Cluster Comput (2018) 21:1189–1202
https://doi.org/10.1007/s10586-017-0941-1

SearchaStore: fast and secure searchable cloud services

Wai-Kong Lee[1] · **Raphael C.-W. Phan**[2] · **Geong-Sen Poh**[3] · **Bok-Min Goi**[4]

Received: 14 June 2016 / Accepted: 22 May 2017 / Published online: 4 July 2017

Abstract The emergence of Cloud Computing is revolutionizing the way we store, query, analyze and consume data, which also bring forward other development that fundamentally changed our life style. For example, Industry 4.0 and Internet of Things (IoT) can improve the quality of manufacturing and many aspects in our daily life; both of them rely heavily on the cloud computing platform to develop. Central to this paradigm shift is the need to keep any common data, often held at remote outsourced locations and usually to be accessed by different authorized parties, secure from being leaked to unauthorized entities. When using the cloud services, consumer may want to encrypt sensitive data before uploading it to the cloud, but this will also eliminate the possibility to search the data efficiently in the cloud storage. A more practical solution to this is to employ a searchable encryption scheme in the cloud storage, so that user can query the encrypted data efficiently without revealing the sensitive data to the service provider. Besides the security and search features, performance of searchable encryption schemes is also very important when it comes to practical applications. In this paper, we propose several techniques to accelerate the search performance of encrypted data stored on the cloud. Notably, our techniques include massively parallel file encryption, multi-array keyword red black tree (KRBT) implementation, batched keyword search and enhanced parallel search in KRBT. To the best of our knowledge, SearchaStore is the first work that attempts to accelerate searchable encryption using GPU technology.

Keywords Cloud service · Secure outsourcing · Keyword Red Black Tree · Searchable symmetric encryption

✉ Wai-Kong Lee
dexter6855@hotmail.com; wklee@utar.edu.my

Raphael C.-W. Phan
raphael@mmu.edu.my

Geong-Sen Poh
gspoh@mimos.my

Bok-Min Goi
goibm@utar.edu.my

[1] Faculty of Information and Communication Technology, Universiti Tunku Abdul Rahman, Kampar, Malaysia

[2] Faculty of Engineering, Multimedia University, Cyberjaya, Malaysia

[3] MIMOS Berhad, Kuala Lumpur, Malaysia

[4] Lee Kong Chian Faculty of Engineering and Science, Universiti Tunku Abdul Rahman, Sungai Long, Malaysia

1 Introduction

1.1 Problem statement

The advancement of Cloud Computing is producing significant impact in our daily life in the past decade. Some of the influential examples of this advanced technology are health services [1], cloud manufacturing [2], asset tracking [3], security [4], personalized entertainment [5] and power grid monitoring [6]. With such massive amounts of heterogeneous data often comes the need for more centralized storage hosted by organisations which have more capabilities and resources for storage and computation, something not all organisations can afford to set up and maintain on their own. Therefore, outsourcing data to third party cloud storage systems to provide real time and ubiquitous access is one of the current industrial trends. However, the collected data and the analysis generated subsequently are sensitive information for many organizations, which should be protected from unauthorized access, including the cloud storage service provider itself. In view of this, security plays a vital role [7] in the

next generation of the Cloud Computing, to protect the outsourced data from various malicious attacks internally and externally. This is crucial, such that industrial secrets can be safeguarded from competitors, or that the personal customer related information is kept private and confidential.

One of the effective solutions to protect the sensitive data in the cloud services is by encryption. However, with this the user can no longer search the data in the cloud effectively, since the encrypted data is already scrambled and randomized. In contrast, searchable encryption scheme can be employed to secure the sensitive data in the cloud and provide the convenience to search efficiently. Hence, the performance of searchable encryption is of utmost importance for practical applications. This is true especially for cloud storage servers which potentially service numerous clients and therefore need to process and respond to queries without unnecessary delay. In this paper, we present SearchaStore, a GPU accelerated searchable encryption solution that is capable to achieve impressive search performance with 1.4 ms in a 535 MB dataset for single keyword search.

1.2 Searchable encryption

Searchable encryption schemes can enable the search ability to be provided for cloud-based data stores without compromising the confidentiality of the stored information, which potentially could include industrial secrets or confidential customer data. Some schemes such as searchable symmetric encryption (SSE) are designed based on symmetric primitives [10–23] for data encryption and some specific data structures to hold the encrypted indexes.

There are some searchable encryption schemes designed based on public-key mechanisms [24], which rely heavily on computationally expensive number theory. Searchable encryption schemes can also be constructed based on general primitives such as oblivious RAM [25] and homomorphic encryption [26]. Compared to other constructs (based on asymmetric primitives, ORAM or homomorphic encryption), symmetric primitives are usually more light weight and suitable for parallel implementation.

On the other hand, the GPU (graphics processing unit) with massively parallel architecture has recently emerged rapidly as an effective accelerator for accelerating many computationally extensive algorithms. In view of the huge computational power of the GPU, we are interested to investigate the potential of improving SSE performance by utilizing this cost-effective and energy-efficient platform for secure outsourced storage.

Porting these SSE constructions to the GPU platform is non-trivial as the GPU exhibits memory structure and execution model that are very different from the CPU platform. To harvest the processing power of the GPU for high speed SSE, substantial work is needed to design the implementation carefully for optimized performance. Successful application of GPU technology to accelerate SSE could greatly improve the practicality of searchable encryption for cloud storage, which is also the main motivation of this work. From our experimental results, SearchaStore, the optimized GPU implementation of SSE (based on Kamara et al. [21]) is able to achieve search efficiency of 16.9x greater than the multi-core CPU.

1.3 Overview of searchable symmetric encryption

In this section, we introduce a generic system model of SSE and the development of SSE to date. In general, SSE environment consists of a group of users and a server (or storage provider). The user has sensitive information that he wishes to store in a remote storage. He uses a SSE scheme to encrypt the data and build encrypted index structure, which is then stored in the remote storage. The encrypted index structure can be used to retrieve sensitive data securely. It may be constructed based on the following elements:

(i) A keyword index list.
(ii) A list of document/pointer to document matching the keyword indexes.
(iii) A masking mechanism to preserve the privacy of the keyword (or document) index and list.

In order to search the encrypted data, an authorized user generates a search token based on SSE scheme. The server upon receiving this search token, retrieves the matching encrypted indices from the encrypted index structure and the encrypted information based on the SSE protocol. The overall system is illustrated in Fig. 1.

SSE scheme was first introduced by Song et al. [10] by dividing a document into fixed or variable-length words. These words are encrypted and masked'? by bitwise exclusive-or (XOR) with a pseudorandom bit strings. In order to search for a document containing certain keyword, the keyword is encrypted and used as a search token. The remote storage server searches matching documents using

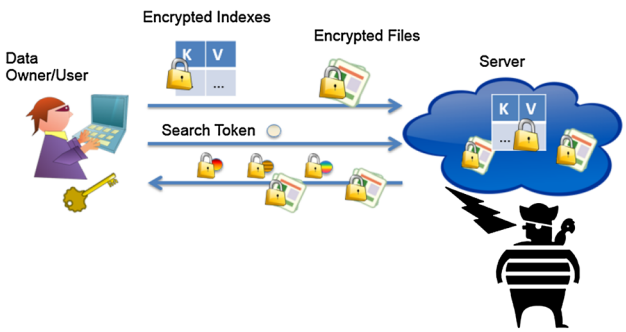


Fig. 1 Generic system model of SSE

the encrypted keyword by sequentially scanning through all documents. Due to the inefficiency in searching, index-based search mechanisms were also suggested by them to overcome this limitation. Later on, Goh et al. [11] proposed improved secure indexes based on Bloom filters.

More recent research on SSE scheme caters for improved security notion [12], arbitrarily-structured data [13], scalable updates [14,15], multi-user [16], privacy preservation [17] and conjunctive Boolean search to improve on all previous schemes that only cater for single keyword search, as proposed by Cash et al. [18] and Moataz et al. [19]. Ranked keyword search schemes over cloud data is proposed by Yu et al. [20].

Recently, Kamara et al. [21] proposed to utilize Keyword Red Black Tree (KRBT) for storing encrypted index, which eventually allow keyword search to be completed within O(r log n) in sequential time and O(r/p log n) in parallel time. But the authors did not provide parallel implementation performance, as also mentioned by Cash et al. [22]. The SSE scheme proposed by Kamara et al. [21] can be generalized to use other balanced tree (e.g. B-tree) for handling very large dataset whereby indexes cannot fit into main memory. The flexibility of this SSE scheme convinced us that it has potential for high speed SSE that support single keyword search.

Cash et al. [22] presented an interesting dynamic SSE scheme that scales well for very large dataset with O(r/p) parallel search time, where r is the number of files containing keyword and p is the number of processors. The proposed scheme takes into consideration low-level space utilization and I/O parallelism, which are not addressed in previous research work.

Although there are many SSE schemes being proposed recently, none of them had actually discussed the possibility to utilize accelerator (e.g. GPU and FPGA) to speed up the encryption/decryption and search process. In order to take advantage of the raw processing power of accelerators, the designed SSE schemes has to be parallelizable and uses data structures that are optimized for memory access specific to the hardware platform. The SSE scheme proposed by Cash et al. [22] uses a data structure based on open-addressed hash table with a specific array construct for multi-core CPU implementation, in which adopting it to GPU is also a non-trivial task and we intend to investigate this as a future work.

In this paper, we focus on designing efficient implementation of KRBT SSE [21] in GPU, and defer the generalization to other tree structures as future work. It is more straightforward to extend the techniques developed in this work to other balanced tree in GPU, compared to the specific index structures proposed by Cash et al. [22] and Naveed et al. [15]. Our work may benefit other SSE constructions that utilized balanced tree for efficient search [23], and subsequently benefit the practical use of SSE in the consumer applications. We denote the GPU based KRBT SSE [21] as SearchaStore in subsequent discussion. KRBT relies heavily on symmetric primitives and Red Black Tree (RBT) data structure, which can run efficiently in parallel platform. However, naive implementation of red-black-tree traversal in GPU usually yield poor performance, due to a few reasons:

(i) Conventionally, RBT is implemented using pointers and structure. Traversal from one node to another requires multi levels pointer indirection, which leads to massively increased memory latency (un-coalesced global memory access) in GPU.
(ii) Traditionally, traversal in RBT is implemented using stack and recursive function call. It requires a stack to record tree nodes being traversed so that we can return to it later. However, recursive function call is only supported in GPU with CUDA compute capability 3.5 and above (Dynamic Parallelism in CUDA terminology). This feature only supports maximum 32 levels of recursion, so it may not be suitable to use Dynamic Parallelism when the KRBT level grows beyond 32.
(iii) To utilize the GPU as accelerator, we need to copy the entire tree structure (in our case, the KRBT) from CPU memory to GPU memory. The search traversal is performed entirely in GPU to leverage the massively parallel architecture for high speed search. However, copying RBT implemented in nested pointers form is a challenging task, as we need to traverse each node to uncover the pointers of their child nodes, which is requiring special design effort.

Several works on accelerating tree traversal in GPU exists in the literature. Hughes et al. [27] proposed kd-jump to allow implicit kd tree traversal can be done stacklessly. Kaczmarski et al. [28] suggested to store key-value pairs in separate arrays for fast implementation of B+-tree in GPU. On the other hand, Kim et al. [29] presented binary tree search techniques that are sensitive to target architecture (CPU and GPU). KRBT is not sorted according to any order, so the techniques presented in [28] and [29] are not directly applicable to KRBT. Moreover, the structure and traversal algorithm of KRBT is inherently different from kd tree, which in turn motivates us to design efficient tree traversal for KRBT in GPU. As far as we are concern, there is no published prior work on optimized implementation of RBT in GPU platform.

We adopt the idea of using separate arrays for B+-tree implementation in GPU [28] and construct our own data structure for KRBT. The key difference between our work and the one presented in [28] is that KRBT is not sorted based on any order. Besides, the search process in B+-tree returns only one node that contain the desired value, but KRBT search process may traverse all possible nodes and return multiple results. This highlights that significant effort

is required to leverage the techniques developed from prior works to accelerate KRBT in GPU.

1.4 GPU for cryptography

Recently, GPU emerged as one of the most promising platforms in accelerating many algorithms, including sparse matrix solver [30], artificial intelligence [31] and consumer application [32]. The use of GPU in accelerating the implementation of advanced cryptographic algorithms is also becoming popular in recent years. GPU is proposed as an accelerator for symmetric [33] and asymmetric [34,35] cryptographic primitives, which enjoy great success in boosting the throughput of these algorithms. GPU can be used to accelerate the index setup process for KRBT, which mainly involves parallel implementation of symmetric primitives (block ciphers). GPU can also accelerate the keyword search process when the search traffic is huge. To fully utilize the processing power of GPU, we need to carefully consider the smart utilization of deep memory architecture in GPU, efficient kernel code and the threads occupancy in GPU.

1.5 Contributions and limitations

In this paper, we focus on developing fast implementation techniques for KRBT SSE in GPU platform. To the best of our knowledge, we are not aware of any existing SSE schemes designed based on GPU platform, so we only benchmark our work against existing solutions in multi-core CPU. Our contributions can be summarized as below:

(i) Accelerate the index setup phase by utilizing GPU for parallel encryption. Generally, the number of keyword is much lesser compare to the total files in dataset, so we only focus on designing parallel file encryption scheme. We proposed to operate block cipher in counter mode (CTR) and perform file encryption using GPU. As file encryption is the most time consuming part in index setup phase, parallel encryption can greatly reduce the time required to setup indexes for a large dataset.

(ii) Propose a new data structure to represent KRBT for efficient GPU search traversal without recursive function call. As mentioned in earlier section, RBT implemented in nested pointers form is not suitable for implementation in GPU, due to architectural limitations in GPU. Our new multi-array data structure is able to reduce the uncoalesced memory accesses to GPU's global memory.

(iii) Propose a new strategy to search batch of keywords in KRBT concurrently using GPU. Kamara et al. proposed parallel search algorithm in their original work [21], but it does not fully utilize the parallel structure of KRBT. We proposed to divide the KRBT into multiple sub-tree, then assign each sub-tree to a thread in GPU. Multiple threads can traverse the KRBT sub-tree concurrently as they are not dependent on each other. In addition to it, we also utilize the GPU to perform batch keyword search, which is an extension to the original search algorithm proposed by Kamara et al [21]. This is especially useful when there are many users attempt to perform secure search on the same dataset [12] hosted in the cloud storage. These improvements can only be done after implementing the KRBT in multi-array structure.

(iv) Design a mechanism to automatically select the suitable KRBT search accelerator (multi-core CPU or GPU) based on search traffic. Utilizing GPU as an effective co-processor does incurs some overhead, so we design a mechanism to select an initial threshold level based on the hardware resource available, then schedule CPU or GPU to perform the keyword search based on the traffic. Our proposed techniques only focus on the static construction at the moment. We do not implement the dynamic update function offered by the KRBT SSE proposed in [21] as we believe that improving search efficiency of current SSE schemes are more important to practical use.

2 Keyword red black tree

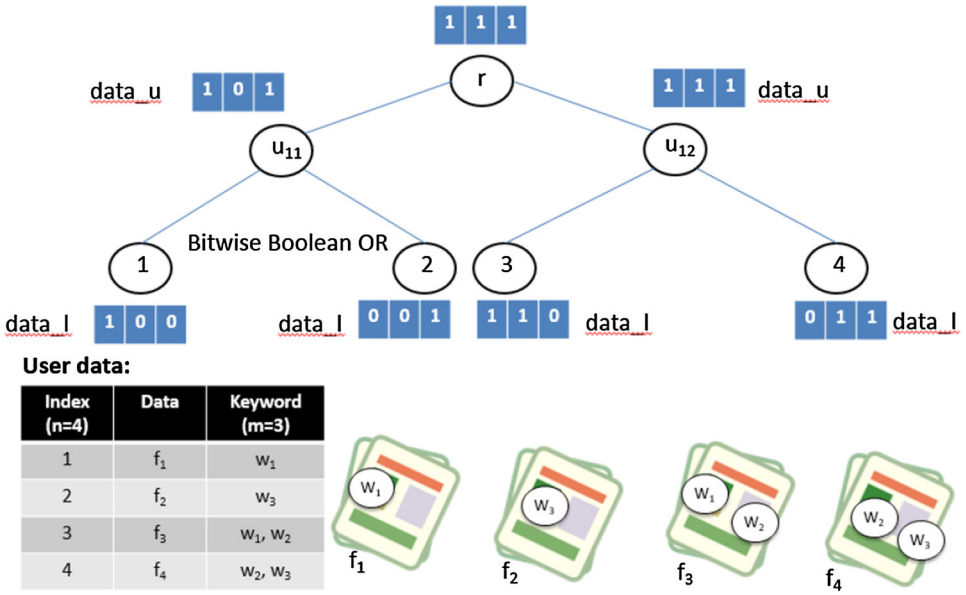
RBT is a self-balancing binary search tree commonly used in computer science. Since RBT is a balanced tree, the insertion, search and deletion guarantee to complete in O(log n) time, with n represent the number of elements in tree. Kamara et al. [21] introduced the concept of KRBT that build around the conventional RBT. The proposed KRBT SSE is divided into four phases: build index, encrypt index, search and update (insert or delete).

We first briefly describe the build index process based on KRBT, follow by the index encryption and search process. The update process is not described in this paper as we are only focusing on the static version of KRBT SSE. Interested reader can refer to the original work by Kamara et al [21].

KRBT SSE make use of a private-key encryption scheme consists of three algorithms $\epsilon = (Gen, Enc, Dec)$. Specifically, $Gen(1^k; r)$ is a probabilistic polynomial-time (PPT) algorithm that takes a security parameter k and randomness r and returns a secret key K. $Enc(K, msg)$ is PPT algorithm that takes a key K and a message msg and returns a ciphertext c. $Dec(K, c)$ is a deterministic algorithm that takes a key K and a ciphertext c and returns msg if K was the correct key that produces c.

KRBT is constructed from a set of files $\mathbf{f} = (f_{i_1}, ..., f_{i_n})$ with identifier $\mathbf{i} = (i_1, ..., i_n)$, and a universe of keyword $\mathbf{w} = (w_1, ..., w_m)$. The data structure is constructed in three steps:

Fig. 2 Keyword red black tree



i) With a set of files **f**, build a KRBT, **T** on top of the identifier **i**. The leaves of **T** stores the pointers to the appropriate files, and the files **f** are stored separately (e.g. hard disk, SSD).
ii) Construct an m-bit wide vector $data_u$ in each internal node u. The i-th item of $data_u$ represents the keyword w_i for $i = 1, ..., m$. Specifically, if $data_u[i] = 1$, then one or more path from u to the leaves store some identifier j, with f_j contain keyword w_i.
iii) For every leaf l storing identifier j, set $data_l[i] = 1$ if the file f_j contains keyword w_i. Proceed a level up the tree, with u be the internal node of tree **T**, with left child v and right child z. $data_u$ is constructed recursively up to the root, by using the formula below:

$$data_u = data_v + data_z \tag{1}$$

The operation + is bitwise boolean OR operation.

With this construction, the KRBT can be searched in this way. Assume that we are searching a keyword with position i in the keyword universe **w**. Starting from the root node, we check $data_u[i] = 1$ to examine if the children (left, right or both) contain the keyword we are looking for. When the traversal is over, return all the leaves that were reached.

Figure 2 shows an example of KRBT with only four files and three keywords. Assume that we are searching for keyword w_2, we first traverse through root node and found that $data_u[2] = 1$, indicating both paths *may* contain this keyword. We then traverse to u_{11} and found that $data_u[2] = 0$, so we stop traversal for this sub-tree. We continue our traversal to u_{12} and found that $data_u[2] = 1$, so we proceed to both of the children nodes. Finally, the keyword w_2 is found in f_3 and f_4.

Let $\mathbf{f} = (f_{i_1}, ..., f_{i_n})$ be the set of files and **w** be the universe of keyword. Three cryptographic primitives are used to encrypt the KRBT:

1) pseudo-random function $G : \{0, 1\}^k \times \{w_1, ..., w_m\} \rightarrow \{0, 1\}^k$;
2) Another pseudo-random function $P : \{0, 1\}^k \times \{w_1, ..., w_m\} \rightarrow \{0, 1\}^k$;
3) A random oracle $H : \{0, 1\}^k \times \{0, 1\} \rightarrow \{0, 1\}$;

The processes to build index are described in detail in the following paragraphs.

Algorithm $K \leftarrow Gen(1^k)$: This is a process to generate keys. First generate three random k-bit strings K_1, K_2 and r. Then instantiate a private key CPA-secure encryption algorithm ϵ. Generate K_3 by calling $K_3 \leftarrow \epsilon.Gen(1^k; r)$. K_3 is used for encrypting files. Set $K{:=}(K_1, K_2, K_3)$.

Algorithm $(\gamma, \mathbf{c}) \leftarrow Enc(K, \delta, f)$: δ be the KRBT from the build index process.

1) Instantiate another private key CPA-secure encryption algorithm R. For each keyword w_i derive a secret key $SK_i = R.Gen(1^k; G_{K_2}(w_i))$, for $i = 1,...,m$.
2) For $1 \leq j \leq n$, Encrypt the files to generate ciphertexts $c_{i_j} \leftarrow \epsilon.Enc(K_3, f_{i_j})$. Store this ciphertexts **c** on disk and delete **f**.
3) For every node v in KRBT, T that has identifier **id**(v), do the following:

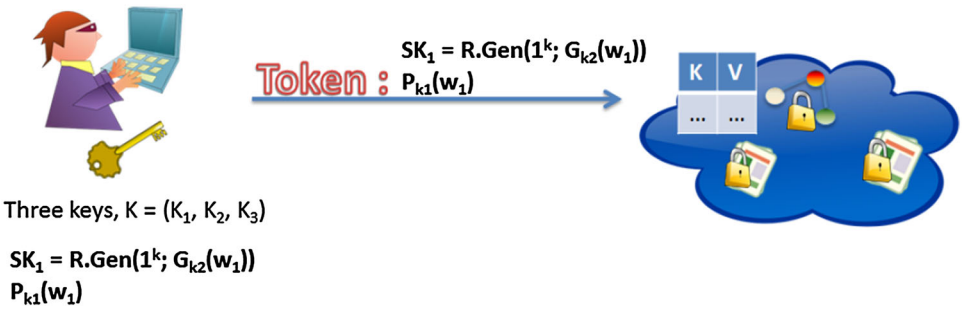


Fig. 3 Search token for keyword w_1.

- Construct two keyword hash tables, λ_{1v} and λ_{0v}. Store both λ_{1v} and λ_{0v} at node v;
- For $i = 1,...m$, compute a bit value b = $H(P_{K_1}(w_i), id(v))$, where $data_v$ is the vector at node v in T. Set $\lambda_{bv}[P_{K_1}(w_i)] \leftarrow R.Enc(SK_i, data_v[i])$ with the b generated. This process is equivalent to picking one of the two hash tables to store the actual entry for keyword w_i, based on the bit value b.
- Store a random value at $\lambda_{|1-b|v}[P_{K_1}(w_i)]$

4) Delete vector $data_v$.
5) Output $\gamma :=$ T and $\mathbf{c} := (c_i, ..., c_{i_n})$

The encrypted index can be stored in cloud server for search and update.

Algorithm $\tau_s \leftarrow SrchToken(K, w_i)$: Output the secret key $SK_i = R.Gen(1^k; G_{K_2}(w_i))$ and generate search token $\tau_s = (P_{K_1}(w_i), SK_i)$;

Algorithm $i_w \leftarrow Search(\gamma, \mathbf{c}, \tau_s)$: Parse τ_s as (τ_1, τ_2). Search the KRBT from root node. Let v and z be the left and right child of a node u. Algorithm search(u) is recursively called to perform the following steps:

1) Output a bit b = $H(\tau_1, id(u))$ and compute a = $R.Dec(\tau_2, \lambda_{bu}[\tau_1])$;
2) If a = 0. return;
3) If u is a leaf, set $\mathbf{c}_w :=\mathbf{c}_w \cup c_u$, where c_u is the ciphertext corresponding to file identifier u and stored at node u(we have found the file containing keyword searched). Else call search(v) and search(z).

The output of this algorithm is $\mathbf{c}_w$ correspond to the ciphertexs of files containing the keyword being searched (Fig. 3).

Algorithm $f_i \leftarrow Dec(K, c_i)$: This process outputs the plaintext $f_i = \epsilon.Dec(K_3, c_i)$

3 Overview of the target platform

3.1 CUDA heterogeneous programming model

Compute Unified Device Architecture (CUDA) is developed by NVIDIA to facilitate the use of GPU for generic computations. CUDA API is able to reduce the complexity to program GPU for general purpose computing. However, a deeper knowledge of the GPU's architecture, particularly memory, threads and blocks, is crucial in order to harness its great computational power.

In general case, CPU and GPU have their own memory space, referred to as host memory and device memory respectively. CUDA program usually follows the steps below:

(i) Allocate host and device memory respectively.
(ii) Copy data from host memory to device memory.
(iii) Start kernel execution. The pointer(s) for the device memory and some other parameters are passed to the kernel.
(iv) Copy data from device memory back to the host when all GPU executions are completed.

3.2 Memory hierachy

The memory hierarchy in GPU is different from the generic CPU, due to its deep memory architecture. GPU memory can be divided into on-chip memory and off-chip memory with vast difference in bandwidth. *Global memory* is off-chip memory with largest capacity, but it is also the slowest. It is used to store the data transferred from the host, accessible by all threads in all SM. Access to Global memory needs to be done in coalesced manner (128 bytes) in order to achieve high performance.

Shared memory is accessible by all threads within the same thread block. It is commonly used to hold temporal data so that threads within the same block can exchange data.

Registers are the fastest memory in GPU, and only accessible locally by each thread. There are limited register inside a GPU, so the use of register can affect the maximum threads that can run simultaneously.

3.3 GTX980

GTX980 is a device with Maxwell architecture and compute capability 5.2. It has 16 SMs, each of the SMs consists of 128 cores, so the total cores available are 2048. Each SM is running at 1126Mhz. It is equipped with 4GB global memory, 64KB register file and 96KB shared memory per SM. GTX980 also supports Dynamic Parallelism, warp shuffle and Hyper-Q features available in Kepler (earlier generation) GPU. In this paper, GTX980 is used as a co-processor to perform the computationally intensive computation.

4 SearchaStore implementation techniques

In this section, we provide the implementation details of SearchaStore in heterogeneous platform consists of multi-core CPU and many-core GPU. GPU is used as an accelerator for certain computationally intensive algorithms. The overall work flow of SearchaStore is described below.

Client Side:

(i) Build KRBT for a set of files, **f**.
(ii) Encrypt KRBT and **f**.
(iii) Upload the encrypted KRBT and encrypted files, **c** to server.

Server Side:

(i) Receive the encrypted KRBT and **c** from client.
(ii) Construct multiple arrays based on the encrypted KRBT and copy it to GPU global memory.
(iii) Accept search token from client.
(iv) Decide to search keywords with CPU or GPU based on the proposed mechanism.
(v) Search keywords with CPU or GPU.
(vi) Return ciphertexts containing the keywords.

4.1 Build index

RBT can be implemented using record (i.e. *struct*) structure. An example of RBT node structure is shown Fig. 4.

KRBT is constructed based on RBT, hence the natural choice of implementing KRBT is to build a similar structure that allow convenient coding effort. We first construct a KRBT with amount of leafs equals to the amount of files in the dataset. Each leaf store the pointer to a file in the dataset. As the construction of RBT is inherently serial, the process does not benefit from GPU acceleration. Even though there are some parallel RBT algorithms proposed in the literature [43], it involves a lot of expensive memory operations. If GPU is used to construct RBT, the tree structure should resides in global memory as shared memory has very limited size (16-96KB). The insertion and re-balancing of RBT will then involves a lot of global memory access, which is the most expensive operation in GPU. Moreover, the tree structure is going to be transferred from CPU to GPU memory through the slow PCIe bus, but the construction of KRBT index only involves very little computation. This implies that the entire RBT build process in GPU is bound by the slow PCIe bus and global memory in GPU. In view of this, we proposed to build index using CPU only.

```
struct rbtree_node {
    struct rbtree_node *left, *right;
    struct rbtree_node *parent;
    enum rbcolor color;
    uint64_t key;
    char *data;
    bool traversed;
};
```

Fig. 4 Red black tree node structure.

4.2 Encrypt index

We generate the three master keys, K_1, K_2 and K_3 based on the *Gen* algorithm in Sect. 2. Then with K_2, we generate a series of secret key (SK) for each keyword (w_i), as described in *Enc* algorithm, step 1. This process is done in parallel in CPU, as the number of keyword is usually much smaller compared to the number of files in dataset, so this process is not very computationally intensive.

However, encrypting the entire dataset (*Enc*, step 2) is a time consuming process, so we propose to utilize the GPU to accelerate this process. Firstly, we generate a random number from PRNG in CPU as nonce; then we instantiate large pool of GPU threads, each thread encrypting a counter block based on the generated random number. Thread ID in each thread is concatenated with the nonce to form the counter value (refer to Fig. 5). From the work proposed by Kamara et al. [21], the encryption algorithm used can be any CPA-secure block cipher (we use AES in this paper) operating in counter mode (CTR). These encrypted counter blocks are XOR-ed with plaintext to encrypt the entire dataset. Since block cipher encryption in counter mode can run in parallel, it will benefit from the massively parallel architecture in GPU [39].

Block cipher operating in CTR mode can also be used to generate pseudo-random number. In this paper, we use Camelia and SEED operating in CTR mode to represent the pseudo-random function G and P. The operation to randomize the keyword location ($P_{K_1}(w_i)$) and encrypt the $data_v$ with secret key (SK) can run in parallel as well. However, these operations are not suitable to run in GPU, as we need

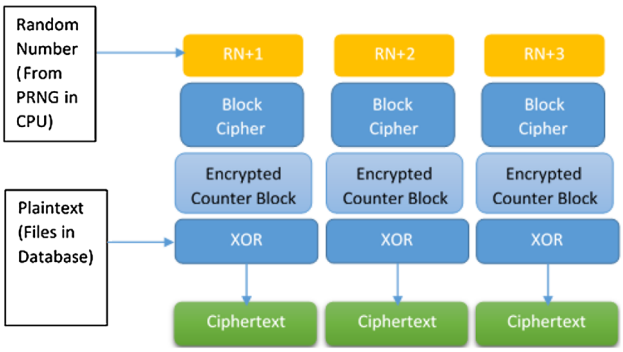


Fig. 5 CTR mode parallel encryption

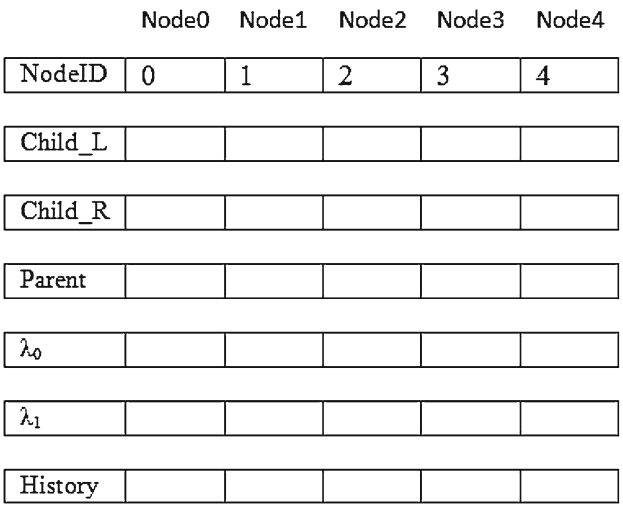


Fig. 6 Data representation for multiple array KRBT

to move part of the KRBT to and from GPU, which deteriorates the overall performance gain. Hence, we proposed to implement this part (*Enc*, step 3) entirely in CPU.

4.3 Search

The random oracle H can be implemented by using HMAC, and we use SHA-2 to generate this HMAC. It is used to determine the bit value b.

When the dataset is getting larger, the depth of KRBT is also increasing, which in turn caused the slower search speed. The server may also have problem to handle huge number of requests to search keywords simultaneously. Hence, we propose to utilize GPU as a co-processor to accelerate the search process.

State of the art GPU is only able to support recursive function call up to 32 levels [41], so it is not possible to implement KRBT in pointer form in GPU when the tree level grows beyond 32. This implies that conventional way to implement RBT in pointer-structure form (as explained in Sect.4) in GPU, can only work for small dataset.

To overcome this limitation, we propose to redesign the conventional implementation of RBT by using multiple arrays instead of pointer-structure. We proposed to store the KRBT data structure to multiple 1-D arrays, with each array representing one field in the original KRBT node structure. The data representation of proposed array based KRBT is shown in Fig. 6.

$NodeID$ is essentially the 'key' of each node. $Parent$, $Child\_L$ and $Child\_R$ are used to store the location of parent, left child and right child of current node. If current node is root, the 'parent' field will be masked with a tag value to indicate there is no parent node available. Similarly, if current node is leaf node or it does not have left or right child, the corresponding field will be masked with a tag value. λ_0 and λ_1 are the keyword hash table used for search traversal. $History$ is used to keep track a traversed node.

By representing the KRBT in multiple arrays, we can perform traversal without using stack and recursive function call. This enables us to use GPU as co-processor to accelerate the keyword search. The new search algorithm is described in Algorithm 1. We omit the notations of original KRBT SSE *Search* in order to present the new search algorithm in array based KRBT more clearly.

Assume w_i is a keyword in a dictionary of m keywords, with $i = 1,...,m$. To search for a keyword w_i, the index i is passed as input to the search module. The search process starts from the root node and perform depth first traversal to the left side of KRBT, until it reaches the left most leaf node or stops when it detects the keyword does not exists in any of the leaf nodes. Each visited node is marked as "traversed" by setting the history bit to '1'. The search continues by moving a level up, traverse to right child and continue the depth first search for this sub-tree. This process continues until all the nodes in left half of KRBT is checked for the existence of w_i. The search process then continues with the right half of KRBT with the same traversal pattern. Note that the search algorithm may not visit every nodes in KRBT to search for w_i; it only visits the child node if the keyword exists in one of the leaf nodes under this sub-tree. With this search algorithm, the root node is visited for three times only. First time when the search process starts; second time when it finishes searching left half of KRBT, it will traverse through the root node and visit right half of KRBT; third time when both left and right half of KRBT are being searched. When this condition is met, the search algorithm terminates and return the search result(s).

The original KRBT search algorithm supports parallel search which executed as follow. Assume there are p processors, processor 0 queries the root of KRBT T for a specific keyword. If the search is to be continued on both sub-tree T_v and T_u, processor 0 will continue with one sub-tree and assign another sub-tree to processor 1. The same mechanism

Algorithm 1 Search Algorithm for Array Based KRBT

```
Input:
Index i of keyword w_i to be searched.

Output:
Indexes pointing to the files that contain searched keyword.

visitRootCount = 0;
while visitRootCount <3 do                          ▷ Exit condition
  if node=root∧data_v[i]=0 then                     ▷ root node and
    return                                          ▷ no keyword
  end if
  if node=root then
    visitRootCount++
  end if
  while node_left!=NULL do                          ▷ Left child is not null
    if data_v[i]=1 then                             ▷ Depth first search
      Mark node traversed
      node=node_left
      if node_left=NULL∧node_right=NULL
       ∧data_v[i] = 1 then                          ▷ Reach leaf node
        Found keyword in leaf node
        Save index of current node.
      else
        if node=left child of parent node then
          Mark node not traversed.
        end if
      end if
    end if
  end while                                         ▷ Finished depth first search
  node=node_parent                                  ▷ Move a level up
  if node=root then
    visitRootCount++
  end if
  while node_right not traversed
       ∧visitRootCount<4 do
    Mark node_right not traversed
    if node_parent!=NULL then
      node=parent
    end if
    if node=root then visitRootCount++
    end if
  end while
  if visitRootCount<3 then
    node=node_right
    Mark node traversed
    if node_left=NULL∧node_right=NULL
      ∧data_v[i] = 1 then
      Found keyword in leaf node
      Save node index.
    end if
  end if
end while
```

is applied to both nodes u and v recursively. When there are no more available processors (i.e. all processors are working actively), the current processor selects one of the children to continue, mark the other child c as "unexplored" and push c into local stack. After all p processors finished the first round execution, each processor starts over second round by popping a node c from the local stack. This process repeats until all files containing the keyword being search are retrieved.

In this paper, we enhanced the search algorithm to map better to multi-processor architecture. Again, we assume that there are p processors available, but we do not start searching from the root of KRBT. Instead, we start traversal at level $\sqrt{p}$ from the root, by breaking the KRBT T into multiple sub-tree T_i with $i = 1,...,p$. Each processor p is assigned a sub-tree T_i, so they can search with their own sub-tree in parallel.

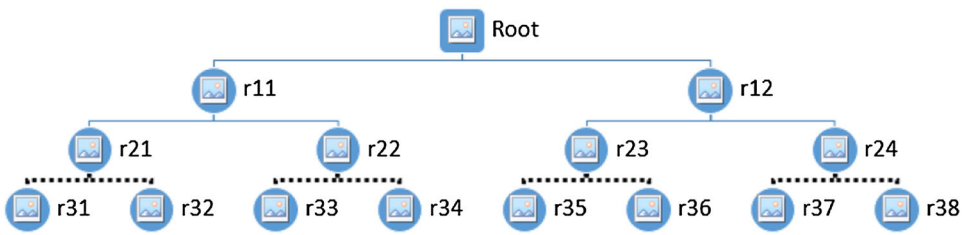
Figure 7 illustrates the enhanced parallel search algorithm. Assume that there are four available processors ($p = 4$), the search process starts at level 2 ($\sqrt{p} = 2$) from the root. Sub-tree r21, r22, r23 and r24 are assigned to one of the four processors respectively, and the search can run in parallel. This algorithm only suitable for array based KRBT, as the data of relevant node can be obtained before traversal starts by calculating the corresponding index. As for conventional pointer-structure approach, the tree traversal must start from the root node in order to reach a particular node.

Implementing KRBT in multiple arrays also helps in optimizing the memory access throughput. Since we are using multiple threads to search a keyword, each thread within a warp (32-threads) access the adjacent memory locations to read required data concurrently, so the global memory traffic is optimized. Refer to Fig. 8 for an illustrated example, thread T_0 to T_N access $NodeID$ stored in contiguous memory location, resulting coalesced memory access. This technique only

	T_0	T_1	T_2	...	T_N
NodeID	0	1	2	…	N
Child_L	156	189	356		X
Child_R	157	190	357		X+1

Fig. 8 Coalesced memory access in SearchaStore

Fig. 7 Enhanced parallel Search in SearchaStore



works for initial stage of the KRBT tree traversal. When the traversal process continues, this effect diminishes slowly, as the child of each node points to various locations that might not be adjacent to each other. However, this technique is able to improve the global memory access pattern compare to implementing KRBT in single array, whereby coalesced global memory access is almost impossible to obtain.

We implemented the new search algorithm in CPU (OpenMP) and GPU (CUDA). For implementation in CPU, each OpenMP thread represents one processor and assigned a sub-tree. Multiple threads traverse the sub-tree in parallel to search for a keyword, and the result is returned immediately after all threads finished their search. Assume that there are N keyword search requests in the queue; the proposed parallel search is repeated N times to ensure all the keywords in the queue is searched.

For implementation in GPU, we designed the system to handle batch keyword search. CUDA required the programmer to organize the thread pool in blocks and threads. For GPU with compute capability 5.2, the maximum allowable thread within a block is 1024, so we assign each block to contain 1024 threads; each thread is assigned a sub-tree to search in parallel. In addition, we also launch multiple blocks and assign one keyword to each block. With this arrangement, the GPU can search multiple keywords simultaneously, which is an improvement to the original search algorithm proposed in [21]. To implement this, we need to insert multiple *History* arrays (refer Fig. 6) into the KRBT data structure for every node. This introduces insignificant memory overhead to the original data structure. For search algorithm in GPU, the result of keywords search will be returned only when all blocks had finished searching keywords, which is different compare to CPU version.

In general, we need to load the GPU with sufficient tasks in order to fully utilize its processing power. Moreover, transferring the encrypted KRBT to GPU memory is also time consuming. Considering this limitation, we only utilize GPU for fast parallel search when the traffic of keyword search is heavy. When the traffic is low, we proposed to search the keyword with multi-core CPU only. This also corresponds to the typical industrial scenario whereby the query traffic may differ between various applications [8,9] and changes from time to time.

The overall execution steps for SearchaStore are illustrated in Fig. 9.

We proposed a mechanism to automatically select CPU or GPU for keyword search, based on the search traffic. Initially, we pick a threshold level R_c based on the hardware resources available in the computing platform (number of CPU hardware threads). When $R \leq R_c$, the software selects CPU to perform keyword search; else it will perform keyword search in GPU. This simple mechanism may not work

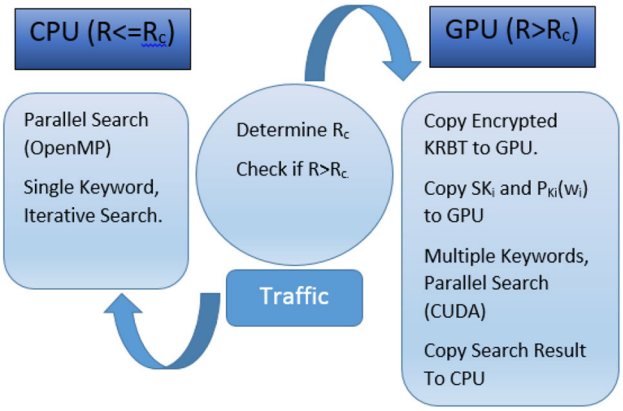


Fig. 9 Execution steps in SearchaStore

for all conditions, as the overhead of using GPU (moving data between CPU and GPU) is potentially affected by the traffic of PCI-e bus. If the PCI-e bus is occupied by other peripherals at the time we move data between CPU and GPU, it will limit the performance of keyword search in GPU; searching keywords with CPU may yield better performance in this situation. To resolve this issue, R_c can be calculated on-line based on Algorithm 2.

Algorithm 2 Accelerator Selection Mechanism for SearchaStore

Input:
R_c: Threshold level
R_{step}: Threshold adjusting step
R: Number of keyword search request
T_{up}: Time interval to update R_c
Variable:
T_{cur}: Current counter
T_{gpu}: Time taken for GPU to complete keyword search
T_{cpu}: Time taken for CPU to complete keyword search

R_c = number of CPU hardware threads
$T_{cur} = 0$
if $T_{cur} < T_{up}$ **then** ▷ Within time interval
  if $R \leq R_c$ **then**
    Perform parallel keyword search in CPU
  else
    Perform parallel keyword search in GPU
  end if
$T_{cur} = T_{cur} + 1$ ▷ Increase counter
else ▷ Update R_c
  Perform parallel keyword search in GPU
  Perform parallel keyword search in CPU
  Calculate T_{gpu} and T_{cpu} respectively.
  if $T_{cpu} \leq T_{gpu}$ **then** ▷ CPU search faster, increase R_c
    $R_c = R_c + R_{step}$
  else ▷ GPU search faster, decrease R_c
    $R_c = R_c - R_{step}$
  end if
$T_{cur} = 0$ ▷ Reset counter
end if

Initially, R_c is assigned to a value equivalent to the number of CPU hardware threads. R_{step} and T_{up} are set to a default value based on user's preference. The system selects accelerator to perform keyword search based on the traffic input R and threshold R_c. The system performs a performance check for every T_{up} elapsed to evaluate which accelerator (CPU or GPU) is performing better, based on current threshold R_c, and adjusting R_c accordingly. The value for T_{up} should not be too small as it will affect the overall search performance by performing updates too frequently. At the same time, R_{step} needs to be chosen appropriately so that R_c does not get changed too fast or too slow.

5 Results and discussion

5.1 Experimental setup and dataset

We present SearchaStore with the proposed techniques to accelerate KRBT using C language in Windows 8.1. We used CUDA SDK 7.5 for GPU computing and OpenSSL for general purpose cryptographic computation (HMAC). The experiments run on a workstation equipped with Intel Core(TM) i7-4790K (4.0 GHz) which has 4 cores and supports 8 parallel threads, 32GB RAM and 4TB SATA Hard Disk. The GPU used is NVIDIA GTX980 with compute capability 5.2.

We use the latest Enron email dataset [36] with 517,452 files and 1.32 GB in size. We extracted a subset of emails (256,000 files, 535 MB in size) containing total 16,770,000 unique file/keyword pairs.

SearchaStore utilizes the GPU's massively parallel architecture for fast index encryption and highly parallel keyword search. We do not consider the communication cost (upload encrypted index and search tokens) involved as we believed it is very much affected by the local network speed. All the client side and server side computations are executed within the same workstation. However, in actual usage the client side (build index, encrypt index and search token generation) and server side (search keywords) are executed in separate machines.

5.2 Index construction

Index construction includes the steps to build and encrypt index. In this experiment, the index is build using CPU and encrypt using GPU to utilize its massively parallel architecture for fast computation. Fig. 10 shows the time taken to build and encrypt index for SearchaStore, expressed as the cost per file/keyword pair. The lower file/keyword pair leads to higher cost of index construction, as the cost is not amortized over as many pairs in this case. The index construction cost decreases to 9.41μs per file/keyword pair when the file/keyword pair is large enough.

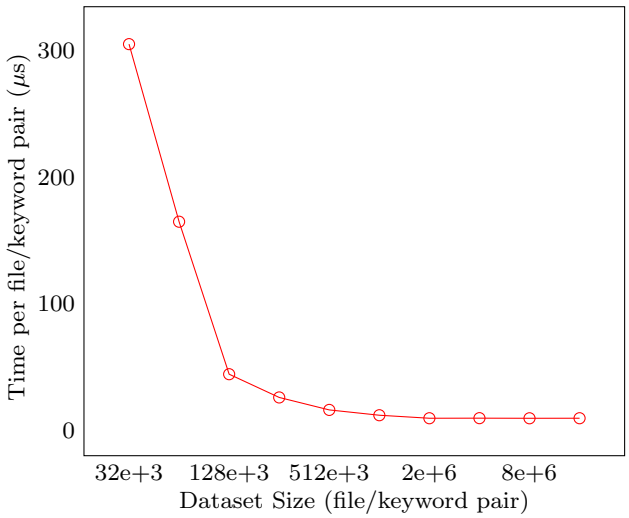


Fig. 10 Time taken for index construction (build and encrypt index)

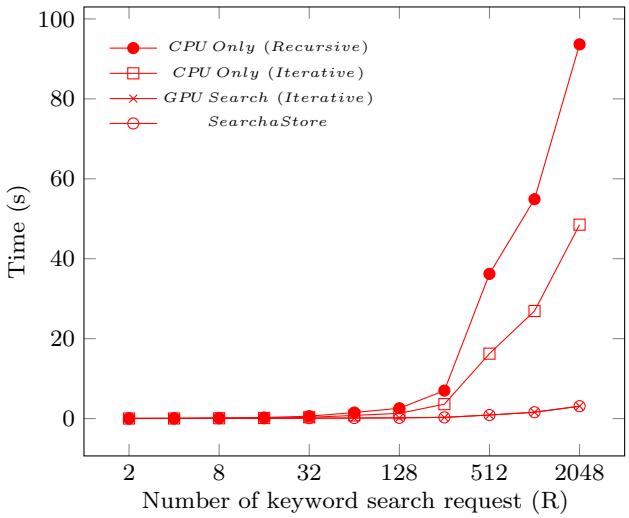


Fig. 11 Time taken for batch keywords search (linear scale)

5.3 Search

Based on the techniques proposed in Sect.5, the keyword search can be executed in CPU as well as GPU. In our experiment, we implemented both versions to compare the performance, and the results are shown in Figs. 11, 12 and Table 1. We performed the experiments with batch keyword searches ranging from 2 to 2048. In order to simulate the worst case scenario, we search common keywords that should appear in almost all documents (e.g. the, this and is). For search performance, we use the largest subset of Enron email that we have with 16,770,000 unique file/keyword pairs. We do not benchmark the search token generation as

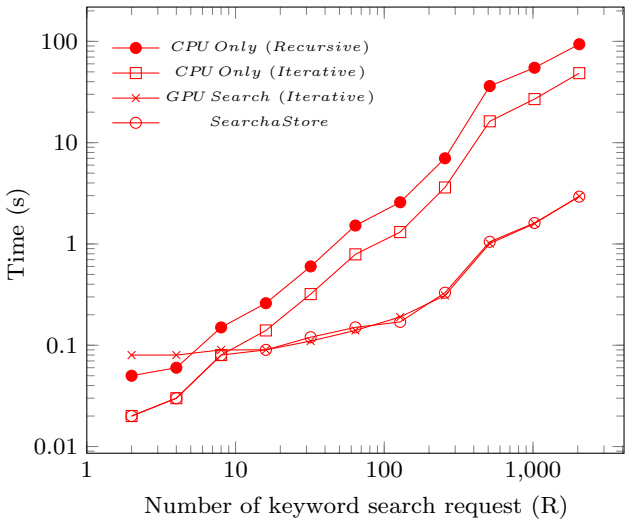


Fig. 12 Time taken for batch keywords search (logarithmic scale)

Table 1 Performance comparison for CPU and GPU search

R	Time (s)					
	CPU only recurs.	CPU only iterat.	GPU only iterat.	Sear-cha-store	Key-word found	Speed-up
2048	93.64	48.52	2.96	2.93	5821321	16.4
1024	54.90	26.92	1.59	1.61	4982146	16.9
512	36.21	16.25	1.01	1.05	3601423	16.1
256	7.01	3.61	0.31	0.33	2428648	11.7
128	2.58	1.31	0.19	0.17	1934862	6.9
64	1.52	0.79	0.14	0.15	1349653	5.6
32	0.60	0.32	0.11	0.12	1086333	2.9
16	0.26	0.14	0.09	0.09	972365	1.6
8	0.15	0.08	0.09	0.08	782169	0.9
4	0.06	0.03	0.08	0.03	329135	0.4
2	0.05	0.02	0.08	0.02	178296	0.3

it is a lightweight process compared to other computations. Moreover, the secret key SK_i and random string $P_{K_1}(w_i)$ are pre-computed in index encryption (*Enc*) process. However, we do consider the time taken for data transfer between CPU and GPU during the keyword search process.

We first compare the performance of searching keywords in CPU with conventional recursive KRBT and iterative KRBT. Table 1 shows that iterative KRBT is performing better in CPU compared to the recursive version.

The keyword search using CPU and GPU takes almost same amount of time to complete when the number of search requests (R) are small ($R \leq 8$). When R increases, GPU performance starts to overtake CPU, and the speed up saturate at certain level ($R \geq 1024$). This is due to the fact that when R is small, GPU is not fully loaded, hence its search performance tends to be slower compare to CPU as most of the cores in GPU are idle. Moreover, GPU search requires two memory transfer operations: copy the encrypted KRBT to GPU and copy the search results from GPU to CPU. This introduces some overhead to the GPU search, which explain why GPU performs poorly in this situation. However, when R increases, GPU is loaded with more work to perform simultaneous keywords search. Although the overhead to copy results from GPU to CPU also increases in this situation, the overall performance of GPU search is still superior to CPU search.

By performing a simple analysis on Table 1, we found that the search performance in GPU overtakes CPU when $R > 8$. Since we are using a four cores CPU with eight threads, theoretically it is capable to run eight keyword searches in parallel. When $R > 8$, the CPU search takes relatively longer time to complete as the parallel processes are always limited to eight only. From this, we are able to conclude that CPU search alone is sufficient to handle small number of keyword searches, and GPU can be used as an accelerator when the search traffic is huge. As a result, SearchaStore combine these two approaches: when the number of search requests is small, keyword search will be done in CPU; otherwise the search will run in GPU for better performance. The SearchaStore version is able to provide most optimized search performance in all scenario. Compare to CPU only iterative search, SearchaStore is 16.9x faster when $R = 1024$.

5.4 Results from related work

To the best of our knowledge, this paper is the first work that discusses on the implementation performance of SSE based on GPU, hence we are unable to benchmark with any existing work. This is also the first work that presents the implementation performance for the work in [21] in CPU. In this section, we present some results from Kamara et al. [14], Naveed et al. [15] and Cash et al. [22] and that implemented SSE in CPU.

For index construction, we are able to achieve 9.41μs per file/keyword pair when the file/keyword pair is large enough. For the same operation, SSE scheme proposed by Kamara et al. [14] required 35 s. On the other hand, the SSE scheme proposed by Cash et al. [22] and Naveed et al. [15] only takes 3s and 1.58s per file/keyword pair for index construction. In terms of single keyword search, we are able to achieve good performance with 1.4 ms in 535 MB dataset. This is comparable to the work presented by Naveed et al. [15] (5 ms in 4MB dataset), Cash et al. [22] (7 ms in 65 GB dataset) and Kamara et al. [14] (17 ms in 4MB dataset). SearchaStore is able to perform batch keyword search, which is not achievable by the work presented in [14,15,21,22].

We are aware that it is not fair to compare our work with the other three research works, as the experimental platform,

dataset, software libraries and algorithms used by different researchers are vastly diverse. We do not intend to sh ow that SearchaStore is superior to other existing SSE work, but rather to show that GPU can be utilized as an efficient platform to accelerate SSE computation. The results presented in Sect. 5 shows that GPU can accelerate the index encryption and keyword search process effectively.

6 Conclusion and future work

In this paper, we proposed SearchaStore with several techniques to accelerate KRBT SSE in a heterogeneous computer system consist of multi-core CPU and many-core GPU. The results show that GPU is efficient in accelerating SSE scheme when the keyword search traffic is huge (16.9x faster than CPU version), which is a common scenario for organizations that host their data in the cloud. The index encryption is accelerated by GPU to achieve high encryption speed. Besides, we also enhanced parallel search algorithm allows batch keyword search and shows superior performance compare to the original algorithm presented by Kamara et al. [21].

Currently, we are only focusing on the static construction of KRBT SSE, whereby the dataset is fixed and does not allow real time update (add or delete). The dynamic construction of KRBT SSE is useful when there is demand for frequent update on the dataset, which is an interesting aspect for future expansion based on our current work. We are also interested to extend the GPU implementation to other SSE schemes that support ranked multi-keyword search [23].

Acknowledgements This work was supported partially by Universiti Tunku Abdul Rahman Research Fund (UTARRF) under Grant IPSR/RMC/UTARRF/2016-C1/G1.

References

1. Yang, G., Xie, L., Mantysalo, M., Zhou, X., Walter, S.K., Chen, Q., Zheng, L.: A healthcare information sharing scheme in distributed cloud networks. J. Clust. Comput. **18**(4), 1405–1410 (2015)
2. Tao, F., Zuo, Y., Xu, L.D., Zhang, L.: IoT-based intelligent perception and access of manufacturing resource toward cloud manufacturing. IEEE Trans. Ind. Inf. **10**(2), 1547–1557 (2014)
3. A. Mhlaba, M. Masinde.: Implementation of Middleware for Internet of Things in Asset Tracking Applications: In-lining Approach. IEEE International Conference on Industrial Informatics, INDIN, pp. 460-469, 2015
4. Mhlaba, A., Masinde, M.: Secure outsourcing of modular exponentiations in cloud and cluster computing. J. Clust. Comput. **19**(2), 460–469 (2015)
5. Lee, S.G., Lee, D., Lee, S.: Personalized DTV program recommendation system under a cloud computing environment. IEEE Trans. Consum. Electron. **56**(2), 1034–1042 (2010)
6. Kim, Y., Ko, J., Shin, D., Kim, C., Park, C.: A frequency monitoring system development for wide-area power grid protection. J. Clust. Comput. **16**(2), 209–219 (2013)
7. Park, S., Park, E., Seo, J., Li, G.: Factors affecting the continuous use of cloud service-focused on security risks. J. Clust. Comput. **19**(1), 485–495 (2015)
8. Fang, S., Xu, L., Pei, H., Liu, Y.: An integrated approach to snowmelt flood forecasting in water resource management. IEEE Trans. Ind. Inf. **10**(1), 548558 (2014)
9. Xu, L.: Introduction: Systems science in industrial sectors. Syst. Res. Behav. Sci. **30**(3), 211213 (2013)
10. Song, X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. SP 00: Proceedings of the IEEE Symposium on Security and Privacy, pp. 44, (2000)
11. Goh, E.J.: Secure indexes. Cryptology ePrint Archive. Report 2003/216. http://eprint.iacr.org/2003/216/
12. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. ACM Conference on Computer and Communications Security, CCS, pp. 7988. (2006)
13. Chase, M., Kamara, S.: Structured Encryption and Controlled Disclosure. ASIACRYPT, Lecture Notes in Computer Science. **6477**, pp. 577594. Springer, Heidelberg(2010)
14. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. ACM Conference on Computer and Communications Security. pp. 965976. (2012)
15. Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. Proceedings of the IEEE Symposium on Security and Privacy, pp. 639–654. (2014)
16. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. J. Comput. Secur. **19**(5), 895–934 (2011)
17. Moataz, T., Justus, B., Ray, I., Cuppens-Boulahia, N., Cuppens, F., Ray, I.: Privacy-preserving multiple keyword search on outsourced data in the clouds. Lect. Notes Comput. Sci. **8566**(2014), 66–81 (2014)
18. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Highly-scalable searchable symmetric encryption with support for Boolean queries. Advances in Cryptology. Lecture Notes in Computer Science, vol. 8042, pp. 353–373. Springer, Berlin (2013)
19. Moataz, T., Shikfa, A.: Boolean symmetric searchable encryption. 8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS, pp. 265276. (2013)
20. Yu, J., Lu, P., Zhu, Y., Xue, G., Li, M.: Toward secure multikeyword top-k retrieval over encrypted cloud data. IEEE Trans. Dependable Secur. Comput. **10**(4), 239–250 (2013)
21. Kamara, S., Papamanthou, C.: Parallel and Dynamic Searchable Symmetric Encryption. Financial Cryptography, pp. 258–274. Springer, Berlin (2013)
22. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M.C., Steiner, M.: Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. Network and Distributed System Security Symposium, NDSS (2014)
23. Xia, Z., Wang, X., Sun, X., Wang, Q.: A secure and dynamic multikeyword ranked search scheme over outsourced cloud data. IEEE Trans. Parallel Distrib.Syst. **27**(2), 1–13 (2015)
24. Boneh, D., Kushilevitz, E., Ostrovsky, R., Skeith, W.E. III.: Public key encryption that allows PIR queries. CRYPTO, Lecture Notes in Computer Science. 4622, pp. 5067. Springer, Heidelberg. (2007)
25. Stefanov, E., Shi, E.: ObliviStore: High Performance Oblivious Cloud Storage. Proceedings of the IEEE Symposium on Security and Privacy, pp. 253–267. (2013)
26. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. Advances in Cryptology—EUROCRYPT, Lecture Notes in Computer Science, vol. 7237, pp. 465–482. Springer, Berlin (2012)
27. Hughes, D.M., Lim, I.S.: Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on GPUs. IEEE Trans. Vis. Comput. Graph. **15**(6), 1555–1562 (2009)

28. Kaczmarski, K.: B+-tree optimized for GPGPU. Lect. Notes Comput. Sci. **7566**, 843–854 (2012)
29. C. Kim, J., Chhugani, N., Satish, E., Sedlar, A., Nguyen, D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: FAST: fast architecture sensitive tree search on modern CPUs and GPUs. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 339–350. (2010)
30. Chen, X., Ren, L., Wang, Y., Yang, H.: GPU-accelerated sparse LU factorization for circuit simulation with performance modeling. IEEE Trans. Parallel Distrib. Syst. **26**(3), 786–795 (2015)
31. Mei, S., He, M., Shen, Z.: Optimizing Hopfield Neural Network for Spectral Mixture Unmixing on GPU Platform. IEEE Geosci. Remote Sens. Lett. **11**(4), 818–822 (2014)
32. Hu, L., Nooshabadi, S., Mladenov, T.: Forward error correction with Raptor GF(2) and GF(256) codes on GPU. IEEE Trans. Consum. Electron. **59**(1), 273–280 (2013)
33. Lee, W.K., Cheong, H.S., Phan, Raphael C.-W., Goi, B.M.: Fast implementation of block ciphers and PRNGs in Maxwell GPU architecture. J. Clust. Comput. **19**(1), 335–347 (2016)
34. Yang, Y., Guan, Z., Sun, H., Chen, Z.: Accelerating RSA with fine-grained parallelism using GPU. Information Security Practice and Experience, Lecture Notes in Computer Science, vol 9065, pp. 454-468. (2015)
35. Park, H., Park, K.: Parallel algorithms for redblack trees. Theor. Comput. Sci. **262**(12), 415435 (2001)
36. Enron Dataset. https://www.cs.cmu.edu/enron/. (2015)

Wai-Kong Lee was born in Malaysia in 1982. He received the B.Eng. in Electronics and M.Sc. degree from Multimedia University in 2006 and 2009 respectively. He is now a PhD candidate with the Faculty of Engineering and Science, University Tunku Abdul Rahman, Malaysia. His research interests are in the areas of cryptography, GPU computing, embedded system design and energy harvesting.

Raphael C.-W. Phan received his B.Eng, M.Eng.Sc and PhD degrees from Multimedia University (MMU), Malaysia in 1999, 2001 and 2005, respectively. He is currently a professor in Faculty of Engineering (FOE), Multimedia University, Malaysia. He is General Chair of Mycrypt '05 and Asiacrypt '07, Program Chair of Mycrypt 2016, and Publicity Co-Chair for IEEE Symposium on Trust, Security & Privacy for Emerging Applications (TSP '10). He annually serves in various technical program committees of cryptology and security conferences. He researches on diverse aspects of security and privacy, including cryptology, protocol security, network security and system security. He is also one of the authors for BLAKE hash function. BLAKE was selected as one of the five finalists for SHA-3 competition by NIST.

Geong-Sen Poh has a Bachelor (Hon.) degree and a Master degree in Computer Science from Universiti Sains Malaysia, and a PhD degree in Information Security from Royal Holloway, University of London, UK. He was an Assistant Professor and Dean of Centre for Research and Industrial Collaboration at UniMy, Malaysia. He is now a staff researcher in MIMOS Berhad. His main research interests include cryptographic schemes for computations in the encrypted domain such as searchable encryption, protocols for distributed systems and multimedia security. He currently serves as a committee member in the ISO standard cryptography working group (Malaysia chapter), and committee members for various international conferences. He has published in the field of searchable symmetric encryption, watermarking and information security.

Bok-Min Goi received his B.Eng degree from University of Malaya (UM) in 1998, and the M.Eng.Sc and PhD degrees from Multimedia University (MMU), Malaysia in 2002 and 2006, respectively. He is now the Deputy Dean (Academic Development & Undergraduate Programmes) and a professor in the Faculty of Engineering and Science, Universiti Tunku Abdul Rahman (UTAR), Malaysia. Prof. Goi is the Chairperson for Centre for Healthcare Science & Technology, UTAR. He was also the General Chair for ProvSec 2010 and CANS 2010, Programme Chair for IEEE-STUDENT 2012, and the PC members for many crypto/security conferences. His research interests include cryptology, security protocols, information security, digital watermarking, computer networking and embedded systems design.