical counter is shared by several virtual counter vector. Let $g$ represent the number of counters in a virtual counter vector and $2^c$ represent the number of counters in the pool. A host’s cardinality is estimated by the equation 2.

$$|OP(aip, t, k)| = g * \ln(Z_v) - g * \ln(Z_p)$$

Equation 2 removes the bias caused by counters sharing. This paper also uses virtual counter vector to estimate host cardinality. But we replace the
counter in the pool with \( AT \) to work under sliding time window. Unlike estimator-matrix based algorithms, virtual estimator based algorithms only need to update a counter for each packet, because a host only corresponds to an estimator. The VATE proposed in this paper is an algorithm based on virtual estimator. VATE uses \( AT \) as the counter of the virtual estimator to estimate multi-host cardinalities under the sliding time window. \( AT \) and VATE are described below in detail.

3. Asynchronous timestamp

In a discrete-time window, one bit is sufficient to indicate whether the host has appeared in the current time window. For example, a bit is initialised to zero at the beginning of a time window. It will be set to 1 if one or some of its opposite hosts appear in the time window; otherwise, it will remain zero. But a bit has only two possible values, that is, the counter is active or inactive in the current time slice. Counter having only one bit limits the application of cardinality estimation under the sliding time window. The critical step in estimating cardinality under sliding time window is to determine whether the counter is active in the current time window (in successive \( k \) time slices). In this section, we will introduce a new counter, Asynchronous Timestamp (\( AT \)), to solve this problem.

\( AT \) is a counter consisting of \( \lceil \log_2(2 \times k + 1) \rceil \) bits. So \( AT \) can represent at least \( 2 \times k + 1 \) different values, from 0 to \( 2 \times k \). Each \( AT \) also has an asynchronous current timestamp (\( ACT \)). \( ACT \) is a runtime variable that does not occupy memory space and has a range of values from 0 to \( 2 \times k - 1 \). The \( ACT \) remains unchanged for a time slice. Let \( at \) denote an instance of \( AT \), \( Value(at) \) denotes the current value of \( at \), and \( ACT^at \) denotes the value of the asynchronous current timestamp of \( at \) in time slice \( t \). The relationship between the values of \( ACT \) in two adjacent time slices conforms to the equation:

\[
ACT^{at}_{t+1} = (ACT^at_t + 1) \mod (2 \times k) \tag{3}
\]

One of the key tasks of \( AT \) is to determine whether it is active in the current time window based on its recorded value and \( ACT \). Let the value \( 2 \times k \) denote the inactive state of \( AT \), that is, if \( Value(at) \) equals \( 2 \times k \), then \( at \) is inactive. When \( Value(at) \) is less than \( 2 \times k \), its activity needs to be determined by \( ACT^at \). \( AT \) has four operations: initializing(\( InitAT() \)), setting(\( SetAT() \)), maintaining state(\( PreserveAT() \)) and checking state(\( CheckAT() \)). In the following section, \( t \) represents the current time slice if there is no special explanation.

- \( InitAT(at) \): Set the value of \( at \) to \( 2 \times k \). \( AT \) is set inactive at initialization.
- \( SetAT(at) \): Set the value of \( at \) to \( ACT^at \).
- \( CheckAT(at, k') \): At the end of the time slice \( t \), return whether \( at \) is active in the time window \( W(t-k'+1,k') \), and \( k' \) is a positive integer not greater than \( k \). The specific operation is shown in the algorithm 1.
- \( PreserveAT(at) \): Update the value of \( at \) at the beginning of some time slices, as shown in the algorithm 2.

Algorithm 1 checkAT

\begin{itemize}
\item \textbf{Input:} Asynchronous timestamp \( at \), Time slices number \( k' \)
\item \textbf{Output:} ActiveState
\end{itemize}

\begin{algorithm}
\begin{algorithmic}[1]
\State \text{act} \Leftarrow ACT^at \label{alg1:1} \\
\If {\text{Value}(at) == 2 \times k} \State Return False \label{alg1:2} \\
\Else \State \text{Return True} \label{alg1:3} \\
EndIf \\
\If {\text{dis} \Leftarrow (\text{act} + 2 \times k - \text{Value}(at)) \mod (2 \times k)} \\
\If {\text{dis} \leq k' - 1} \State Return True \\
\Else \State Return False \label{alg1:4} \\
\EndIf \\
\EndIf \\
\Endalgorithm
\end{algorithm}

Algorithm 2 preserveAT

\begin{itemize}
\item \textbf{Input:} Asynchronous timestamp \( at \), Max time slices number \( k \)
\end{itemize}

\begin{algorithm}
\begin{algorithmic}[1]
\State \text{act} \Leftarrow ACT^at \label{alg2:1} \\
\State \text{dis} \Leftarrow (\text{act} + 2 \times k - \text{Value}(at)) \mod (2 \times k) \label{alg2:2} \\
\If {\text{dis} \geq k} \State \text{Value}(at) \Leftarrow 2 \times k \label{alg2:3} \\
\EndIf \\
\Endalgorithm
\end{algorithm}

Firstly, the algorithm 1 checks whether \( Value(at) \) is \( 2 \times k \), and if so, decides directly that \( at \) is inactive. If not, \( ACT^at \) is used to further determine whether \( at \) is active. According to \( SetAT() \) operation, it is known that \( AT \) saves its
The state of $\text{at}$ is an asynchronous timestamp. When the state of $\text{at}$ is maintained by $\text{PreserveAT()}$ operation at the beginning of time slice $t'$, then in time slice $t'$ and the $k-1$ time slices after $t'$, $\text{CheckAT(}\text{at},k')$ can accurately give whether $\text{at}$ is active, where $k' \leq k$.

Proof. Let $t''$ denote any time slice between time slice $t'$ and time slice $t'+k-1$. According to the setting operation of $AT$ and the algorithm 1, if $\text{at}$ is set in a time slice between $t'$ and $t''$ (setAT($\text{at}$)), then $\text{CheckAT(}\text{at},k')$ can output the exact state of $\text{at}$.

Next, we discuss the case where $\text{at}$ is not set between $t'$ and $t''$. Let $v_0$ and $d_0$ denote the value and distance of $\text{at}$ before it was maintained by $\text{PreserveAT()}$ operation at the beginning of $t'$. When the $\text{PreserveAT()}$ operation is used to maintain the state of $\text{at}$, the value of $\text{at}$ has two situations: the value of $\text{at}$ becomes $2*k(\text{at becomes inactive})$ and the value of $\text{at}$ is still $v_0$. For the first case, $\text{CheckAT(}\text{at},k')$ can accurately identify $\text{at}$ as inactive at the end of time slice $t''$. For the second case, $\text{CheckAT(}\text{at},k')$ needs to determine whether $\text{at}$ is active according to the $\text{Dist(}\text{at})$ at the end of $t''$. If the distance at the end of $t''$ does not exceed $2*k$, $\text{CheckAT(}\text{at},k')$ can give the state of $\text{at}$ accurately. According to the process of $\text{PreserveAT()}$ operation, the value of $\text{at}$ remains unchanged only when the distance of $\text{at}$ is less than $k$. So at the end of $t''$, if $\text{Value(}\text{at})$ equals $v_0$, then $d_0 < k$. At the end of $t''$, $\text{Dist(}\text{at}) = d_0 + t'' - t'$, and $t'' - t' < k' \leq k$. So $\text{Dist(}\text{at}) < 2*k$, and $\text{CheckAT(}\text{at},k')$ can determine whether $\text{at}$ is active according to whether $\text{Dist(}\text{at})$ is less than $k'$.

In summary, for any of the above cases, $\text{CheckAT(}\text{at},k')$ can accurately give the state of $\text{at}$ in the time slice $t'$ and the $k-1$ time slices after $t'$.

Based on theorem 1, corollary 1 gives a method to determine when to maintain the state of $AT$ according to the value of $ACT$.

Corollary 1. Let $\text{at}$ be an asynchronous timestamp. If $\text{PreserveAT()}$ is performed on $\text{at}$ when $ACT_{t'} = 0$ or $k$, at the end of any time slices, $\text{CheckAT(}\text{at},k')$ can accurately give whether $\text{at}$ is active or not, where $k' \leq k$.

Proof. Assuming $ACT_{t'} = 0$, according to the formula 3 we can get $ACT_{t'+k} = k$ and $ACT_{t'+k+k} = 0$. According to Theorem 1 if $\text{PreserveAT(}\text{at})$...
is executed at the beginning of time slice \( t' \), \( \text{CheckAT}(\text{at}, k') \) can give the state of \( \text{at} \) accurately between time slice \( t' \) and time slice \( t' + k - 1 \); if \( \text{PreserveAT}(\text{at}) \) is executed at the beginning of time slice \( t' + k \), then between time slice \( t' + k \) to time slice \( t' + k + k - 1 \), \( \text{CheckAT}(\text{at}, k') \) can give the state of \( \text{at} \) accurately. Because \( \text{PreserveAT}() \) is performed when \( \text{ACT}_{t'}^{at} \) is 0 or \( k \), \( \text{CheckAT}(\text{at}, k') \) can accurately give the state of \( \text{at} \) between time slice \( t' \) and time slice \( t' + k + k - 1 \). When the time slice \( t' + k + k \) is entered, the \( \text{PreserveAT}() \) operation will maintain the state of \( \text{at} \) again. So at the end of any time slices, \( \text{CheckAT}(\text{at}, k') \) can accurately give whether \( \text{at} \) is active.

According to corollary 1, the state-maintenance operation of \( \text{AT} \) can be simplified. When \( \text{PreserveAT}() \) operation uses the algorithm 2 for \( \text{AT} \) state-maintenance, it is necessary to calculate the distance of \( \text{AT} \), and then compare it with \( k \). According to the formula 5, it requires one addition, one subtraction and one mod operation to calculate \( \text{AT} \) distance. So the algorithm 2 needs four operations to determine whether the distance of \( \text{AT} \) exceeds \( k \): one addition, one subtraction, one mod and one comparison. When corollary 1 is used for state-maintenance, only two or three comparing operations are needed to determine whether the distance of \( \text{AT} \) exceeds \( k \). Because when \( \text{ACT}_{t'}^{at} \) is given, we can know which \( \text{Value}(\text{at}) \) causes \( \text{Dist}(\text{at}) \) to be greater than or equal to \( k \).

According to the formula 5 at the beginning of time slice \( t \), when \( \text{ACT}_{t'}^{at} = 0 \), if \( 0 \leq \text{Value}(\text{at}) \leq k \) and \( \text{at} \) is not set in time slice \( t \), then at the end of time slice \( t \), \( \text{Dist}(\text{at}) \geq k \); when \( \text{ACT}_{t'}^{at} = k \), if \( k \leq \text{Value}(\text{at}) \leq 2k - 1 \) or \( \text{Value}(\text{at}) = 0 \), and \( \text{at} \) is not set in time slice \( t \), then \( \text{Dist}(\text{at}) \geq k \) at the end of time slice \( t \). So at the beginning of time slice \( t \), if \( \text{ACT}_{t'}^{at} = 0 \) and \( 0 \leq \text{Value}(\text{at}) \leq k \), then \( \text{at} \) becomes inactive; if \( \text{ACT}_{t'}^{at} = k \) and \( k \leq \text{Value}(\text{at}) \leq 2k - 1 \) or \( \text{Value}(\text{at}) = 0 \), then \( \text{at} \) becomes inactive.

For example, when \( k = 9 \), Figure 3 illustrates how to maintain the state of \( \text{AT} \). In Figure 3, the value on the edge of the circle represents the possible values of \( \text{Value}(\text{at}) \). In time slice \( t \), if \( \text{ACT}_{t'}^{at} = 17 \), then \( \text{ACT}_{t'}^{at} = 0 \). According to corollary 1 at the beginning of time slice \( t + 1 \), the state of \( \text{at} \) should be maintained. As can be seen from the upper part of Figure 3 at the end of time slice \( t \), if \( \text{Value}(\text{at}) \) is between \([10,17]\) and \( \text{at} \) is not set in time slice \( t + 1 \), then \( \text{Dist}(\text{at}) < 9 \) at the end of time slice \( t + 1 \), and in this case, the value of \( \text{at} \) remains unchanged. But at the end of time slice \( t \), if \( \text{Value}(\text{at}) \) is between \([0,9]\) and \( \text{at} \) is not set in time slice \( t + 1 \), then \( \text{Dist}(\text{at}) \geq 9 \) at the end of time slice \( t + 1 \), and in this case, \( \text{at} \) should be marked inactive. By analogy, when \( \text{ACT}_{t'}^{at} = 9 \), if \( \text{Value}(\text{at}) \) is between \([9,17]\) or \( \text{Value}(\text{at}) \) is equal to 0, \( \text{at} \) is marked inactive. A slight difference from the case when \( \text{ACT}_{t'}^{at} = 0 \) is that when \( \text{ACT}_{t'}^{at} = k \), one more comparing operation (compare whether \( \text{Value}(\text{at}) = 0 \)) is needed to determine whether \( \text{Dist}(\text{at}) \) is less than \( k \).

\( \text{PreserveAT}() \) operation can use the algorithm 3 to maintain \( \text{AT} \) state more quickly when following corollary 1. The algorithm 3 uses two or three comparisons to determine whether \( \text{AT} \) needs to be marked inactive. The algorithm 3 is faster than the algorithm 2 because it does not need addition and mod operations.

In the cardinality estimation algorithm, a large number of fixed counters are used. Because \( \text{AT} \) preserves its state incrementally, when \( \text{AT} \) is used as the counter of these algorithms, cardinality can be estimated under sliding time window continuously. If there are \( 2k \) \( \text{ATs} \) and their \( \text{ACTs} \) are different from each other, only 2 \( \text{ATs} \) need \( \text{PreserveAT}() \) operation at the beginning of each time slice. Based on this idea, we propose a virtual asynchronous timestamp estimator VATE. It estimates the cardinality under sliding time window based on a fixed length \( \text{AT} \) pool.
Algorithm 3 preserveAT

Input: Asynchronous timestamp at, Time slices number k
1: act ← the ACT of at
2: if act == 0 then
3: if 0 ≤ Value(at) ≤ k then
4: Value(at) ← 2 * k
5: end if
6: end if
7: if act == k then
8: if k ≤ Value(at) ≤ 2 * k − 1 then
9: Value(at) ← 2 * k
10: end if
11: if Value(at) == 0 then
12: Value(at) ← 2 * k
13: end if
14: end if

4. Cardinalities estimation under sliding time window

To calculate the cardinality of all hosts in ANet, we only need to extract IP address pairs, like \{< aip, bip > | aip ∈ ANet, bip ∈ BNet\}, from network packets.

When ANet is a core network, such as a city-wide network, there are millions of hosts in it. One way to accurately estimate the cardinality of each host is to save and maintain the state of each host and their opposite hosts. For example, an AT is assigned to each opposite hosts of aip, and the number of active ATs (acquired by CheckAT()) operation in each time window is counted to obtain the cardinality of aip under the sliding time window. Although this method can get accurate results, it takes up a lot of memory and processing time for the core network. Inspired by GSE, a new algorithm, virtual asynchronous timestamp estimator (VATE), is proposed to estimate the cardinality of each host.

4.1. Virtual asynchronous timestamp estimator

In cardinality estimation algorithms, each host usually uses the same number of counters. For hosts in ANet, their cardinality ranges from 1 to tens of thousands. The larger the cardinality, the more counters the host needs to get an accurate estimation. However, allocating the same number of counters for low-cardinality hosts and high-cardinality hosts causes memory waste. If AT can be shared by multiple hosts, the utilization of counters owned by low-cardinality hosts can be improved. VATE is put forward under this idea. Let ATP denote the AT pool consisting of 2e ATs and ATP[i] denote the i-th AT in ATP. VATE assigns g virtual ATs to each host, and each virtual AT corresponds to a physical AT in the ATP. VATE is derived from GSE (in Figure 2). Unlike GSE, VATE uses AT instead of bits, as shown in Figure 4.

The state of AT in ATP is maintained by corollary 1. If each AT in ATP has the same ACT, there are 2e ATs needing state-maintenance in these time slices where ACT equal to k or 0. But no AT needs state-maintenance in other time slices. At this time, the load of the system is unbalanced in different time slices. Moreover, when the value of c is big (for example, c is larger than 27), the time of state-maintenance in these time slices, where the ACT is 0 or k, will exceed the length of a time slice, and it causes that the algorithm can not run in real time. Therefore, it is necessary to reduce the peak time of state-maintenance operations in all time slices.

Because when to maintain the state of AT is determined by its ACT, and ACT has 2 * k different values (from 0 to 2 * k − 1), ATP can be divided according to ACT values. In this paper, AT in ATP is divided into 2 * k blocks, each AT in a block has the same ACT. The ACT of a AT block is defined as the ACT of the AT in it. Let B[i] denote the i-th AT block, |B[i]| denote the number of ATs in B[i], and BACTi denote the ACT of B[i] in time slice t, \(0 ≤ i ≤ 2 * k − 1\). The ACTs of any two AT blocks are different, and the ACTs of adjacent AT blocks have the following relationship:

\[ BACT_{i+1} = (BACT_i + 1)(mod(2 * k)) \]  (6)

In any time slice t, when the ACT of any AT block is known, the ACT of other AT blocks are calculated according to the formula 6. So we only need to record and maintain the ACT of an AT block. In this paper, we record the ACT of the first AT block in memory and update its value at the
the beginning of each time slice. The update method is as follows:

\[
BACT_{i+1}^t = (BACT_i^t + 1) \mod (2 \times k) \quad (7)
\]

In any time slice \( t \), \( BACT_i^t \) is calculated according to the following formula, where \( 0 \leq i \leq 2 \times k - 1 \):

\[
BACT_i^t = (BACT_i^t + i) \mod (2 \times k) \quad (8)
\]

According to the formulas 6 and 8, only two \( AT \) blocks have \( ACT \) of 0 or \( k \) in any time slice. So only two \( AT \) blocks in any time slice need the state maintenance. To maintain the same number of \( AT \)s in each time slice, the number of \( AT \)s contained in each \( AT \) block should be as same as possible. Because \( 2^i \) is not necessarily a multiple of \( 2 \times k \), when we divide the \( 2^i \) \( AT \)s into \( 2 \times k \) groups, the number of \( AT \)s in each group may not be the same. A simple method is to make the number of \( AT \)s in the previous \( 2 \times k - 1 \) \( AT \) blocks the same, leaving all the differences in the last \( AT \) block, as shown in Figure 4. Let \( a = |B[i]| \) \( (0 \leq i \leq 2 \times k - 2) \) and \( b = |B[2 \times k - 1]| \), then \( a \) and \( b \) are calculated according to the following formula.

\[
a = \frac{2^c}{2 \times k - 1} \quad (9)
\]

\[
b = 2^c \mod (2 \times k - 1) \quad (10)
\]

According to Figure 4, the \( i \)-th \( AT \) in \( ATP \) is assigned to \( B[i/a] \). This method quickly calculates that an \( AT \) belongs to which block, but the difference between \( |B[2 \times k - 1]| \) and \( |B[0]| \) may be more than 1.

Another method is to distribute the differences evenly among the last several \( AT \) blocks, as shown in Figure 5. Let \( a' = 2^i/(2 \times k), b' = 2^i \mod (2 \times k) \). In the \( AT \) block partition method shown in Figure 5, for the former \( 2 \times k - b' \) \( AT \) blocks, each contains \( a' \) \( AT \)s; for the latter \( b' \) \( AT \) blocks, each contains \( a' + 1 \) \( AT \)s. At this time, the difference of the number of \( AT \)s in any two \( AT \) blocks will not exceed 1. But when using this partition method, mapping \( AT \) in \( ATP \) to \( AT \) block is more complicated. For the \( i \)-th \( AT \) in \( ATP \), if \( i < a' * (2 \times k - b' + 1) \), \( ATP[i] \) belongs to \( B[i/a]' \); if \( i \geq a' * (2 \times k - b' + 1) \), \( ATP[i] \) belongs to \( B[i + 2 \times k - b'] \).

The \( AT \) block partition methods shown in Figures 4 and 5 have their own advantages. Considering the simplicity of calculation, the \( AT \) block partition method shown in Figure 4 will be adopted in the following part.

For each \( aip \) in \( ANet \), \( VATE \) associates it with \( q \) \( AT \)s in \( ATP \). For \( aip \), each of its \( AT \) is called virtual asynchronous timestamp (\( VAT \)). Let \( VAT_{aip}^t \) denote the \( i \)-th \( VAT \) of \( aip \). \( VAT_{aip}^t \) is mapped to a physical \( AT \) in \( ATP \) by hash function \( H(aip, i) \), that is, \( VAT_{aip}^t = ATP[H(aip, i)] \), where \( H(aip, i) \) belongs to \( [0, 2^i - 1] \) and \( i \) belongs to \( [0, q - 1] \). The state of \( VAT \) is the state of its corresponding physical \( AT \) in \( ATP \). For example, if \( ATP[H(aip, i)] \) is active, then \( VAT_{aip}^t \) is active, otherwise, \( VAT_{aip}^t \) is inactive. Each opposite host \( bip \) of \( aip \) appearing in time slice \( t \) will select and set up a \( VAT \) of \( aip \). Let the hash function \( BH(bip) \) map \( bip \) randomly to an integer between 0 and \( q - 1 \). \( BH(bip) \) is used to determine which \( VAT \) is chosen by \( bip \), that is, \( VAT_{aip}^{BH(bip)} \). Setting \( VAT \) is ultimately the setting of physical \( AT \) in \( ATP \), that is, when \( bip \) appears, \( SetAT([ATP[H(aip, BH(bip))]]) \) operation is performed.

According to the attributes of \( AT \), at the end of time slice \( t \), \( VATE \) can not only know which \( AT \)s have been set in time slice \( t \), but also determine which \( AT \)s have been set in the previous \( k' - 1 \) time slices. In another word, at the end of time slice \( t \), \( VATE \) can report which \( AT \)s are active in time window \( W(t - k' + 1, k') \), where \( k' \) is a positive integer less than or equal to \( k \). For \( ATP \), the active \( AT \) can be regarded as the bit “1” in the algorithm GSE, and the inactive \( AT \) corresponds to the bit “0” in the GSE. According to the GSE algorithm, we can get the formula 8 for estimating the cardinality of \( aip \) in time window \( W(t - k' + 1, k') \) at the end of time slice \( t \). In time window \( W(t - k' + 1, k') \), \( Z_{k'}^{aip} \) represents the fraction of inactive \( AT \) in \( ATP \) and \( Z_{aip}^{k'} \) represents the fraction of inactive \( VAT \) in all \( VAT \) of \( aip \).

\[
|OP(aip, t - k' + 1, k')| = g * ln(Z_{aip}^{k'}) - g * ln(Z_{k'}^{aip}) \quad (11)
\]

Let \( IPair(t) \) represent all IP address pairs in time
slice $t$. By scanning the IP address pairs in each time slice, $VATE$ calculates the host cardinality under the sliding time window. The algorithm fully describes how $VATE$ estimates the cardinality of each host in $ANet$ under the sliding time window.

**Algorithm 4 $VATE$**

Input: $c,k,k'$

1. $a \leftarrow \frac{2^k}{k}$ /*there are $2^c$ AT in ATP, and there are most $k$ time slices in a time window.
2. for $at \in ATP$ do
3. $InitAT(at)$
4. end for
5. $BACT^0 \leftarrow 0$
6. $t \leftarrow 0$
7. while (1) do
8. for $<aip,bip> \in IPair(t)$ do
9. $vid \leftarrow BH(bip)$
10. $pid \leftarrow H(aip, vid)$
11. $bid \leftarrow \frac{vid}{\ln 2}^k$
12. $ACT^TP[pid] \leftarrow (BACT^0 + bid) \text{mod} (2 * k)$
13. $SetAT(ATP[pid])$
14. end for
15. for $aip \in ANet$ do
16. $Cardinality \leftarrow g \ast \ln (2^k) - g \ast \ln (Z_k^p)$
17. Output $aip$ and its $Cardinality$
18. end for
19. $BACT^0 \leftarrow (BACT^0 + 1) \text{mod} (2 * k)$
20. $t \leftarrow t + 1$
21. for $i \in [0, 2 * k - 1]$ do
22. $BACT_i^T \leftarrow (BACT^0 + i) \text{mod} (2 * k)$
23. if $(BACT_i^T == 0)$ or $(BACT_i^T == k)$
24. for $at \in B[i]$ do
25. $SetAT(at)$
26. end for
27. end if
28. end for
29. end while

$VATE$ initialises all AT's in ATP first. Then start scanning IP address pairs in each time slice. For each IP address pair in time slice $t$, lines [2] to [3] of the algorithm map it to an AT in ATP and set the AT. After scanning all IP address pairs in IPair(t), $VATE$ begins to calculate and output the cardinality of each $aip$ in $ANet$ in the time window $W(t - k' + 1, k')$ (from line [15] to [18]). After calculating the cardinality of all $aip$ in $ANet$, $VATE$ updates the $ACT$ of the first AT block in line [19] and then slides the window by a time slice. After the window sliding, $VATE$ maintains all ATs in the $AT$ block whose $BACT$ is 0 or $k$, as shown from line [21] to line [28] in algorithm [4].

$VATE$ can run continuously with only one initialisation. And $VATE$ is suitable for running on a parallel computing platform to process high-speed network data in real time. The next section describes how to deploy $VATE$ on the GPU.

4.2. Deploy $VATE$ on GPU

In high-speed networks, such as 40 Gb/s, millions of packets pass through the edges of the network every second. Scanning so many packets in real time requires a lot of computing resources. CPU is one of the most common computing components. Each core of CPU can handle complex tasks running different instructions. Although the computing core of CPU is powerful, its price is very high. If we want to use hundreds of CPU computing cores to process high-speed network data, we must adopt a cluster composed of multiple CPUs. The cost of the cluster will increase with the increase of scale. Graphics Processing Unit (GPU) is one of the most popular parallel computing platforms in recent years. A GPU chip contains hundreds to thousands of processing units, far more than that in the CPU. For tasks without data access conflicts and using the same instructions to process different data (single instruction multiple data streams, SIMD), GPU can achieve high speedup.

As can be seen from the algorithm, $VATE$ has three main steps: scanning IP address pairs, estimating cardinality and maintaining state. Each step uses the same steps to process different data (IP address pairs, host in $ANet$, AT in $ATP$), and there is no read-write conflict between different data processing. So $VATE$ is a SIMD program. When $VATE$ runs on GPU, $s$ threads can be used to process $s$ data at the same time. The value of $s$ is usually more than one thousand. Figure illustrates how $VATE$ runs on GPU.

$VATE$ first allocates $2^c \ast \text{ceil}(\log_2(2*(k+1)))$ bits on GPU graphical memory to store ATP and initialise each AT in the ATP. Then $VATE$ starts processing IP address pairs in each time slice. GPU is connected to the server through PIC bus. Since threads in the GPU can only access the graphics memory of the GPU, IP address pairs must be copied from the server to the graphics memory of the GPU. Network packets come in turn. It is not efficient to extract and copy IP address pair one by one, because each
data transmission session between server memory and GPU memory requires additional starting and ending operations. To improve efficiency, we allocate two buffers to store $r$ IP address pairs on the server side and GPU side respectively. For IPv4 addresses, an IP address pair takes 8 bytes, so the size of each IP address pair buffer is $r \times 8$ bytes. On the server, after extracting $r$ IP address pairs, VATE copies them to the buffer on the GPU. At the end of a time slice, if the IP address pairs buffer on the server is not full, it will be filled with the last IP address pair in the buffer, and then copied to the GPU. The larger the $r$, the more efficient the copying process is. However, the delay of scanning IP address pairs will also increase, because the scanning process of IP address pairs will not run until the IP address pairs on the GPU receive $r$ IP address pairs on the buffer. And under the sliding time window, IP address pairs are divided into time slices. If $r$ is greater than the number of IP address pairs in a time slice, the extra part will cause memory waste. The value of $r$ in this article is set to $2^{15}$.

When the IP address pair buffer on the GPU receives $r$ IP address pairs, VATE enters the “Scanning IP address pairs” process. This process starts $r$ GPU threads, each GPU thread reads an IP address pair in the buffer, and then processes the IP address pair according to the steps from line 9 to 13 of algorithm 4. At the end of a time slice, the “estimating cardinality” process begins. In this process, VATE first divides the IP addresses in $A_Net$ into groups and each group contains $s$ IP addresses. The number of IP addresses in the last group may be less than $s$, and the insufficient part can be filled with 0. Then VATE calculates the cardinality of IP addresses in $A_Net$ group by group on the GPU. For each IP group, VATE starts $s$ threads on the GPU. Each thread calculates the cardinality of an IP address by the steps from line 16 to 17 of algorithm 4.

After finishing estimating cardinality process, VATE slides the time window, updates $BACT^0$, and then begins to maintain the state of $AT$s in two blocks. VATE divides $AT$ in these two $AT$ blocks into groups, and each group contains $s$ ATs (the number of AT in the last group may smaller than $s$, and the deficiencies will not be dealt with). Then VATE maintains the state of $AT$ group by group on GPU. For every group, VATE launches $s$ threads, and each thread preserves the state of an AT (apply $\text{PreserveAT}()$ operation).

After state-maintenance, VATE starts to scan IP address pairs in the next time slice. In this way, VATE continuously estimates the cardinality of each IP address under the sliding time window on GPU in real time.

As can be seen from the Figure 6, the three main processes of VATE are executed in sequence. Each process handles multiple data concurrently on the GPU. And each process does not read and write the same memory area at the same time. So VATE does not have read-write conflicts when running in parallel. Only in the “scanning IP address pairs” process, multiple threads may write an AT ($\text{SetAT}()$, set the AT value to 0) at the same time. However, various threads set the same AT, regardless of the order in which these threads write the AT, the result is the same (AT value is set to 0). So VATE does not have write conflicts when running in parallel.

When running on GPU, even if $ATP$ contains up to $2^{32}$ ATs, VATE can maintain the state of each AT in real time and estimate the cardinality of each host under the sliding time window. In the next section, we will demonstrate the advantages of the VATE algorithm in memory usage and state-maintenance through experiments on real high-speed network traffic.
5. Experiments

In order to evaluate the performance of VATE, we downloaded traffic data for 1 hour from a 10 Gb/s core network (Caida\cite{30}) and a 40 Gb/s core network (IPtas\cite{31}) respectively. Caida’s traffic was the network packets between Seattle and Chicago from 13:00 to 14:00 on February 19, 2015. IPtas traffic was the network packets between CERNET Jiangsu node and other networks from 13:00 to 14:00 on October 23, 2017. In these experiments, for Caida traffic, the hosts in Seattle compose ANet, the hosts in Chicago compose BNet; for IPtas traffic, hosts in CERNET Jiangsu node make up ANet, hosts in other networks make up BNet.

Table 7 lists the main characteristics of these two sets of traffics in a five-minute time window. In table 7, “#AIP” denotes the number of hosts in ANet, “#BIP” denotes the number of hosts in BNet, and “#Flow” denotes the number of different IP address pairs. “Speed(kpps)” denotes the speed of packets with unit one thousand packets per second. “#IPc100” denotes the number of hosts whose cardinality is greater than 100, and “%IPc100” denotes the percentage of hosts whose cardinality is greater than 100.

As can be seen from table 7, hosts with cardinalities greater than 100 account for only a small part of all hosts. In practical applications, researchers and managers are interested in those hosts with high cardinalities. Moreover, cardinality estimation for a large number of small-cardinality hosts will take too much computing time and reduce the efficiency of an algorithm. So this experiment only focuses on those hosts whose cardinalities are greater than or equal to 100.

In our experiment, we set the size of a time slice to 1 second, and the first time slice lasting from 13:00:00 to 13:00:01 is regarded as time slice 0. \( k' \) is set to 300, that is, to calculate the cardinality in 5 minutes. In a five-minute window, the cardinality of most hosts is less than 5000. For example, Figure 8 shows the distribution of cardinalities of these two traffic in \( W(600, 300) \).

In the time window \( W(600, 300) \), there are only 10 and 72 hosts respectively whose cardinalities are greater than or equal to 5000 in Caida 2015_02_19 and IPtas 2017_10_23. Cardinalities of most hosts (99.733% in Caida 2015_02_19 and 98.816% in IPtas 2017_10_23) are less than 5000. VATE is based on GSE algorithm. According to GSE, the larger the number of virtual counters \( g \), the larger cardinality that can be estimated by VATE. But the larger the \( g \), the more time it takes to estimate the cardinality because VATE needs to check the states of more virtual counters. To compare the effects of the number of physical counters (for VATE, the number of counters is the number of ATs in ATP, i.e. \( 2^k \)) and the number of virtual counters\( (g) \) on cardinality estimation, we tested estimation accuracy of VATE under different parameters. Figure 9 and 10 show the cardinalities of Caida 2015_02_19 and IPtas 2017_10_23 estimated by VATE under time window W(600,300). In these two graphs, the abscissa represents the actual cardinality, and the ordinate is the cardinality estimated by VATE. When the points in the figure are closer to the oblique line (the oblique line composed of the points whose estimated cardinality equals the actual cardinality), the accuracy of the estimated result is higher. When the point in the graph is below the right of the oblique line, it means that the cardinality is underestimated; when the point is above the left of the oblique line, it means that the cardinality is overestimated.

As can be seen from Figure 9 and 10, when the number of counters remains unchanged \( (2^{23}, 2^{26} \text{ or } 2^{29}) \) and \( g \) increases from 1024 to 4096, the number of hosts whose cardinality cannot be approximated by VATE is also decreases. When \( g = 4096 \), for Caida 2015_02_19, the cardinality of all hosts in the time window W(600,300) can be approximately
estimated. When the value of $g$ is constant, the larger the number of counters, the closer the points are to the oblique line, that is, the closer the estimated value is to the actual cardinality. Therefore, to improve the efficiency of VATE, $g$ should be determined according to these large cardinalities in the window; in order to improve the accuracy of the algorithm, more counters should be stored in the counter pool. But the more counters there are, the more memory the counter pool occupies and the more time it takes to maintain the state of the counter.

**DR** is a counter under the sliding time window with optimal memory and small state-maintenance operation. Similar to VATE, when the counter in the counter pool is DR, the VDRE algorithm can be obtained (the setting of AT, state-maintenance and state checking become the corresponding operation of DR). Both VATE and VDRE are based on GSE. When the number of counters in the counter pool is the same, and the mapping method of IP address to virtual counters and virtual counters to physical counters in the counter pool are the same, VATE and VDRE will get the same estimating value. The difference lies in memory usage and state-maintaining time. When $k'$ is set to 300, each DR takes 9 bits (ceil(log$_2$(k' + 1))) and each AT takes 10 bits (ceil(log$_2$(2 * k' + 1))). Although AT contains one bit more than DR, AT has less state-maintaining time. The running time of VATE and VDRE can be divided into three parts: packet scanning time (ST), cardinality estimation time (ET) and counter state-maintaining time (PT). Different running platforms influence these three kinds of running time. We use three different Nvidia GPUs to test the running time of these algorithms: GTX650 with 1 GB graphic memory, GTX950 with 4 GB graphic memory and
TitanXp with 12 GB graphic memory. The computing power and graphics memory of these three GPUs are improved in sequence. The three GPUs are represented by GTX650-1GB, GTX950-4GB and TitanXP-12GB, respectively. Tables 3 and 4 show the average running time(avgST, avgET and avgPT) of VDRE and VATE when estimating the cardinalities of these hosts in Caida 2015_02_19 on the three GPU platforms. Among them, $g$ is set to 1024, and the number of counters increases from $2^{23}$ to $2^{32}$. #Cnt.($lg2$) is the exponent of the number of counters when the base number is 2.

In tables 3 and 4, NULL indicates that the counter pool is larger than the GPU memory and cannot run on the GPU. As can be seen from these two tables, the three running times of VDRE and VATE are greatly reduced with the improvement of GPU computing power. According to algorithm 4, ST and ET are not affected by the number of counters.

![Figure 10: Cardinality estimation of IPTas 2017_10_23](image)

**Table 3: Time consumption of VDRE on different GPU**

| $g$ (cnt.) | GTX650-1GB | GTX950-4GB | TitanXP-12GB |
|-----------|------------|------------|--------------|
| $2^{23}$  | 2.252      | 23.273     | 23.723       |
| $2^{24}$  | 9.955      | 89.039     | 89.039       |
| $2^{25}$  | 51.461     | 389.688    | 389.688      |
| $2^{26}$  | 137.632    | 842.977    | 842.977      |

**Table 4: Time consumption of VATE on different GPU**

| $g$ (cnt.) | GTX650-1GB | GTX950-4GB | TitanXP-12GB |
|-----------|------------|------------|--------------|
| $2^{23}$  | 2.252      | 23.273     | 23.723       |
| $2^{24}$  | 9.955      | 89.039     | 89.039       |
| $2^{25}$  | 51.461     | 389.688    | 389.688      |
| $2^{26}$  | 137.632    | 842.977    | 842.977      |
With the number of counters in the counter pool, there is little change of ST or ET. The difference between counters are contained in the counter pool, there is on the same GPU, no matter how many counters are contained in the counter pool, there is little change of ST or ET. The difference between VDRE and VATE is reflected in PT. PT increases with the number of counters in the counter pool, as shown in Figure 11. For VDRE, this upward trend is more obvious.

On the same GPU, when the counter pool contains the same number of counters, VATE’s PT is significantly less than VDRE’s (VATE’s PT accounts for only 25% to 0.25% of VDRE’s PT). On GTX650-1GB, when the number of counters is \(2^{28}\), the PT of VDRE is as high as 1296 milliseconds, and the sum of three running times is 1447 milliseconds. For the algorithm under the sliding time window, the total running time in each time slice should not exceed the length of a time slice. Otherwise the algorithm cannot run in real time. In this experiment, the length of the time slice is 1 second, while when the number of counters is \(2^{28}\) on GTX650-1GB, the running time of processing data in each time slice is 1.4 times of the length of time slice. TitanXP-12GB has more computing power than GTX650-1GB, which can effectively reduce the running time. But the price of TitanXP-12GB is also higher than that of GTX650-1GB, and when the number of counters reaches \(2^{32}\), even on TitanXP-12GB, VDRE cannot run in real time, because its PT exceeds one second. So it is not the GPU memory that limits the accuracy of VDRE, but the state-maintaining time of VDRE. VATE solves this problem very well. As can be seen from the table, the PT of VATE does not exceed 14 milliseconds on any GPU.

On different GPUs, when the number of counters takes different values, the difference between VATE’s PT and VDRE’s PT is also changed. To measure this difference, we define the PT speedup as the ratio of VDRE’s PT to VATE’s PT on the same GPU with the same number of counters. Table 5 shows the change of PT speedup with different counter number on different platforms. As can be seen from the table, VATE can achieve better PT speedup for low-computing capacity GPUs. On the same GPU, the greater the number of counters, the greater the PT speedup. On TitanXP-12GB with \(2^{32}\) counters, VATE even acquires a PT speedup as high as 409.

When the number of counters is constant, the PT in each time slice is not necessarily the same on the same GPU. PT of each time slice will fluctuate within a certain range. Figure 12 shows the PT of each time slice on different GPUs when the number of counters is \(2^{29}\). As can be seen from Figure 12, PT changes greatly at the beginning, and then gradually stabilizes. The higher the computing power of GPU, the lower the PT variance of different time slices. On the same GPU, the variance of VATE’s PT is lower than VDRE’s PT. Therefore, the fluctuation of VATE’s PT is small, and it runs smoothly. It can also be seen from the figure that in each time slice, the PT of VATE is smaller than that of VDRE. Regardless of the computing capacity of GPU, VATE can make full use of the graphics memory of GPU to estimate cardinalities in real time with higher accuracy.

6. Conclusion

The AT proposed in this paper has the advantages of less memory consumption and low state-maintaining time. For sliding time window with \(k\) time slices at most, each AT only takes up \(\text{ceil}(\log_2(2 + k + 1))\) bits, and the time complexity of maintaining AT state is only \(O(1/k)\). Based on AT, this paper designs a new multi-hosts cardinalities estimation algorithm VATE. VATE allocates a virtual AT vector for each host to estimate the cardinality. VATE improves memory utilisation by sharing ATs in an AT pool. The higher the number of ATs in ATP, the higher the accuracy of VATE. Because AT occupies less memory and requires smaller state-maintaining time, VATE can use more counters than existing algorithms to estimate cardinalities of different hosts under sliding time windows in real time with higher accuracy. VATE is also a parallel algorithm that can be deployed on GPU. With the parallel processing capacity of GPU, VATE can estimate the cardinalities of hosts in a 40 Gb/s high-speed network with the length of time slice as small as 1 second. In the further work, we will deploy VATE in CERNET network to provide critical host attribute for network security and management.
Acknowledgment

The authors would like to thank anonymous reviewers. The research work leading to this article is supported by the National Natural Science Foundation of China under Grant No. 61602114

[1] C. C. Lo, C. C. Huang, J. Ku, A cooperative intrusion detection system framework for cloud computing networks, in: 2010 39th International Conference on Parallel Processing Workshops, 2010, pp. 280–284. doi:10.1109/ICPPW.2010.46

[2] S. Karthik, J. J. Shah, Analysis of simulation of ddos attack in cloud, in: International Conference on Information Communication and Embedded Systems (ICICES2014), 2014, pp. 1–5. doi:10.1109/ICICES.2014.7033841

[3] R. E. Harang, P. Mell, Evasion-resistant network
Figure 12: Preserving time in different time slices

[Image of four graphs showing time slices for VDBE and VATE on GTX60-1GB, GTX50-4GB, and TitanXP-12GB]

[1] M. Bonomi, S. Krishnamurthy, M. Medvidovic, Efficient and scalable secure data publishing, in: Proceedings of the 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM ’13, ACM, New York, NY, USA, 2013, pp. 325–336. doi:10.1145/2484854.2484883
URL http://dx.doi.org/10.1145/2484854.2484883

[2] N. Duffield, C. Lund, M. Thorup, Estimating flow distributions from sampled flow statistics, in: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM ’03, ACM, New York, NY, USA, 2003, pp. 325–336. doi:10.1145/863955.863992
URL http://doi.acm.org/10.1145/863955.863992

[3] S. Venkataraman, D. Song, P. B. Gibbons, A. Blum, New streaming algorithms for fast detection of super-spreaders, in: in Proceedings of Network and Distributed System Security Symposium (NDSS, 2005, pp. 149–166.
[4] X. Zhou, W. Liu, Z. Li, W. Gao, A continuous virtual vector-based algorithm for measuring cardinality di-
erating data transfers for distributed gpu parallel

programs, Journal of Parallel and Distributed
Computing 73 (12) (2013) 1649 – 1660, hetero-
genecity in Parallel and Distributed Computing.
doi:https://doi.org/10.1016/j.jpdc.2013.07.022
URL http://www.sciencedirect.com/science/
article/pii/S0743731513001603

[30] C. for Applied Internet Data Analysis, The caida
anonymized internet traces, http://www.caida.org/
data/passive online;accessed 2017 (2017).

[31] CERNET, China education and research net-
work, http://iptas.edu.cn/src/system.php on-
line;accessed 2017 (2017).