GPU-Assisted Cryptography of Log-Structured Indices

by

Michael Warren Kirby

B.S., University of Colorado, 2007

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2012

This thesis entitled:
GPU-Assisted Cryptography of Log-Structured Indices
written by Michael Warren Kirby
has been approved for the Department of Computer Science

_____

Willem Schreüder

_____

Dirk Grunwald

_____

Roger King

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Kirby, Michael Warren (MS., Computer Science)

GPU-Assisted Cryptography of Log-Structured Indices

Thesis directed by Professor Willem Schreüder

General purpose programming of Graphics Processing Units (GPUs) is a relatively new technological advancement. GPUs contain vast amounts of computational power with their many core architectures. Within many computer systems the power of these GPUs often goes unused outside the realm of graphics. Many of today's common computational tasks are well suited for the single instruction, multiple data (SIMD) architecture of the GPU. Commonly used algorithms within storage systems such as block based hashing and cryptography perform exceptionally well within the GPU architecture, often far exceeding the performance of CPUs. Researched within this thesis is the viability of utilizing GPUs within modern storage systems, unlocking the capabilities of the otherwise idle graphics processor. Data throughput, hashing, and cryptography are examined with the assistance of a general purpose GPU. Along with these stand-alone tasks, a proof of concept log-structured index is designed and implemented to take advantage of GPU cryptography for at-rest data encryption. Results shown in this work demonstrate that it is feasible to achieve significant performance gains with the assistance of a GPU for cryptographic tasks within a log-structured index.

Contents

Chapter

1 Introduction 1

1.1 Motivation . . . 2

1.2 Contributions . . . 3

1.3 Assumptions . . . 3

1.4 Organization . . . 4

2 Background 5

2.1 GPU Architecture . . . 5

2.2 Log-Structured Indices . . . 6

2.3 Storage Related GPU Tasks . . . 7

3 Design and Implementation Details 9

3.1 Memory Throughput . . . 9

3.2 CRC32 . . . 10

3.3 Tiny Encryption Alogorithm . . . 11

3.4 Log-Structured Index . . . 12

3.4.1 Red-Black Tree . . . 14

3.4.2 B+ Tree . . . 17

3.4.3 Index Log . . . 18

3.4.4 Super Block . . . 19

3.4.5 Insert . . . 20
3.4.6 Remove . . . 20
3.4.7 Search . . . 20
3.4.8 Merge . . . 21

4 Results 24

4.1 Hardware . . . 24
4.1.1 Platform . . . 24
4.1.2 CPU and GPU . . . 24
4.1.3 System Configurations . . . 26
4.2 Memory Throughput . . . 26
4.3 CRC32 . . . 29
4.4 TEA Encryption . . . 33
4.5 Log-Structured Merge Index . . . 37
4.5.1 Insert . . . 38
4.5.2 Remove . . . 42
4.5.3 Search . . . 43

5 Conclusions 46

6 Future Work and Extensions 48

Bibliography 50

Appendix

A Appendix A: Source Code 52

Tables

Table

4.1 CPU Information 25

4.2 GPU Information 25

4.3 CPU to GPU System Pairing 26

4.4 CRC32 Throughput, 512 byte data blocks 29

4.5 CRC32 GPU Throughput Percent Change Over CPU, 512 byte data blocks 30

4.6 CRC32 GPU Percent Time Data Copy, 512 byte data blocks 30

4.7 TEA Throughput, 512 byte data blocks 33

4.8 TEA GPU Throughput Percent Change Over CPU, 512 byte data blocks 34

4.9 TEA GPU Percent Time Data Copy, 512 byte data blocks 34

4.10 LSM CPU/GPU Insert Index Encryption Relative to No Encryption 39

4.11 LSM Insert, Object Encryption, Operations/Second, 512 byte data blocks 39

4.12 LSM GPU Insert Object Encryption Operations per Second Relative to CPU 40

4.13 Remove Operations/Second 42

4.14 LSM Search without Object Decryption, Operations/Second 43

4.15 LSM Search, Object Decryption, Operations/Second, 512 byte data blocks 44

4.16 LSM GPU Search Object Decryption Relative to CPU 44

Figures

Figure

2.1 CUDA Process Flow 5

3.1 Tree Structures 13

3.2 Log Structure 19

3.3 Example Merge Operation 21

4.1 Memory Copy Throughput, System A 28

4.2 Memory Copy Throughput, System B 28

4.3 Memory Copy Throughput, System C 28

4.4 Memory Copy Throughput, System D 29

4.5 CRC32 Throughput Graph, System A 31

4.6 CRC32 Throughput Graph, System B 31

4.7 CRC32 Throughput Graph, System C 31

4.8 CRC32 Throughput Graph, System D 32

4.9 CRC32 Throughput Graph, System E 32

4.10 CRC32 Throughput Graph, System F 32

4.11 TEA Throughput, System A 35

4.12 TEA Throughput, System B 35

4.13 TEA Throughput, System C 35

4.14 TEA Throughput, System D 36

4.15 TEA Throughput, System E 36
4.16 TEA Throughput, System F 36
4.17 LSM Insert, CPU/GPU w/o log encryption, System A 40
4.18 LSM Insert, CPU/GPU w/o log encryption, System B 40
4.19 LSM Insert, CPU/GPU w/o log encryption, System C 40
4.20 LSM Insert, CPU/GPU with log encryption, System A 41
4.21 LSM Insert, CPU/GPU with log encryption, System B 41
4.22 LSM Insert, CPU/GPU with log encryption, System C 41
4.23 LSM Search, System A 45
4.24 LSM Search, System B 45
4.25 LSM Search, System C 45

Chapter 1

Introduction

Graphics Processing Unit (GPU) hardware has existed for a number of years, first dating back to the late 1990s [21]. Since the advent of GPUs, general purpose programming of graphics specialized hardware has become possible. OpenGL and DirectX were the first pioneers in the field. Clever tricks using graphics APIs allowed early programmers to perform general purpose tasks by using code which appeared to the GPU as graphics computations [9]. Eventually, the Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL) arrived. CUDA and OpenCL are programming extensions of the C and C++ languages that allow the direct programming of GPU hardware without the requirement of cumbersome graphics APIs. Starting in 2008, all NVIDIA graphics chips have supported the CUDA environment [9].

Since GPU hardware is relatively new, many modern computer systems do not take full advantage of their often unused graphics hardware. As general purpose GPUs become more prevalent in computer systems, extending their use beyond the realm of graphics presents a unique opportunity to offload general computing tasks. GPU task-offloading can reduce overall system cost through lower CPU requirements, while simultaneously improving overall performance. Because of their roots in computer graphics, GPU cores are designed as single instruction, multiple data (SIMD) processing units. This architecture limits their general applicability; however, GPUs can be effectively used in storage systems for tasks such as hashing, Reed-Solomon coding, and cryptography, to name a few.

Another storage construct, the log-structured merge tree, forms the basis for many indices

within modern file systems and databases. NoSQL databases such as Google's BigTable [7] and Cassandra [1] contain indices based on the Log-Structured Merge-Tree (LSM-tree) [22]. LSM-trees utilize a memory binary tree in conjunction with a circular log to facilitate a high rate of fault-tolerant index updates through sequential disk access. A memory resident binary search tree is used for update buffering, where updates are eventually merged to an on-disk B+ tree. When considering the incorporation of a GPU into an LSM system, this index buffering technique allows for batched data transfers to and from the GPU device, amortizing the cost of GPU instantiation.

The work presented within this thesis focuses on the use and evaluation of GPU assistance in storage system related tasks. These tasks include stand-alone hash calculations and cryptography. Along with these stand alone tasks, the design and implementation of a log-structured indexing system which utilizes GPU-assisted cryptography is described and analyzed. The log-structured index is designed to take advantage of the computational power of the GPU to perform at-rest data encryption of the system index and log. The design of the LSM index is based on the ideas of the LSM-tree.

1.1 Motivation

The motivation of this work is to research the viability of GPU utilization within modern storage systems. A bulk of the existing GPU research to date has been performed in the realm of graphics and high performance computing. There has been limited research in the field of storage related to GPUs. Much of the storage related research focuses on specific tasks such as hashing, cryptography, Reed-Solomon coding, etc. Within the limited GPU research related to storage, there is very little that looks at a fully integrated GPU within a storage system. This work's motivation is to utilize existing research and build upon it to examine the use of a GPU as the basis for assisted cryptography of a log-structured index. A log-structured index is chosen because many NoSQL system indices are based on the concept of the LSM-tree [7]. The aim of this research is to show that by using the otherwise idle GPUs within such indexing systems, at-rest data encryption can

be achieved with little to no impact on overall system performance. [1]

1.2 Contributions

This thesis pertains to the design, implementation, and evaluation of an encrypted log-structured index with the assistance of a GPU. The log-structured index described here is a proof of concept encrypted index. CPU and GPU based cryptography of the index are compared for insert, search, and remove operations. Aside from the log-structured index, this thesis evaluates the use of a GPU within the context of hashing, the tiny encryption algorithm (TEA), and the memory throughput of a GPU in conjunction with these tasks. This thesis expands upon prior research by providing additional analysis of the design, implementation, and incorporation of GPU cryptography into a log-structured system.

1.3 Assumptions

The proof of concept log-structured index acts as a starting point for research into a fully featured content indexing storage system. For the purpose of this work, the log-structured index is maintained in-memory and not written back to disk. The focus of this work is to measure the performance impact of utilizing a GPU over a CPU for encryption of the various components of the log-structured index. Thus, it is assumed the use of a GPU will effect only the time required to encrypt and decrypt the components of the index and not affect disk access times. By removing the added element of disk accesses, the direct impact of the GPU can be more accurately measured. That said, the system is designed for write back to disk and implemented in such a way that the feature could be added as future work.

This work assumes that the target application for GPU-assisted log cryptography is a NoSQL type storage system such as Cassandra. These systems typically use commodity hardware which have GPU boards that are otherwise unused. It is assumed that this work would not be applied to

[1] Compression was also considered for this work, but ultimately deemed an ill fit for the GPU architecture since most compression algorithms are inherently sequential in execution.

specialized storage systems which have access to many core architectures or standalone encryption hardware.

1.4 Organization

The thesis is organized as follows: Chapter 2 discusses the background information including GPU architecture and other uses of GPUs in storage related tasks; Chapter 3 discusses the design and implementation of the GPU assisted log-structured index and general throughput, hashing, and cryptography algorithms; Chapter 4 analyzes and discusses the results of the implementation; Chapter 5 discusses the conclusions acquired from the implementation and results; and Chapter 6 takes a look at future work.

Chapter 2

Background

2.1 GPU Architecture

Owing to their heritage in graphics, GPUs are optimized for performing numerous floating point calculations on data streams. As such, GPU architecture implements a limited form of parallel data execution whereby a kernel is executed independently on individual objects within a data stream. Each kernel executes an identical code block simultaneously on a number of stream processing units. This architecture works very well for embarrassingly parallel operations such as graphics processing, block based hash calculations, and encryption algorithms. However, it becomes limited for operations which require a high degree of synchronization.

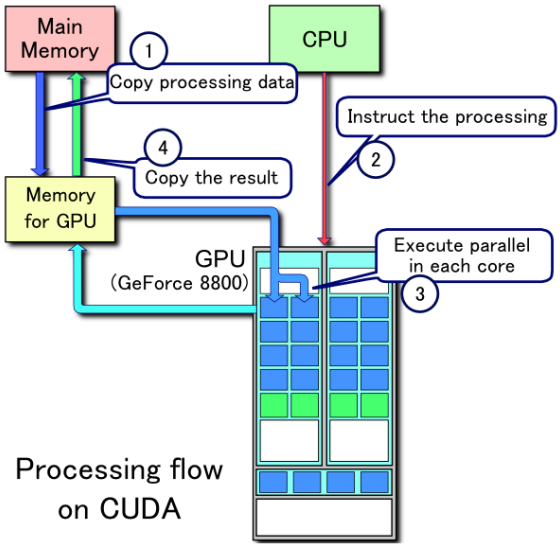

Figure 2.1: CUDA Process Flow

CUDA, OpenCL, and DirectCompute are examples of programming language extensions for GPUs. CUDA is used for the work within this thesis. CUDA adds extensions onto the C and C++ programming languages to facilitate GPU programming. Constructs have been added to schedule the execution of GPU kernels in a similar fashion to function calls in C. Data buffers can be easily allocated, freed, and copied to and from the GPU device. CUDA relies on standard C compilers such as the GNU C Compiler (gcc). Additionally, CUDA specific GPU compilation is performed by NVIDIA's Cuda C Compiler (nvcc) [20].

Figure 2.1 contains a CUDA process flow diagram [28]. Within a typical CUDA kernel execution, the first step is to copy the required data from the host memory to the GPU device. After the input data has been copied, the CPU schedules the execution of the GPU kernel, including the number of parallel execution blocks. After CPU task scheduling, the kernel is executed in parallel within each assigned GPU core. Finally, the resultant data is copied back to the host memory from the GPU device.

2.2 Log-Structured Indices

Log-structured file systems (LSFS) [18] are file systems in which user data and file system metadata are logged sequentially to disk. LSFS are designed to sustain high write throughput. Unlike traditional file systems in which modifications are made in place to existing files, a LSFS appends updates and modifications to a circular log as they occur. This sequential disk access has a tremendous impact on the write performance of file systems backed by spinning media due to the high cost associated with disk seeks. The rationale behind the LSFS is that disk accesses on reads are minimal as ever larger amounts of system memory are reserved for buffer caching. This assumption does not always hold, especially as disk and data set sizes have continued to increase at rates faster than buffer cache sizes. The system memory cost associated with ever increasing buffer requirements leads to smaller relative cache sizes and thus more read accesses, which can be highly random within the LSFS. However, regardless of buffer cache sizes, LSFS work well in systems with a high random write profile. Because LSFS use the disk as a continuous circular log, the file system

must clean up after itself to prevent the file system from running out of space. In the event that a cleaning operation encounters a log entry for stale (since re-written) data, the space can be freed for future consumption. If the cleaner finds the newest version of a data block, it must be moved to the current log location before the existing entry can be re-allocated.

Log-structured Merge Trees (LSM-trees) [22] build on the concept of the log-structured file system but are intended as data content indices for applications such as databases. Like the LSFS, the LSM-tree is designed to facilitate rapid index updates and therefore can sustain a high transaction rate. An LSM-tree utilizes the storage as a circular log by placing index keys, values, and object data sequentially onto the disk. LSM-trees utilize an in-memory search tree, typically a binary tree, as an update buffer in conjunction with a log which can be replayed to protect against failures. Periodic merge operations migrate the updates from the in-memory tree to a disk based tree, typically a B+ tree.

NoSQL [1] systems such as Google's BigTable [7] and Cassandra [1] are relatively new in terms of data storage systems. These NoSQL systems are designed to be large, highly scalable systems in order to accommodate the vast amount of data present with the advent of the Internet. The indices contained in these systems base their ideas on that of the LSM-tree.

2.3 Storage Related GPU Tasks

Hybrid sorting algorithms have been studied on the GPU [6, 14, 19]. These algorithms take advantage of the parallel nature of GPUs to implement quick merge sorts. Sintorn and Assarsson [6] describe a parallel bucket sort utilized to split an input data set into many sub-sets that are then sorted using a parallel merge sort. The use of such a high degree of parallelism on the data set allows the GPU to outperform efficient CPU based sorting algorithms such as Quicksort.

Content-based caching is a means to implement buffer caching by data block content rather

[1] Concerning NoSQL, Wikipedia states "In computing, NoSQL is a broad class of database management systems that differ from the classic model of the relational database management system (RDBMS) in some significant ways, most important being they do not use SQL as their primary query language. These data stores may not require fixed table schemas, usually do not support join operations, may not give full ACID (atomicity, consistency, isolation, durability) guarantees, and typically scale horizontally." [29]

than by traditional block address. Benefits of content-based caching include cache data de-duplication and larger effective cache size. Content-based caching is described by Morrey and Grunwald in [4]. Hash computations are used to determine if two or more block data contents are identical. Hash collisions are handled by performing bit-wise comparisons of data blocks. These types of applications are well suited for GPU use, as many data blocks can be compared in parallel and executed on multiple GPU cores.

GPU-Assisted Buffer Management describes the implementation of a buffer cache utilizing a GPU [13]. Within the paper, two cache algorithms are described. The first uses the GPU memory as additional buffer space. Cached data is gradually staged out of main system memory and moved into the GPU buffer. The second uses the GPU to perform block-based hash computations for content caching. Zhong and He [13] mention that the overhead associated with the GPU instantiation, including data transfer time, was sufficient to warrant the batching of data between main memory and the GPU. Multiple data blocks at a time are staged to and from the GPU, rather than as needed on demand.

GPU-based block cryptography has been shown to outperform CPU based cryptography implementations by as much as 42 times for Blowfish encryption and 12 times for AES encryption [3, 23]. In addition to content and cryptographic uses, Reed-Solomon coding, commonly used for RAID6 disk redundancy, is demonstrated on GPU hardware by Curry, Skjellum, Ward, and Brightwell in [17].

Chapter 3

Design and Implementation Details

3.1 Memory Throughput

A key factor in the performance of offloading work to a GPU is in the data transfer from the host memory to that of the device and back. Most modern GPUs are connected via the PCIe bus. The cost associated with this transfer varies based on a few parameters, the most important being the speed of the PCIe bus and the amount of data being transferred [8].

CUDA allows for two methods of copying data between the host and device memory. Pinned transfers occur when the host memory is physically pinned and is used directly for the DMA transfers between the host and device, while unpinned transfers utilize a smaller secondary staging buffer that is pinned for the DMA operations. Unpinned transfers incur an additional copy, as the data must first be staged into the pinned secondary buffer before the DMA takes place.

For measuring the throughput of the GPU within this work, variable buffer sizes are used when transferring data between the host and device using both the pinned and unpinned memory mechanisms. A fixed block size of 512 bytes is used for transfers, starting with 1 block and increasing by a power of two through 256K blocks, or 128MB in a single transfer. The time taken for each transfer was measured, with the total throughput and latency per block being calculated. Measurements were taken using CUDA event timers, which provide accurate GPU timings. These throughput measurements provide an absolute maximum data throughput for tasks offloaded to the GPU. Using variable block sizes gives an idea of how large transfers must be to justify offloading a given task to the GPU. Results can be found in section 4.2. Sample code can be seen in listings

A.1 and A.2.

3.2 CRC32

CRC32 hash calculations can be easily executed in parallel over a number of non-related data blocks. The order in which hash calculations are performed on individual data blocks has no bearing on the final output. This type of embarrassingly parallel calculation suits itself very well to the SIMD architecture of a GPU. A large data set can be sent to the GPU and split into individual blocks for computation on the GPU cores. CRC32 was chosen for its simplicity over other block based hash algorithms, whose results, in terms of CPU and GPU comparable measurements, would be similar.

The CRC32 algorithm has three main implementations: a single-threaded CPU calculation which iterates through the blocks in sequential order; a threaded CPU calculation which spawns a pthread [1] per CPU core; and a GPU implementation which copies the data to the device, splits up the CRC32 calculations between GPU cores, and copies the result back to the host. With the GPU calculation, two timing measurements are recorded, the total time and the CRC32 time. Total time is defined as the time required to copy the buffer from the host to the device, perform the CRC32 calculations, and copy the data back to the host. The CRC32 time is defined as only the time required to perform the CRC32 calculation on the device. The reasoning for providing two measurements is to obtain a data point for the overhead associated with the buffer copies. With the threaded CPU calculations, the time required for instantiating the pthreads is included in the measurement.

Locking, or other synchronization techniques dramatically impact the performance of GPU calculations such as the one implemented for the CRC32 measurements. As such, this implementation requires no locking and relies on the GPU thread and block identifiers to determine which data

[1] OpenMP was considered for the multi-threaded CPU implementation. Ultimately, pthreads were chosen for the greater degree of control they provide. Preliminary results from OpenMP indicate that threading optimizations are performed on small data sets, outperforming the pthread implementation used here, while with large data sets, the performance differential was found to be negligible.

block(s) within the set to operate on. Similarly, the multi-threaded approach uses thread index values to determine which block(s) within the set a given thread is to operate on. Due to the use of identifiers, neither approach requires locking the data set during execution.

As with the memory throughput implementation, variable buffer sizes are used for performing the CRC32 calculations. A fixed block size of 512 bytes is used for transfers, starting with 1 block and increasing by a power of two through 256K blocks, or 128MB in a single data set. Results can be found in section 4.3. Sample code can be seen in the listings A.3, A.4, and A.5.

3.3 Tiny Encryption Alogorithm

The Tiny Encryption Algorithm (TEA) was chosen over other block encryption algorithms because of its simplicity. TEA can be written in a few tens of lines of code. As a block-based encryption algorithm, TEA is ideal as a proof of concept encryption implementation, since results can be correlated with other block encryption algorithms such as AES. TEA uses a symmetric encryption algorithm consisting of sixty-four rounds of bit XOR operations on a pair of thirty-two bit blocks. Performance comparisons between TEA and other block cryptography algorithms such as AES can be found in [25] [2] .

TEA encryption for the purpose of this thesis is implemented with single-threaded CPU, multi-threaded CPU, and GPU implementations. For the multi-threaded implementation, pthreads [3] were used to create one thread per CPU core. Time measurements include the setup and tear down of these pthreads. With the GPU implementation, two timing measurements are recorded: the total time and the TEA time. Total time is defined as the time required to copy the buffer from the host to the device, perform the TEA encryption, and copy the data back to the host device. TEA time is defined as only the time required to perform the TEA encryption on the device. The reasoning for the two measurements is to provide a data point for the overhead associated with the

[2] When comparing CPU and GPU block-based cryptography algorithms, the architectural differences between integer bit-wise operations is worth considering. GPU accelerated AES encryption has been shown to be effective in [26].

[3] As with the CRC32 implementation, OpenMP was considered over pthreads for the multi-threaded TEA implementation. Ultimately, pthreads were chosen for the same reasons given for CRC32.

buffer copies.

As with the throughput and CRC implementations, the TEA implementation uses variable buffer sizes for each implementation type. A fixed block size of 512 bytes is used for transfers, starting with 1 block and increasing by a power of two through 256K blocks, or 128MB in a single data set. Each data set is split up into independent thirty-two bit block pairs for TEA operations. These independent blocks do not require synchronization and as such the threaded and GPU implementations do not require locking. Results can be found in section 4.4. Sample source code can be found in the listings A.6, A.7, and A.8.

3.4 Log-Structured Index

Log-Structured Merge Trees (LSM-Trees) are based on a similar concept to the Log-Structured File System (LSFS). The prevailing rationale behind log-structured systems is that they typically sustain a high rate of updates and thus a high write to read ratio. Writing updates as a sequential log can greatly improve overall system performance. This also happens to fit well with the GPU architecture, where bulk data transports can be ushered to the GPU for encryption at once, rather than as updates occur. Because of PCIe transfer times, the cost of moving data to and from a GPU device is amortized by using large buffers. The design of the log-structured index discussed here achieves at rest data encryption of the log as well as the index. The log-structure contains B+ tree index information as well as data objects. Since the log is written sequentially, the cost of the PCIe transfer can be shared between many updates.

The LSM index discussed here is designed as a proof of concept which could be integrated into a larger system such as Cassandra or another NoSQL database which uses an LSM-Tree like data structure for content indexing. A proof of concept was implemented since the measurement harness could be more easily controlled within a custom framework than with integration into a larger system such as Cassandra. CUDA and OpenCL extensions are also not readily available for languages other than C and C++. The Cassandra open source NoSQL system is written in Java, and therefore incorporating CUDA and or OpenCL into Cassandra would take additional libraries.

The LSM index proof of concept is designed so that it can be plugged into these systems either as a port to Java, or utilizing the Java C native environment.

As a proof of concept, this LSM index does not perform disk IO accesses for updating the log or sending merged index nodes to disk. There are a few reasons this decision was made. First, disk accesses would not have a direct impact on the metric being measured, namely the performance implications of GPU-assisted cryptography. Second, disk access times do not directly effect the performance comparisons of encryption with the CPU or GPU, and removal of the disk allows for more accurate timings. For the purposes of this work, the log consists of a rolling memory buffer which acts in the same fashion as a disk-based log. Similarly, the B+ tree is implemented as an in memory index which is not written back to disk. These features are designed and implemented such that disk accesses could be added with future work [4] .

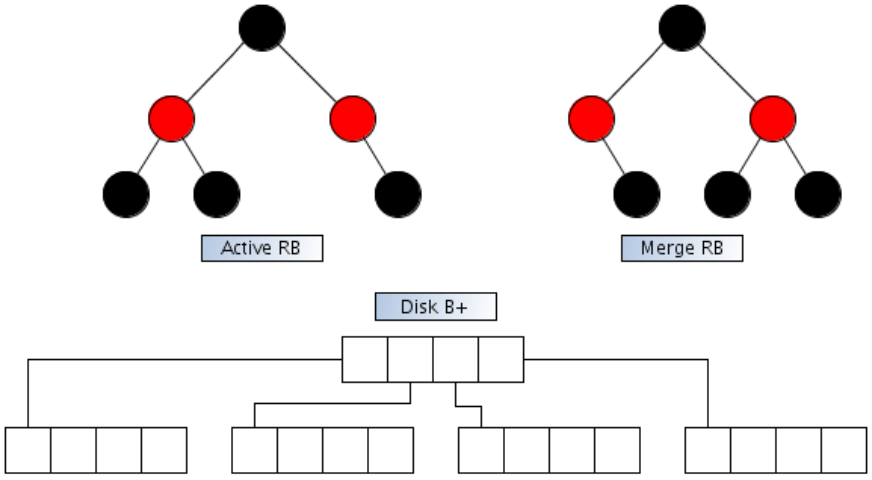

Figure 3.1: Tree Structures

The implementation of this LSM index differs slightly from that described in the LSM-Tree [22] paper in a few ways. Two in-memory binary trees are used for the purpose of updates to the index in order to allow concurrent access of the memory tree for index updates during a merge

[4] If battery or flash backed DRAM hardware were present, the log could be sent to non-volatile memory prior to being flushed to disk. This would allow the coalescing of index updates, thereby improving overall encryption throughput.

operation. These two trees are an active tree and a merge tree. The active tree handles current updates to the index and has read-write access. The merge tree handles merges to the B+ tree and is read-only. This prevents the need for coarse-grained locking of the binary tree during concurrent updates and merges. Updates are freely added to the active tree without affecting the merge. Merges do not require an exclusive lock on the read-only merge tree. This design decision does impact searches since there is now an additional tree that needs to be queried for index information. An example layout of the index trees can be seen in Figure 3.1.

3.4.1 Red-Black Tree

The binary search component of the LSM index is implemented as a red-black (RB) binary search tree. An RB tree was chosen as it is a well-known, efficient, and balanced binary search tree. There are two RB tree instances in the system: the active and merge trees. As stated previously, the active tree handles current updates to the index while the merge tree handles merges into the B+ tree. The RB trees are effectively double buffered; when a merge request is generated, the active tree transitions into the merge tree and begins merging index entries into the B+ tree. The double buffered RB tree then becomes the active tree, starting a new tree for future updates. An RB binary search tree is used for the memory trees instead of a B+ tree because there is no need to minimize tree depth at the cost of CPU-time complexity. An RB binary search tree has a time complexity of O(log N) for insert operations, while a B+ tree has a time complexity of O(t log N), where t is the minimum degree of the tree [27].

Within this implementation, there is a configurable number of total entry elements for the RB trees. Changing this value is analogous to changing a buffer cache size. More entries provide a larger buffer, whereas fewer entries provide a smaller buffer. Merge operations are started at a configurable threshold of in-use RB entries. For example, if a threshold is set at fifty percent of the entries in-use, a merge would be initiated once the number of in-use entries reaches half of the total. The active tree would transition to the merge tree while a new tree is created for future updates. In the event that no new entries are present, future updates are blocked until entries become available

by way of the merge process. Each index element contains an RB entry structure which is used to attach the element to an RB tree (active or merge). The RB entry structure contains an offset, a flag for red or black color, and left, right, and parent pointers. The offset is used to track the RB entry offset within a parent data structure. Along with the RB entry, each structure contains a linked list structure for placing the element on the free list for new allocations. Aside from these book-keeping structures, the LSM index elements contain an operation, a key, and a value. The operation field is used to denote if this is an insertion or a deletion. Because updates are buffered and merged to disk at a later time, it is important to denote if each entry is an insert or a delete, so search operations are able to find removed entries that may not have been committed to the B+ tree. The key is a 63 bit identifier for the element. The value is a 64 bit address containing the location of the object within the log. Listing 3.1 contains a source sample of these data structures.

Listing 3.1: RB Tree Data Structures

```
typedef struct ges_listElement_s {
    unsigned int offset;
    struct ges_listElement_s *next;
    struct ges_listElement_s *prev;

} ges_listElement_st;

typedef struct ges_rbElement_s {
    unsigned int offset:31;
    unsigned int red:1;
    struct ges_rbElement_s *left;
    struct ges_rbElement_s *right;
    struct ges_rbElement_s *parent;

} ges_rbElement_st;

typedef enum ges_lsmOp_e {
    GES_LSM_OP_INSERT,
    GES_LSM_OP_REMOVE,

} ges_lsmOp_et;

typedef struct ges_lsmDiskElement_s {
    uint64_t op:1;
    uint64_t key:63;
```

```
    uint64_t value;

} ges_lsmDiskElement_st;

typedef struct ges_lsmElement_s {
    ges_rbElement_st rbElement;
    ges_listElement_st listElement;
    ges_lsmDiskElement_st disk;

} ges_lsmElement_st;
```

3.4.1.1 Active Tree

The active tree is an RB tree whose purpose is to buffer recent index updates within the system. As new updates are entered into the index, they are first sent to the sequential log. The log is used to protect against failures before the update information has been merged to the B+ tree and hardened on disk. Index updates added to the RB tree use standard RB tree algorithms. The tree is protected from concurrent access by a mutex. The active tree is the first tree searched when querying the index for an element. All new updates, be they an insert or remove, use the active tree.

3.4.1.2 Merge Tree

The merge tree is an RB tree whose purpose is to merge RB tree entries into the B+ tree. The merge tree and the active tree are double buffered. When the number of entries within the active tree reaches a configurable threshold, the tree transitions into a merge tree. Merge trees are read-only within the system, therefore, no new entries may be added. Search operations look within the merge tree if the desired entry was not first found in the active tree, while merge operations read and merge the elements within the tree into the B+ tree. Merge trees cease to exist after the completion of a merge operation, with their tree elements being placed onto the system free list.

3.4.2 B+ Tree

On-disk, the index is stored as a B+ tree for efficient storage utilization and access. As mentioned previously, for the purpose of this proof of concept, the B+ tree is implemented as an in-memory tree. The design and implementation does not preclude writing the B+ tree to disk. The reason for this decision is that measurements are taken based on the time requirements for both encrypting the tree nodes as they are merged as well as decrypting them as they are searched. This timing can be accurately measured with a memory resident B+ tree. Encryption within the B+ tree is configurable to use either the CPU, or GPU, or it can be disabled.

B+ tree nodes are aligned as 8KB blocks. 8KB was chosen in order to be a multiple of the standard 512 byte disk sector size as well as the 4KB page size of most operating systems. 8KB is also small enough as to not incur excessive disk reads in the case of tree node look-ups. Stored within the 8KB disk nodes are 511 key-value pairs at 128 bits each (64 bit key, 64 bit value) as well as a 128 bit header. The header contains the number of keys and a monotonic sequence number for the node.

A B+ tree was chosen over a standard B-tree for space savings since B+ trees force the data values to be pushed to the leaf nodes. Internal nodes then contain only child node pointers. This structure has a number of advantages. First, this allows for easy in-order traversal and iteration of the key-value pairs within the index. Starting with the left most child, the keys can be traversed in order by scanning through the leaf nodes. Second, the B+ tree is more compact as the inner nodes can re-use the value entry as a child pointer. LSM disk-based data structures can be seen in listing 3.2.

Listing 3.2: LSM Tree Structure

```
#define GES_LSM_ENTRIES_PER_BTREE_ELEMENT ( 511 ) /* 8K btree nodes (16 byte
    header) */

typedef struct ges_lsmDiskElement_s {
   uint64_t key;
   union {
      uint64_t value;
```

```
        uint64_t child;
    } use;

} ges_lsmDiskElement_st;

typedef struct ges_lsmBTreeElement_s {
    uint64_t keys;
    uint64_t seq;
    ges_lsmDiskElement_st element[GES_LSM_ENTRIES_PER_BTREE_ELEMENT];

} ges_lsmBTreeElement_st;
```

A departure was taken from traditional B+ trees in that the nodes of the index are densely packed. This is done as is described in the LSM-Tree [22] because node operations are performed by the background merge task, with multiple entries being merged into nodes at once rather than a single update at a time. This batched update allows the implementation to maintain full index nodes with little additional overhead.

The B+ tree is the final tree to be searched on index retrieval. The existing B+ tree remains constant throughout a merge process. Rather than update existing tree nodes, the merge creates new nodes. As new nodes are created, they are not immediately added into the B+ tree. New B+ tree nodes are allocated and populated with the merged entries both from the RB merge tree and the existing B+ tree. After the merge completes, the root node pointer of the B+ tree is updated to reflect the state of the new B+ tree. This is done before any of the RB entries in the merge tree are freed so that searches will find the correct data if present in the merge tree.

3.4.3 Index Log

The index log serves the dual purpose of being the data store and protecting the index against failures. Before index updates are sent to the B+ tree, they are committed to the log and buffered in the RB tree structures. Should there be a system crash before those entries are merged to the B+ tree, those updates would be lost. To prevent this, the log contains the index information (key-value pair) and object data. If a system crash occurs, the log can be replayed and thus no updates would be lost. As mention previously, the proof of concept log is implemented as an in-memory

log. The design and implementation does not preclude sending this log to disk.

Along with the key-value information, the log also contains the object data. This is an important aspect as it saves potential random write access when updating the index. The value stored within the index is the address pointer to the data object. In the event of a log wrap, in-use data objects need to be moved to the current tail of the log. For the purpose of this proof of concept, log cleaning was not implemented. The use of a log-structured approach trades read accesses for quick index updates in the form of sequential writes. This is deemed an appropriate compromise given the typical access patterns of NoSQL systems.

In this implementation, as with the B+ tree, the log has configurable encryption parameters. The log can be encrypted using the GPU or CPU, or left unencrypted. The size of the data objects is also configurable; because the overall GPU encryption throughput increases with transfer size, a variable data object size demonstrates in which instances using the GPU makes sense over the CPU. Additionally, log entries could be coalesced to improve encryption performance. An example log structure can be seen in Figure 3.2.

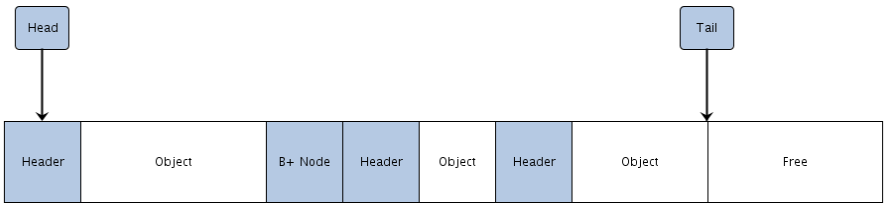

Figure 3.2: Log Structure

3.4.4 Super Block

The LSM-Tree design contains duplicated super blocks at well-known locations on disk for recovery. The super blocks contain a pointer to the current root node of the B+ tree as well as the current replay address for the log. Should the system crash, the super blocks are read, updating the in-memory root node to reflect the data present on disk. The recent log updates are

read and replayed, restoring the state of the index. As previously mentioned, the proof of concept implemented here contains an in-memory B+ tree; therefore, the disk-based super blocks were not implemented. The design and implementation of this index does not preclude their inclusion.

3.4.5 Insert

An insert operation is fairly simple in the LSM index. First, a log entry is inserted for playback in case of system failure. After the log entry has been committed, the entry is added to the active RB tree. Standard RB tree rules apply for the insert and tree re-balancing may be required. An exclusive lock on the active tree is held during an insert. Objects placed into the active tree are not encrypted; only objects logged on disk are configurable for encryption. After adding the object to the RB tree, a check is made to determine if a merge should be started.

3.4.6 Remove

In the case of the LSM index, a remove is essentially an insert noting that this data is to be removed from the B+ tree on the next merge. Search operations that find a removed item terminate since the data is no longer present in the index. Continuing to read the index data from the B+ tree could result in stale data. As with the insert, the first step is to log the key-value pairs denoting the object to be removed. The only difference from the insert is that no data object accompanies the remove; only the key-value pair to be removed is logged. Once the remove has been logged, a negative RB entry is placed onto the active tree. As with the insert operation, an exclusive lock on the active tree is held during the remove. Data placed into the active tree is not encrypted because only data in the log is configurable for encryption. After adding the removed object to the RB tree, a check is made to determine if a merge should be started.

3.4.7 Search

Search operations within the LSM index are not as simple as performing a single tree search operation. There are three trees which must be searched: the RB active tree, the RB merge tree,

and the B+ tree. The search is ordered, with the first tree searched being the active tree, followed by the merge tree and finally the B+ tree. If data is found in any of the trees, the search is completed at that tree. Continuing the search to subsequent trees could result in stale data being returned. It is also crucial to stop the search if a removed entry is encountered in either of the RB trees, as this indicates an item to be removed.

3.4.8 Merge

Merge operations are triggered when the number of entries in the active RB passes a configurable threshold. By default, the merge operation is begun when fifty percent of the total system RB elements are in use. Merge operations are started before the total number of entries are exhausted so that new index updates may be added to the index without having to block while waiting for the merge to complete. There is only one merge operation in progress at a time in the system. Should the system run out of free RB tree elements waiting on a merge, future updates are forced to wait for the merge to complete, thus freeing RB entries. An example merge operation can be seen in Figure 3.3.

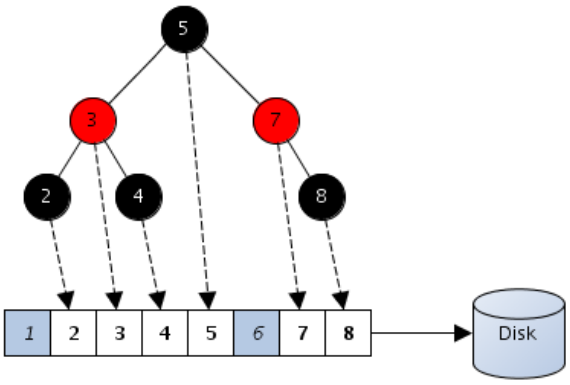

Figure 3.3: Example Merge Operation

Once it is determined that a merge should be performed, a check is made for a current merge in progress. If a merge is in progress, a new merge is not requested. If there is no current merge, the

active RB tree is locked, preventing new updates. While the tree is locked, the active tree pointer is updated to point to the unused double buffered tree. This results in a new empty active tree. The merge tree pointer is then updated to point to the previous active tree. This sets up the tree pointers so that new updates can occur in the system without requiring exclusive locks to be held on the merging tree elements. After the pointers are updated, both trees are unlocked. This break between the active and merge trees also allows for a snapshot in time of the index, which is used for updating the log replay pointer. Once the merge completes, the log replay pointer can be easily updated to the first entry after the start of the merge, lessening the time required for recovery.

The merge process itself is fairly straight forward. Both the RB merge tree and the B+ tree are ordered; thus, merging only requires starting with the left most nodes (lowest index) and performing a merge sort on the nodes. Updates to the B+ tree are not made in place. This is done because updating in place would break the sequential log access of the index and potentially lead to a corrupt tree in the event of a system crash. Preserving the state of the tree during merges allows recovery to utilize the existing B+ tree, playback the log and start a new merge. Newly updated B+ tree nodes allocate new space and write the node out anew. Before nodes are written out they are first filled completely rather than split at the standard B-tree limits. This is done because the merge operation is a background task and coalesces many updates into large bulk operations. Space can be conserved since time constraints are not as critical. The merge progresses from the lowest index value through to the highest.

After the entire RB merge tree has been merged into the B+ tree, the super blocks on disk are updated to reflect the new root node location as well as the new log replay pointer. The replay pointer of the log is noted when a merge operation starts. Because the merge is begun at a clean break point in time (when the new active tree is created), it is easy to track where the first entry not contained in the new B+ tree begins. After the super blocks are updated, the in-memory root node pointer is updated to reflect the new root node.

Once the merge has completed and the new root node has been updated, the merge tree is no longer required. All of the entries contained within the merge tree are freed for future use.

The system is checked to see if a new merge operation should be performed. This can occur if the threshold of in-use entries on the active tree has been reached. If it is determined that a new merge should be begun, the process is started again.

3.4.8.1 Encryption

Encryption is configurable within the LSM-index system. It can be performed by the GPU or CPU, or it can be disabled all together. Encryption only takes place on the log and on the B+ tree during merge operations. The in-memory RB trees are not encrypted; thus, only at-rest encryption of the data is achieved. The same TEA algorithms implemented for benchmarking are reused for the LSM-index. Due to the nature of the merge operations, it is possible to transmit data in bulk to the GPU for encryption following the merge. The log encrypts the key-value information along with the object data. Because the log is updated on each index update and key-value pairs are only 128 bits in size, the object size has the largest effect on encryption throughput. The larger the object size, the more efficient the use of the GPU becomes.

3.4.8.2 Decryption

As with encryption, the same TEA algorithms and code are reused to perform decryption of the index. The primary difference between the encryption and decryption schemes is that when using decryption, the fixed size index nodes are decrypted along with object data. Therefore, each search operation will generally require at least one 8KB B+ tree node to be decrypted. Therefore, GPU throughput is not optimal due to the smaller size data buffer. Decrypting the data objects is similar; as with encryption, the larger the object, the higher the GPU throughput.

Chapter 4

Results

4.1 Hardware

The results were tested using a selection of commodity hardware parts. Using commodity hardware made sense for a number of reasons. First, the hardware is easier to acquire and test on. Second, this type of hardware, albeit server variety, is typically what is in use for NoSQL systems. Thus the results shown here would be applicable to those system setups. Due to the use of CUDA, the graphics cards tested were limited to NVIDIA cards. It is expected the use of OpenCL would produce similar results.

4.1.1 Platform

Linux was used for development and testing for the majority of the results. Linux was chosen over other operating systems for ease of development and availability of resources. Compilation was performed using GCC and NVCC (NVIDIA CUDA Compiler). Version 4.5 of GCC and version 3.2.16 of NVCC were used. For the purpose of obtaining results on a different operating system, OSX was used on one system for results on CRC32 and TEA. Due to hardware constraints, performance measurements were not taken on the OSX system with the LSM index.

4.1.2 CPU and GPU

CPU performance has a direct impact on the overall results. Not only does the CPU provide a benchmark for cryptography, it is also used for general purpose tasks and maintenance within

the LSM index. Those tasks include management of both red-black and B+ trees, free resource list management, and general execution flow control. While the GPU can assist with cryptography, the majority of the flow and control is managed by the CPU. Table 4.1 shows the key performance metrics of the CPUs tested.

	i7-2600	**i7-980X**	**i7-950**	**Core 2 Duo**	**i3-540**	**i5-2520M**
CPU Frequency	3.40 GHz	3.33 Ghz	3.07 GHz	2.53 GHz	3.07 GHz	2.50 GHz
CPU Cores	4	6	4	2	2	2
Hyper-threads	8	12	8	N/A	4	4
Core Frequency	1600 MHz	1596 MHz	1600 MHz	Unknown	1197 MHz	800 MHz
Cache Size	8192 KB	12288 KB	8192 KB	8192 KB	4096 KB	3072 KB

Table 4.1: CPU Information

GPU hardware has a direct impact on the overall system performance when using a GPU for assisted cryptography. The most vital metrics that determine GPU performance in this case are the PCIe bandwidth (1x, 2x, etc), number of CUDA cores, and clock speed. Not as important, but still critical is the memory capacity. A larger memory capacity allows for larger data buffers to be sent to the GPU for computation. The PCIe speed effects the transfer rate of data to and from the GPU card and, as the results indicate, is ultimately the bottleneck within the system. The number of CUDA cores and their clock speed directly effects the rate of computation. Table 4.2 shows the key metrics for the GPUs tested. An important distinction for the NVIDIA GeForce 9400M is that it is not a stand-alone PCIe card; instead, it is directly connected to the motherboard.

	GTX 550 Ti	**GTX 480M**	**GTS 450**	**9400M**
CUDA Cores	192	352	192	16
Processor Clock	1800 MHz	850 MHz	1566 MHz	450 MHz
Memory Clock	2050 MHz	1200 MHz	1804 MHz	Unknown
Memory Size	1 GB	2 GB	1 GB	256MB
Memory Interface	GDDR5	GDDR5	GDDR5	Unknown
Memory Interface Width	192-bit	256-bit	128-bit	Unknown
Memory Bandwidth	98.4 GB/sec	76.8 GB/sec	57.7 GB/sec	Unknown
PCI-E Bus Support	PCI-E 2.0x16	PCI-E 2.0x16	PCI-E 2.0x16	N/A

Table 4.2: GPU Information

4.1.3 System Configurations

The systems used to test the implementations outlined in this thesis had a variety of hardware components as listed in the previous CPU and GPU sections. Table 4.3 contains a listing of the systems with their CPU and GPU configurations. Some of these systems were workstations while others were laptops. Because of these physical limitations, the hardware pairings were not changed to produce different permutations of CPU to GPU pairings. Result figures and tables included within this section refer to the system names within Table 4.3.

	CPU/Host	**GPU/Device**	**OS**
System A	i7-2600	GTX 550 Ti	Linux
System B	i7-980X	GTX 480M	Linux
System C	i7-950	GTS 450	Linux
System D	Core 2 Duo	9400M	OSX
System E	i3-540	N/A	Linux
System F	i5-2520M	N/A	Linux

Table 4.3: CPU to GPU System Pairing

4.2 Memory Throughput

As mentioned in the previous chapter, memory throughput tests measure the bandwidth of sending data to and from the GPU with pinned and unpinned memory configurations. The key difference between the pinned and unpinned transfers is that the host memory is either physically pinned (non-swappable) or unpinned (swappable) memory. In the case of pinned memory, the DMA transfer can be initiated directly from the host buffers because the memory cannot be swapped out. Unpinned memory requires a secondary pinned staging buffer which the DMA operates from. The tradeoff is higher throughput performance for the pinned memory at the cost of having to allocate a potentially large amount of pinned memory. Large pinned allocations can impact overall system performance as other applications may be starved for memory and/or forced to page their memory to satisfy the request. Figures 4.1, 4.2, 4.3, and 4.4 show graphs of the memory throughput

performance[1] . These graphs show host-to-device pinned and unpinned performance as well as device-to-host pinned and unpinned performance. Measuring both directions is critical because the GPU operations performed require sending data to and from the GPU device.

The results show that as the size of the data transfer increases, the throughput improves. This is not surprising because the GPU device is connected via a memory bus, either PCIe or directly to the motherboard. Invoking the memory bus transfer has a cost associated with it, and that cost is amortized the larger the data transfer. The results indicate that using a pinned buffer outperforms non-pinned buffers for mid-sized data transfers on all systems tested. For small data transfers, the memory bus instantiation cost is the limiting factor, and thus the cost of an extra memory transfer on the host for unpinned memory does not show through. With larger transfer sizes of 16MB or more, the unpinned throughput values approach the pinned throughput for the PCIe connected devices. With System D, the pinned memory performance is substantially higher than that of the unpinned, even at very large transfer sizes. This is likely due to the difference in the memory interconnect; however, this machine was also running OSX rather than Linux. Due to resource constraints, running the results with Linux on System D's hardware could not be performed to rule out the difference in operating systems.

The results also indicate that a saturation point is reached at around 4096 blocks (2MB) for pinned memory and 32768 blocks (16MB) for unpinned memory. As expected, the pinned memory throughput reaches the saturation point before the unpinned memory transfers. The throughput eventually reaches the saturation point of the x16 PCIe bus at approximately 6GB per second [8]. At the PCIe bus saturation point, the cost of pinned versus unpinned data becomes less pronounced. The saturation point for the non-PCIe connected GeFore 9400M on System D is reached at approximately 4.5 GB per second for pinned memory and approximately 1.5 GB per second for unpinned memory.

[1] Memory throughput measured on System A indicates that as the size of the transfer buffer increases so does the variance. With 128MB buffers, unpinned memory transfers peaked with a variance of 32 $milliseconds^2$ while pinned memory transfers peaked with a variance of 269 $microseconds^2$. The highest variances were measured when data was copied from the device to the host.

These throughput measurements provide an upper bound for the other operations measured on the GPU. Data for the CRC, TEA, and LSM index must be sent to and from the GPU; thus, the maximum throughput is ultimately bound by the round trip memory throughput of the device. For the operations performed here, the maximum throughput is sufficiently high in regard to most network interconnect and disk throughput capabilities.

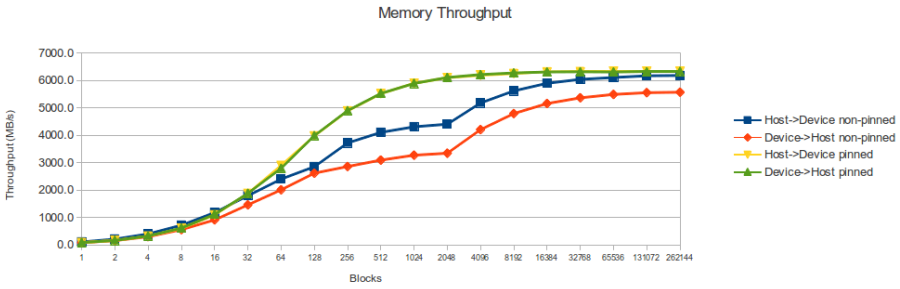

Figure 4.1: Memory Copy Throughput, System A

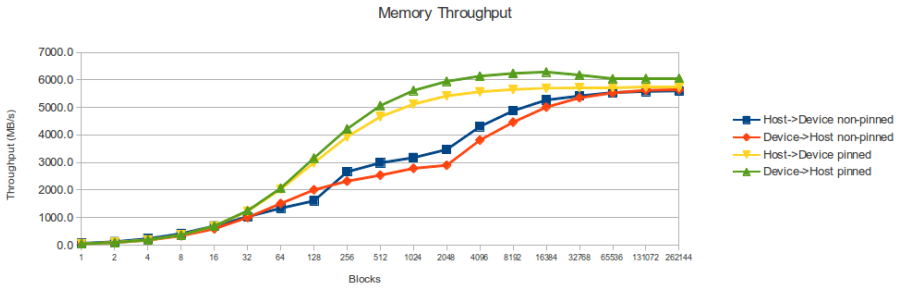

Figure 4.2: Memory Copy Throughput, System B

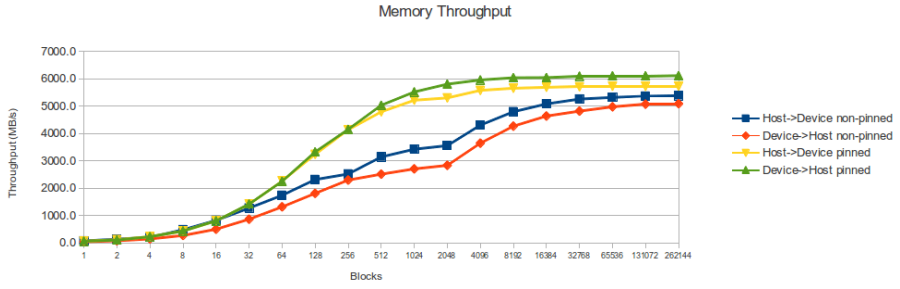

Figure 4.3: Memory Copy Throughput, System C

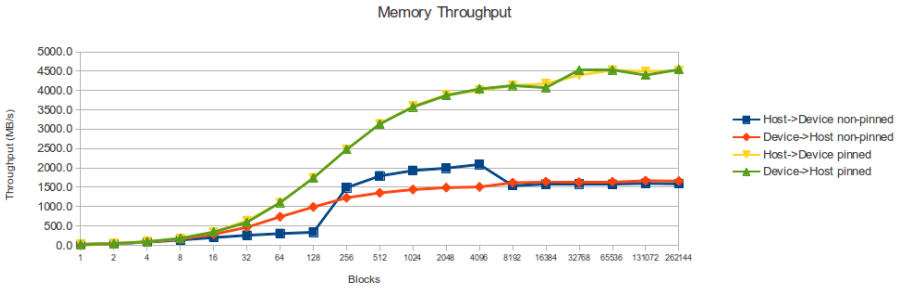

Figure 4.4: Memory Copy Throughput, System D

4.3 CRC32

As mentioned in the previous chapter, the CRC32 results measured the total time and computational time of performing CRC32 calculations with the CPU and GPU. The CPU approaches used a single-threaded and a multi-threaded implementation. The multi-threaded approach utilizes pthreads with one thread created per CPU core. Total time calculations include the full time required for the operation, including the data transfers to and from the GPU device, the instantiation time of the GPU kernels, and the instantiation time of the pthreads. The CRC32 time was calculated using just the time required for performing the CRC32 calculations, not including the data transfers to and from the GPU. GPU times were measured using CUDA event timers, which provide accurate timing measurements for GPU operations. The result graphs for the CRC32 calculations are shown in Figures 4.5, 4.6, 4.7, 4.8, 4.9, and 4.10. Table 4.4 shows throughput numbers for both one and 256K data blocks.

Blocks, Metric	System A	System B	System C	System D	System E	System F
1, 1xCPU	381 MB/s	381 MB/s	363 MB/s	347 MB/s	138 MB/s	193 MB/s
1, NxCPU	3 MB/s	1.8 MB/s	5.1 MB/s	7.5 MB/s	5.2 MB/s	13.0 MB/s
1, GPU total	3.7 MB/s	2.2 MB/s	2.7 MB/s	0.4 MB/s	N/A	N/A
1, GPU crc32	6.9 MB/s	3.6 MB/s	5.5 MB/s	0.5 MB/s	N/A	N/A
256K, 1xCPU	420 MB/s	428 MB/s	397 MB/s	369 MB/s	364 MB/s	350 MB/s
256K, NxCPU	2927 MB/s	4642 MB/s	2938 MB/s	689 MB/s	1417 MB/s	1123 MB/s
256K, GPU total	2705 MB/s	2494 MB/s	2445 MB/s	413 MB/s	N/A	N/A
256K, GPU crc32	19044 MB/s	16696 MB/s	14412 MB/s	506 MB/s	N/A	N/A

Table 4.4: CRC32 Throughput, 512 byte data blocks

The results indicate that CRC32 performance for small data buffers is best performed on a

single-threaded CPU implementation while with large data buffers the GPU and multi-threaded implementations perform better. This is because of the instantiation time required for pthreads and the memory bus data transfers for the GPU. With the PCIe attached devices, the results indicate that the total throughput is limited by the round trip PCIe transfer time. CRC32 throughput approaches 3 GB per second, which happens to be the round trip time observed with the memory transfer measurements. When measuring just the CRC32 calculation time, the PCIe connected GPUs vastly outperform even the multi-threaded CPU implementations for large data sets. Table 4.5 shows the relative percent change for GPU calculations over those of the CPU.

Blocks, Metric	**System A**	**System B**	**System C**	**System D**
1, 1xCPU total	1.0%	0.6%	0.7%	0.1%
1, 1xCPU crc32	1.8%	0.9%	1.5%	0.1%
256K, NxCPU total	92.4%	53.7%	83.2%	59.9%
256K, NxCPU crc32	651%	360%	491%	73.4%

Table 4.5: CRC32 GPU Throughput Percent Change Over CPU, 512 byte data blocks

Table 4.6 shows the relative time spent for GPU CRC calculations on data copies. For large data sets, the PCIe connected cards spend over 80 percent of the total time on data copies. System D, with the GeForce 9400M, has a much lower overall time spent on data copies. This likely is a result of its much lower computational power. The 9400M has only 16 CUDA cores, compared with a minimum of 192 on the PCIe cards, and a clock speed of 450MHz, compared with a minimum of 850MHz on the PCIe cards. This indicates that as GPU power increases, the effect of data transfers becomes more apparent.

	System A	**System B**	**System C**	**System D**
1 Block	46.4%	38.9%	50.1%	20.0%
256K Blocks	85.8%	85.1%	83.0%	18.4%

Table 4.6: CRC32 GPU Percent Time Data Copy, 512 byte data blocks

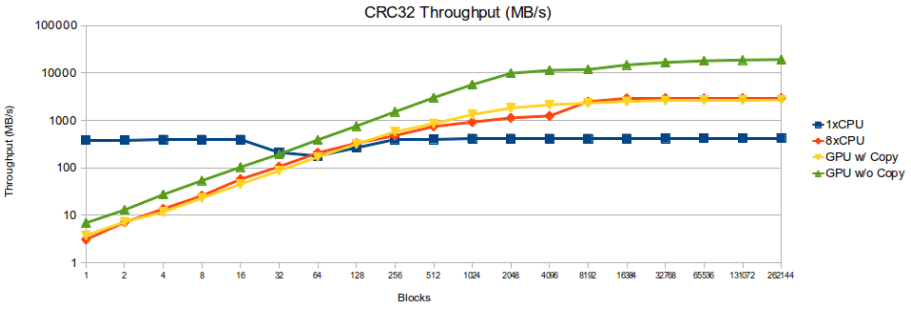

Figure 4.5: CRC32 Throughput Graph, System A

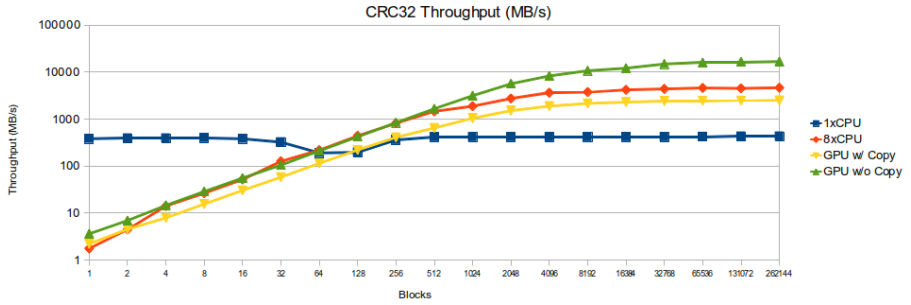

Figure 4.6: CRC32 Throughput Graph, System B

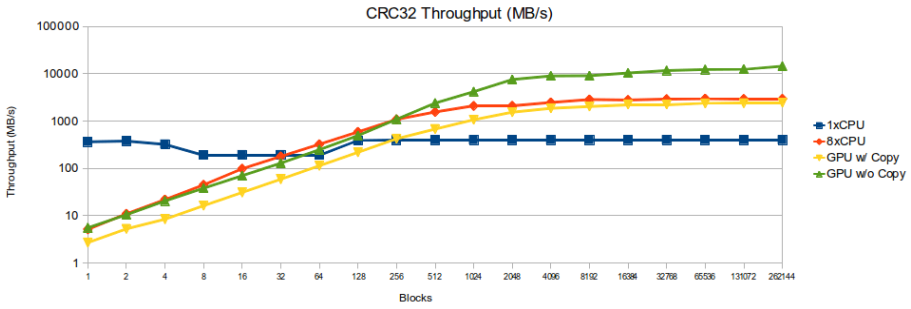

Figure 4.7: CRC32 Throughput Graph, System C

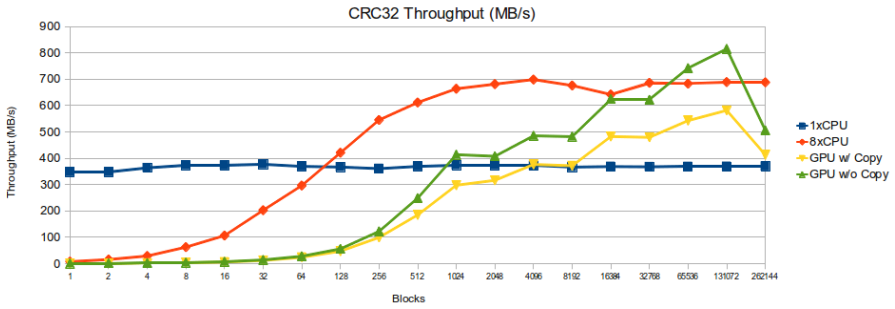

Figure 4.8: CRC32 Throughput Graph, System D

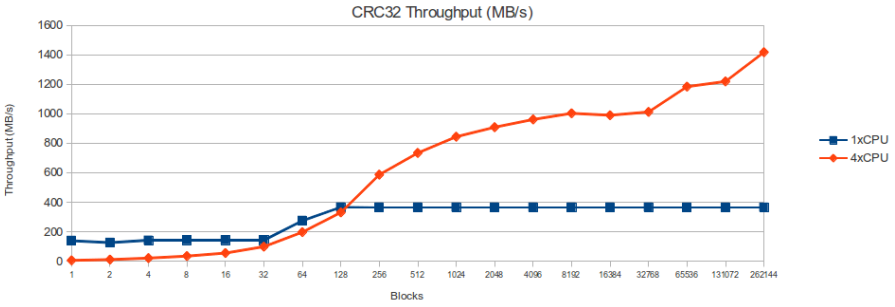

Figure 4.9: CRC32 Throughput Graph, System E

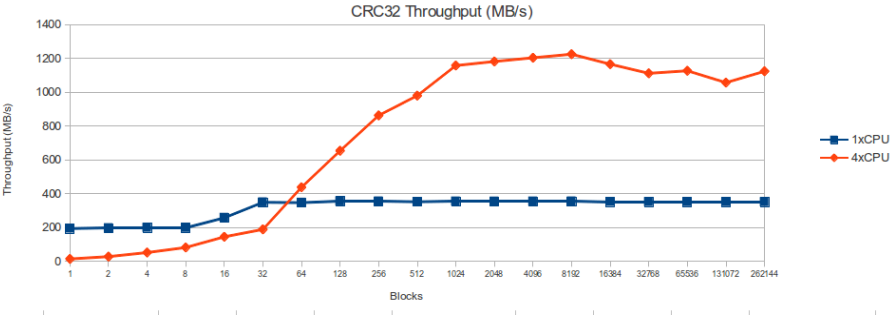

Figure 4.10: CRC32 Throughput Graph, System F

4.4 TEA Encryption

As with the CRC32 implementation, the TEA results measured the total time and computational time of performing the TEA calculations with the CPU and GPU. The CPU approach uses a single-threaded and a multi-threaded implementation. The multi-threaded approach utilizes pthreads with one thread created per CPU core. Total time calculations measure the full time required for the operation, including the data transfer copies to and from the GPU device, the instantiation time of the GPU kernels and the instantiation time of the pthreads. The TEA time was calculated using just the time required for performing the TEA calculations, not including the data transfers to and from the GPU. GPU times were measured using CUDA event timers, which provide accurate timing measurements for GPU operations. The results for the TEA calculations are shown in Figures 4.11, 4.12, 4.13, 4.14, 4.15, and 4.16. Table 4.7 shows throughput numbers for both one and 256K data blocks.

Blocks, Metric	System A	System B	System C	System D	System E	System F
1, 1xCPU	27.3 MB/s	53.7 MB/s	48.9 MB/s	28.1 MB/s	29.5 MB/s	31.1 MB/s
1, NxCPU	2.6 MB/s	0.9 MB/s	0.7 MB/s	5.8 MB/s	2.1 MB/s	55.4 MB/s
1, GPU total	3.5 MB/s	3.1 MB/s	2.98 MB/s	1.9 MB/s	N/A	N/A
1, GPU TEA	39.3 MB/s	23.4 MB/s	37.7 MB/s	13.2 MB/s	N/A	N/A
256K, 1xCPU	65.9 MB/s	55.4 MB/s	50.3 MB/s	28.5 MB/s	47.6 MB/s	3.2 MB/s
256K, NxCPU	358 MB/s	497.8 MB/s	241.6 MB/s	56.6 MB/s	132 MB/s	154 MB/s
256K, GPU total	2006 MB/s	2091 MB/s	1960 MB/s	254.8 MB/s	N/A	N/A
256K, GPU TEA	7041 MB/s	9396 MB/s	8368 MB/s	371.2 MB/s	N/A	N/A

Table 4.7: TEA Throughput, 512 byte data blocks

As with the CRC32 results, the TEA results indicate that the performance for small data buffers is best performed on a single-threaded CPU implementation while with large data buffers the GPU and multi-threaded implementations perform better. This is because of the instantiation time for pthreads and the memory bus data transfers for the GPU. The results again indicate that GPU TEA performance is limited by the round trip memory transfer time. Table 4.8 shows the relative change for GPU calculations over those of the CPU. Unlike the relatively simple CRC32 calculation, the more complex TEA calculation has a vastly higher performance on the GPU with

large data sets, even when taking into account data transfer times.

Blocks, Metric	**System A**	**System B**	**System C**	**System D**
1, 1xCPU total	12.8%	5.8%	6.3%	6.8%
1, 1xCPU TEA	143.9%	43.6%	77.1%	47.0%
256K, NxCPU total	560%	420%	811%	450%
256K, NxCPU TEA	1967%	1888%	3464%	656%

Table 4.8: TEA GPU Throughput Percent Change Over CPU, 512 byte data blocks

Table 4.9 shows the relative time spent for the GPU TEA calculations on data copies. For all data sets, the PCIe connected cards spend over 70 percent of the overall time performing data transfers. As with the CRC32 results, System D, with its GeForce 9400M, has a much lower percentage of time for data copies on large data sets. This is likely a result of its much lower computational power.

	System A	**System B**	**System C**	**System D**
1 Block	91.1%	86.8%	92.1%	85.6%
256K Blocks	71.5%	77.7%	76.6%	31.4%

Table 4.9: TEA GPU Percent Time Data Copy, 512 byte data blocks

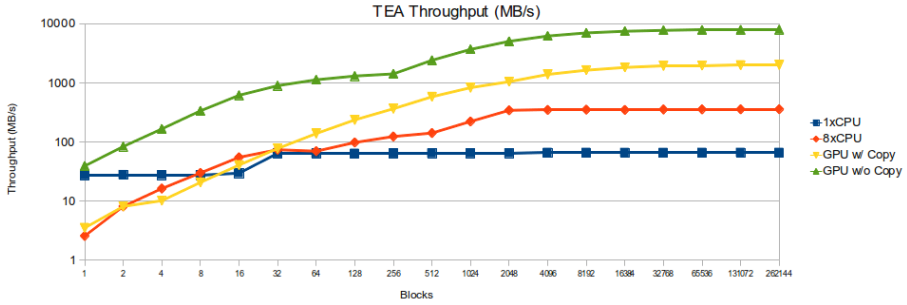

Figure 4.11: TEA Throughput, System A

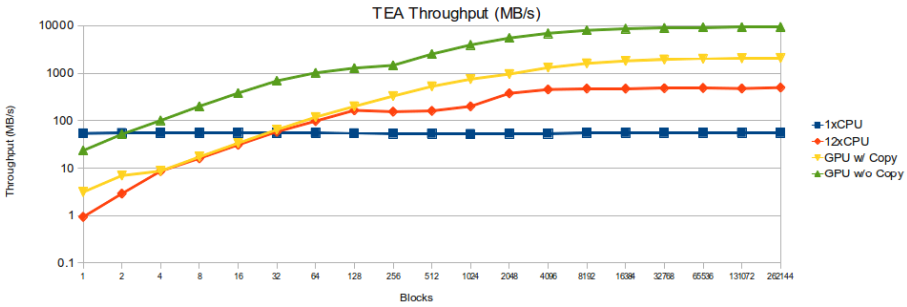

Figure 4.12: TEA Throughput, System B

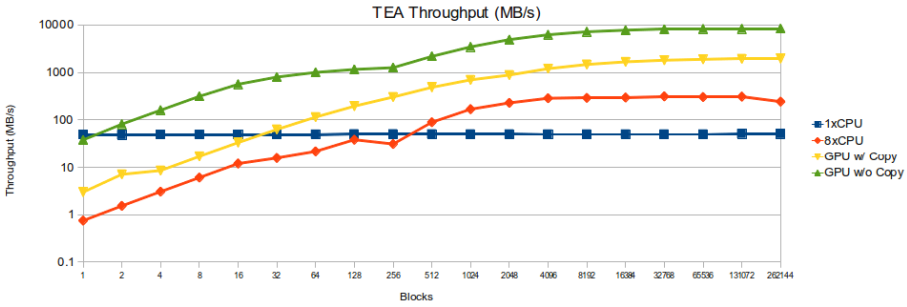

Figure 4.13: TEA Throughput, System C

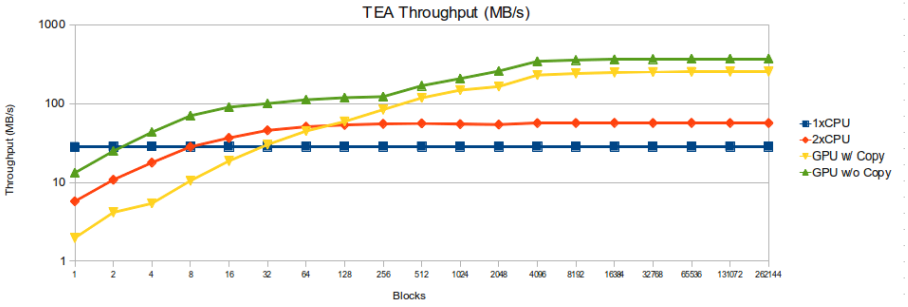

Figure 4.14: TEA Throughput, System D

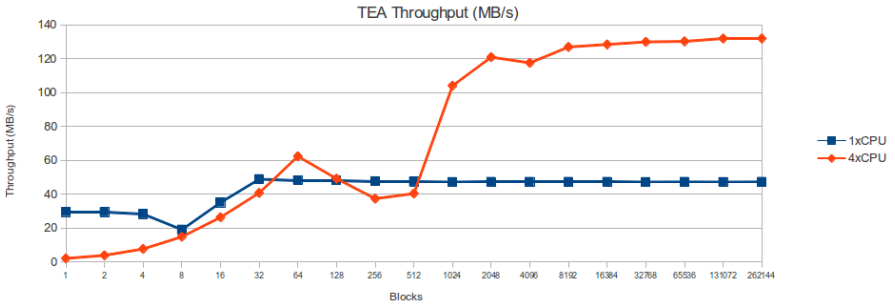

Figure 4.15: TEA Throughput, System E

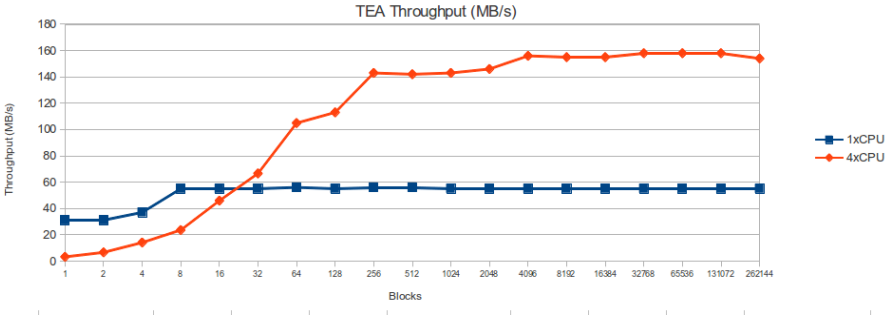

Figure 4.16: TEA Throughput, System F

4.5 Log-Structured Merge Index

The LSM Index implementation has a number of configurable parameters for testing a variety of system permutations. These parameters include encryption of the index, encryption of the log, size of the RB index, and merge threshold. Disabling both encryption parameters gives a baseline for a best case situation; that is, how fast the index system can process requests without the extra overhead of data encryption. Additional encryption options include encryption with the GPU or CPU, in this case no hybrid options were used. Encryption of the log is configurable, allowing for a measurement to be taken of encryption performance solely based on the index merge operations. The size of the RB buffer is variable; however, a fixed size of 64K RB elements was used throughout the benchmarks. The merge threshold was set to be one half of the elements; therefore, a merge would begin when 32K RB elements were within the active tree [2] . For the proof of concept, the in-memory B+ tree contains a root node with 8192 children. Each child node contains a header and 511 index entries as described in the previous chapter. This results in an B+ tree with a depth of one and a maximum of 4186112 entries.

Three operations were tested and measured for the index: insert, remove, and search. These are the basic operations required for an LSM index as described in the previous chapter. Data object sizes varied from 512 bytes up to 256KB. Variable object sizes were used because in a real system, objects are likely to vary in size depending on index use. Also when encrypting the log, the size of the object has a large impact on system performance.

Running without cryptography is tested as a means to get a baseline for the system. It is hard to gauge the impact encryption has on the system without first knowing what the system can sustain without the added cryptographic latency. This provides an upper bound on system performance once encryption is enabled. A simple flag is used for determining whether encryption should be performed. This flag is checked on the log and merge operations. If no encryption is to

[2] Because the index is memory resident, preliminary results indicate that tweaking the merge threshold within a reasonable range does not have a large impact on system performance. In order to keep measurements consistent, a fixed size threshold was used throughout the tests.

be used, the operations continue without first encrypting the data.

Cryptography was measured for index operations utilizing the same code as with the TEA encryption described in the previous chapter for both the CPU and GPU. The items being encrypted are the B+ tree during merge operations and the log on each insert and remove operation. Depending on configuration, search operations may require decryption of the B+ tree nodes and/or the object data within the log. Log encryption was disabled for one run of each operation and enabled for another. This allowed a measurement to be taken with just the impact of encryption on the merge operation. Encryption was tested using both the CPU and GPU, and no hybrid encryption schemes were performed.

4.5.1 Insert

Insert parameters were measured by inserting 256K elements, of variable size and with random keys, into the index. This allowed for testing merge operations, as a number of merges are required to satisfy the inserts. All of the 256K updates could not be buffered within the RB tree. Insert encryption was measured for both the log and merge process. Use of log encryption is configurable; thus, results were taken for both encryption of the log and without encryption. Measuring with and without the log encryption results in a large difference within the system since using log encryption requires an operation on each insert into the log. Measurements without log encryption provide a baseline of the system performance when only performing encryption within the merge operation. Two base system measurements were taken, one which did not encrypt data within the log and another that encrypted log updates before placing them into the index. These two metrics were taken to provide a measurement of the base log overhead.

The graphs seen in Figures 4.17, 4.18, and 4.19 contain results using the LSM index with and without encryption of the index only; no encryption of the log was done with these results. Table 4.10 contains relative percent change over the base system performance for index encryption using the CPU and GPU.

These results show that the GPU index encryption is within 60 percent of the base system

performance for all systems tested. When using CPU index encryption, the performance is near 10 percent of the base system performance. These results are not surprising given the encryption results observed for TEA. The minimal data size encryption which occurs for a merge operation is 512KB, resulting from 32K red-black elements multiplied by 128 bits each. With this size data set, the GPU outperforms the CPU significantly.

	System A	**System B**	**System C**
CPU Index Encryption	10.7%	11.1%	11.0%
GPU Index Encryption	66.7%	65.7%	67.7%

Table 4.10: LSM CPU/GPU Insert Index Encryption Relative to No Encryption

The graphs seen in Figures 4.20, 4.21, and 4.22 contain results using the LSM index with encryption of the index and log. These results are significantly lower than the unencrypted log numbers because each insert operation must first encrypt the data object being inserted. It is not surprising that, with small data objects, the CPU outperforms the GPU [3] . This can be seen from the TEA results earlier. As the data objects increase in size, the GPU begins to outperform the CPU significantly. Table 4.11 shows insert operations per second for 1, 16, 128, and 512 blocks respectively. For table clarity, host refers to the CPU while device refers to the GPU.

	System A	**System B**	**System C**	**System E**	**System F**
1 Block, Host	29412	24898	22298	21161	24945
1 Block, Device	7681	6325	6749	N/A	N/A
16 Blocks, Host	7052	5937	5305	4987	5914
16 Blocks, Device	4797	3994	4058	N/A	N/A
128 Blocks, Host	1048	869	788	743	871
128 Blocks, Device	3597	3087	3035	N/A	N/A
512 Blocks, Host	264	219	201	189	218
512 Blocks, Device	2245	2034	1899	N/A	N/A

Table 4.11: LSM Insert, Object Encryption, Operations/Second, 512 byte data blocks

Table 4.12 contains the relative performance gain when using the GPU on data objects of 1 block and 512 blocks. As expected, the greater the data object size, the higher the performance gain when using the GPU.

[3] Overall GPU performance could be improved by coalescing multiple insert operations into a single encryption request. This measurement provides a worst case metric.

	System A	System B	System C
1 Block	26%	25%	30%
512 Blocks	850%	929%	945%

Table 4.12: LSM GPU Insert Object Encryption Operations per Second Relative to CPU

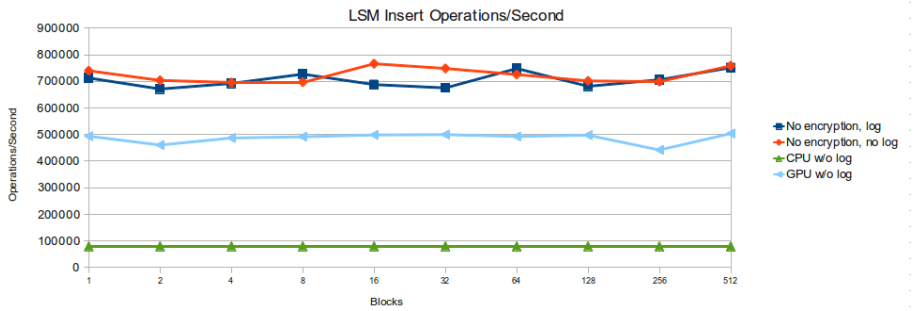

Figure 4.17: LSM Insert, CPU/GPU w/o log encryption, System A

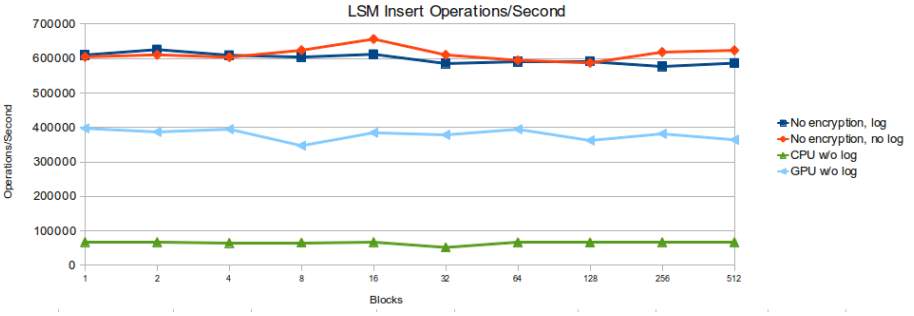

Figure 4.18: LSM Insert, CPU/GPU w/o log encryption, System B

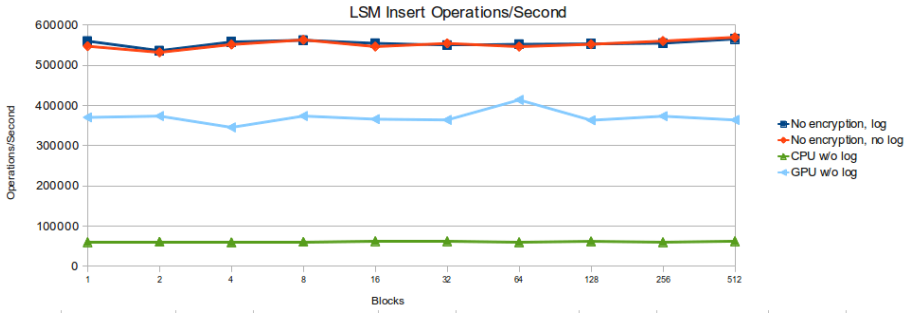

Figure 4.19: LSM Insert, CPU/GPU w/o log encryption, System C

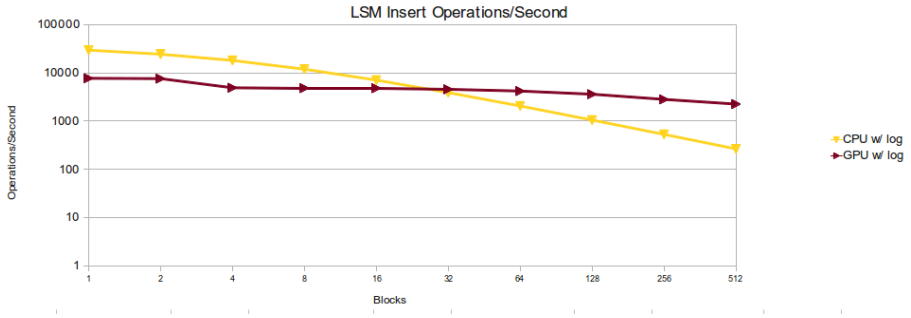

Figure 4.20: LSM Insert, CPU/GPU with log encryption, System A

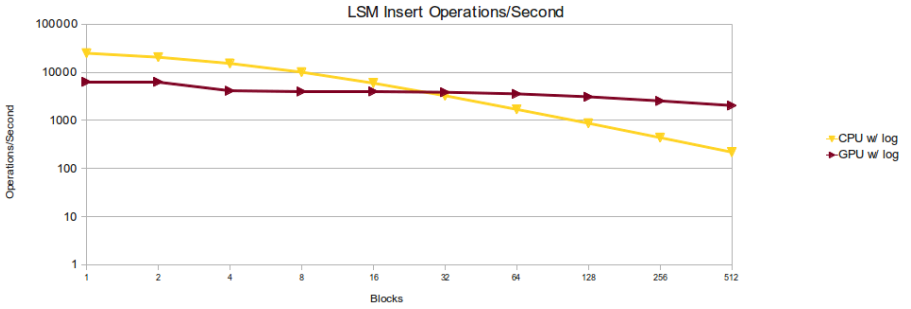

Figure 4.21: LSM Insert, CPU/GPU with log encryption, System B

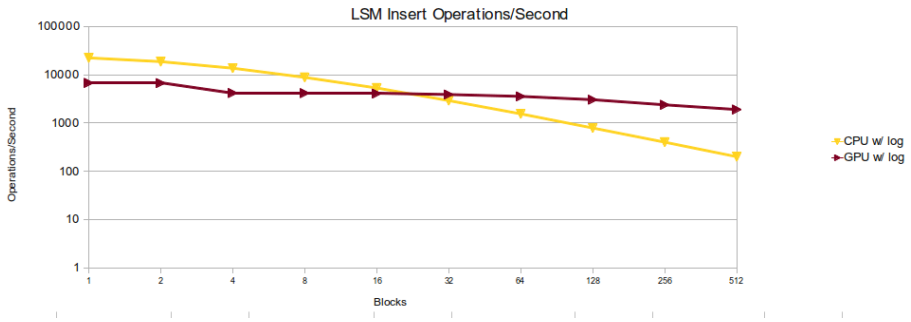

Figure 4.22: LSM Insert, CPU/GPU with log encryption, System C

4.5.2 Remove

Remove parameters were measured by removing 256K elements, with random keys, from the index. This allowed for testing merge operations as a number of merges are required to satisfy the removes. All of the updates could not be buffered within the RB index. The remove operation is unique in that it effectively functions as a negative insert. RB elements are allocated and used for insertion into the tree as markers for the impending removal of this item from the index. Therefore, the log aspect of a remove is very small, just 128 bits for the key and value pair being removed. There is no data object associated with the remove as there is with an insert operation. This key difference with the insert operations leads to much greater impact in regard to the log as the cost of PCIe transfers to the GPU cannot be as easily amortized [4] .

	System A	**System B**	**System C**	**System E**	**System F**
No encryption	681610	555368	609062	459916	626807
Host w/o log	79639	60351	66636	56760	67216
Host w/ log	78232	60184	66791	56840	52225
Device w/o log	503041	359639	379434	N/A	N/A
Device w/ log	7585	6739	6421	N/A	N/A

Table 4.13: Remove Operations/Second

Results from the remove can be seen in Table 4.13. For table clarity, host refers to the CPU while device refers to the GPU. Only results using object sizes of one block were taken because the remove operation only logs the key-value pair being removed, not the object. Results for larger objects would be the same as those for small objects. For remove operations, the CPU has significantly higher performance than the GPU with object encryption. This performance differential is constant regardless of size, as the only object data logged is the 128 bits used for the key-value pair. Without log encryption, the performance is the same as that of the insert because a remove is nearly identical to an insert operation.

[4] Overall GPU performance could be improved by coalescing multiple remove operations into a single encryption request. This measurement provides a worst case metric.

4.5.3 Search

Search parameters were measured by searching 256K elements, of variable size and with random keys, within the index. Search operations are different from the insert and remove in that no merge processes are required for a search. The two RB indexes are searched and then the B+ tree. For measurement purposes, search operations are configured to always require decryption of a B+ tree index node and produce an index hit. This was done to provide a worst case performance measurement on search operations. Therefore, searching of the B+ tree requires a decryption of its 8KB index nodes. After decryption of the index, configurable object decryption from the log is performed. Search results can be seen in figures 4.23, 4.24, and 4.25.

Table 4.14 contains search results without object decryption. These results show the overhead associated with index decryption only, while also demonstrating index miss performance measurements, as no object data is decrypted. The base system search performance is significantly higher than both the CPU and GPU index decryption. This is because the base system performance does not do any decryption of the index. Thus, this is baseline measurement of the maximum search operations within the system. CPU performance is higher than the GPU without object decryption because each index node is 8KB in size. As can be seen from the TEA results, the CPU outperforms the GPU with 8KB data buffers [5] .

	System A	**System B**	**System C**	**System E**	**System F**
Base Search, no Index Decryption	1348082	1327937	1047633	722349	1245139
CPU Index Decryption	8933	6977	6465	5923	7489
GPU Index Decryption	5429	4324	4322	N/A	N/A

Table 4.14: LSM Search without Object Decryption, Operations/Second

These results are consistent with the others seen in previous measurements in that, for small data buffers, the CPU outperforms the GPU while, as the size of the data buffer increases the GPU outperforms the CPU. Table 4.16 shows the relative percent gain by using the GPU for decryption over the CPU. Table 4.15 shows search operations per second for 1, 16, 128, and 512

[5] Techniques such as coalescing multiple B+ tree nodes for decryption could be used to improve the GPU index decryption rate. These measurements provide a worst case metric for index decryption.

blocks respectively. For table clarity, host refers to the CPU while device refers to the GPU.

	System A	**System B**	**System C**	**System E**	**System F**
1 Block, Host	8400	6589	5912	5576	7036
1 Block, Device	3460	2722	2767	N/A	N/A
16 Blocks, Host	4481	3510	3178	2974	3751
16 Blocks, Device	2727	2209	2174	N/A	N/A
128 Blocks, Host	998	782	709	664	836
128 Blocks, Device	2298	1896	1853	N/A	N/A
512 Blocks, Host	266	209	190	178	224
512 Blocks, Device	1435	1242	1186	N/A	N/A

Table 4.15: LSM Search, Object Decryption, Operations/Second, 512 byte data blocks

	System A	**System B**	**System C**
1 Block	41%	41%	47%
512 Blocks	539%	594%	624%

Table 4.16: LSM GPU Search Object Decryption Relative to CPU

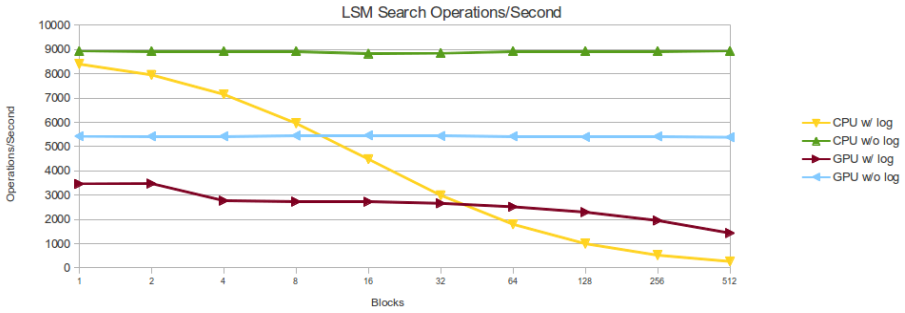

Figure 4.23: LSM Search, System A

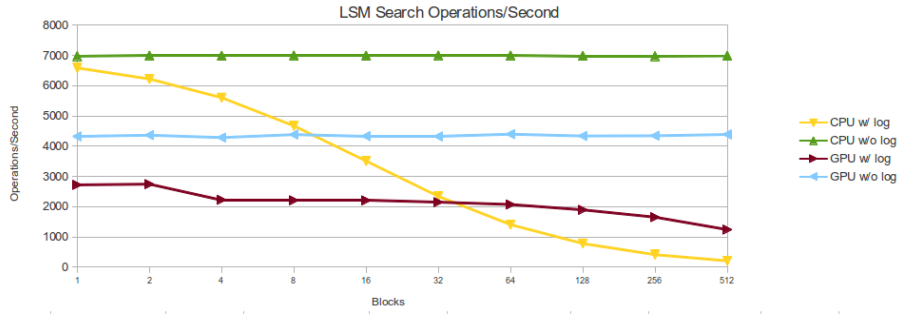

Figure 4.24: LSM Search, System B

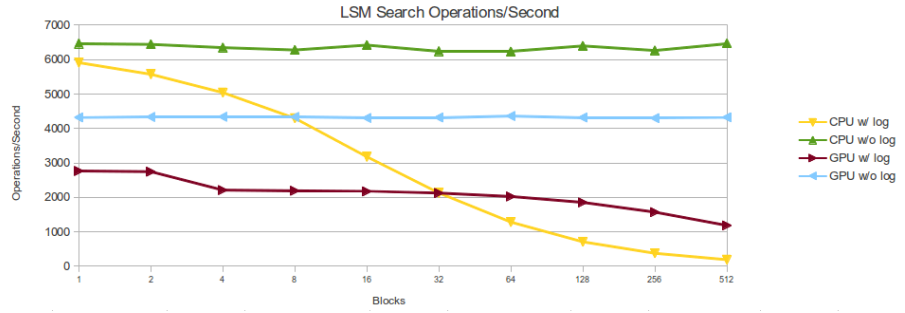

Figure 4.25: LSM Search, System C

Chapter 5

Conclusions

The results shown in the work provided here are consistent with other works in demonstrating that for certain applications offloading work to a GPU can improve overall performance. Since GPUs were originally designed for graphics processing, their SIMD architecture comes with some limitations for general integration into systems. GPUs excel at performing single operations on many data blocks such as pixel graphics effects, hashes, and cryptography. The results presented here show that offloading hash and encryption operations to a GPU can result in a significant performance gain.

The performance picture as a whole, taking into account the time required to transport data back and forth between the host and device, shows that the cost of data movements can significantly reduce the overall performance of task offloading to the GPU. This performance hit could be reduced or removed if the general purpose chips were moved closer to main system memory and the CPU. Indeed, a similar approach has been planned by Intel through its Larrabee [16] project. Alternatively, multi-core GPU like chips could be moved closer to the storage, performing encryption on the backend. Along with architecture changes, data blocks could be cached in the GPU device memory, reducing the required number of buffer transfers for subsequent operations.

With current architectures, the results indicate that it is feasible and beneficial to re-purpose the available horse power of GPUs within storage systems for tasks such as hashes and encryption. These results show performance improvements for encryption of the LSM tree index when assisted by the GPU and suggest the need for a hybrid approach. Encryption of small data blocks on the

CPU outperforms the GPU due to the cost associated with PCIe transfers. This research shows that by harnessing the power of otherwise unused GPUs, at rest data encryption can be provided with minimal overall system performance impact and zero cost increase for the majority of systems which already contain GPUs. Using TEA with large data blocks, GPU encryption performance approaches 2 GB per second. This speed is sufficient to keep pace with many modern data links and disks. The impact of GPU encryption would not be the bottleneck in most systems. Using AES encryption would provide even higher throughput than TEA [3].

Overall, this work shows that there is vast potential for GPU assistance in storage systems. Many storage-related tasks fit the GPU architecture well, including the hash computations and encryption explored here. In most cases, storage systems could benefit from added performance at a zero cost increase as many would already posses the GPU hardware required. The results demonstrated in this work could also be transferred to other systems such as GPU-assisted network encryption. It is certain GPUs and/or their descendants, albeit likely integrated into the core of the system or CPU, will play a major role in future computing systems.

Chapter 6

Future Work and Extensions

There is exciting potential for future work in integrating the LSM index into a fully functional NoSQL system such as Cassandra [1]. Cassandra would be an ideal candidate for such an integration for several reasons. First, and most importantly, Cassandra is an open source project, so access to the source is not problematic. Second, Cassandra is a well known and actively maintained project with a large developer and user base. Perhaps the biggest challenges imposed by integrating this work into Cassandra would be that Cassandra is written in Java, requiring the use of Java native C, or a port of this work to Java. However, third party Java libraries for OpenCL [15] and CUDA [10] do exist.

The TEA encryption algorithm was used for testing the performance trade offs between the CPU and GPU. TEA was chosen for its simplicity; however, TEA is a slower encryption algorithm than AES [25]. It would be worthwhile and interesting to examine the impact of AES and/or DES with encryption in the LSM index. The GPU would likely still have an advantage for large data sizes, but the performance gap may not be as significant [26].

Completing the proof of concept and implementing the disk access routines would provide results indicating the encryption impact on a system with active disks. It may be the case that the performance improvement of GPU encryption could go unnoticed if the disks are the system bottleneck, and likely this would indeed be the case for most low-end storage systems.

Future work in incorporating GPUs into storage systems will likely continue as more research is done in this field. Work presented in [24] researches the integration of a GPU into a distributed

storage system for performing hashes for content addressing and other GPU related tasks. The work presented here, along with that in [24], shows that there is exciting future potential for GPUs within storage systems. The added computational power provided by GPUs opens many possibilities for content addressing, data de-duplication, and encryption, to name a few. These are bound to become more prevalent as advances continue in multi-core architectures.

Bibliography

[1] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.

[2] Bingsheng He, Wenbin Fang, Naga K. Govindaraju, Qiong Luo, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In International Conference on Parallel Architectures and Compilation Techniques, 2008.

[3] Brandon P. Luken, Ming Ouyang, and Ahmed H. Desoky. AES and DES Encryption with GPU. In International Conference on Parallel and Distributed Computing and Communication Systems, pages 67–70, 2009.

[4] Charles B. Morrey III and Dirk Grunwald. Content-Based Block Caching. In IEEE NASA Goddard Conference on Mass Storage Systems and Technologies, 2006.

[5] R. L. Cloud, M. L. Curry, H. L. Ward, A. Skjellum, and P. Bangalore. Accelerating lossless data compression with gpus. Inquiro, 3:1–8, 2011.

[6] Erik Sintorn and Ulf Assarsson. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In Parallel and Distributed Computing Archive, volume 68, 2008.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemaway, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In Conference on Usenix Symposium on Operating Systems Design and Implementation, volume 7, pages 205–219, 2006.

[8] IBM. Introduction to PCI Express, 2004. http://www.redbooks.ibm.com/abstracts/tips0456.html.

[9] Jason Sanders and Edward Kandrot. CUDA by Example. Addison-Wesely, 2011.

[10] jcuda.org. Java CUDA. http://www.jcuda.de/.

[11] Jeff Gilchrist. Parallel Data Compression With Bzip2.

[12] Jens Krueger, Martin Grund, Ingo Jaeckel, Dr. Alexander Zeier, and Hasso Plattner. Applicability of GPU Computing for Efficient Merge in In-Memory Databases. In International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures, 2011.

[13] Jianlong Zhong and Bingsheng He. GPU-Assisted Buffer Management. In International Conference on Computational Science, 2011.

[14] Jim Kukunas and James Device. GPGPU Parallel Merge Sort Algorithm.

[15] jocl.org. Java OpenCL. http://www.jocl.org/.

[16] Larry Seiler, Doug Carmen, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. In In SIGGRAPH '08: ACM SIGGRAPH 2008 papers, pages 1–15. ACM, 2008.

[17] Mathew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. Arbitrary Dimension Reed-Solomon Coding and Decoding for Extended RAID on GPUs. In Petascale Data Storage Workshop, 2008.

[18] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In ACM Transactions on Computer Systems, volume 10, 1992.

[19] Nadathur Satish, Mark Harris, and Michael Garland. Designing Efficent Sorting Algorithms for Manycore GPUs. In IEEE International Parallel and Distributed Processing Symposium, 2009.

[20] NVIDIA. NVIDIA CUDA. http://www.nvidia.com/object/cuda.

[21] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[22] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree), 1996.

[23] Rishabh Mukherjee, M. Suhail Rehman, Kishore Kothapalli, P. J. Narayanan, and Kannan Srinathan. Fast, Scalable, and Secure Encryption on the Gpu.

[24] Samer Al Kiswany, Abdullah Gharaibeh, and Matei Ripeanu. GPUs as Storage System Accelerators. In ACM International Symposium on High Performance Distributed Computing, 2010.

[25] Soren Rinne. Performance Analysis of Contemporary Light-weight Cryptographic Algorithms on a Smart Card Microcontroller. Bachelor's thesis, Ruhr-Universitat Bochum, 2007.

[26] Svetlin A. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In IEEE International Conference on Signal Processing and Communications, 2007.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. McGraw-Hill Book Company, Second Edition edition, 2001.

[28] Wikipedia. CUDA, March 2012. http://en.wikipedia.org/wiki/CUDA.

[29] Wikipedia. NoSQL, March 2012. http://en.wikipedia.org/wiki/NoSQL.

Appendix A

Appendix A: Source Code

Listing A.1: Unpinned Data Transfer Code

```
static void ges_throughputUnpinned(unsigned blockSize, unsigned power)
{
    cudaEvent_t start;
    cudaEvent_t end;
    unsigned index;
    unsigned size;

    GES_INFO("Starting host nonpinned memory test...");

    size = blockSize;
    for (index = 0; index < power; index++) {
        unsigned char *hostBuffer;
        unsigned char *devBuffer;
        float elapsedTime;
        unsigned iterations;

        ges_gpuEventCreate(&(start));
        ges_gpuEventCreate(&(end));

        hostBuffer = malloc(size);
        if (!hostBuffer) {
            GES_ERR("Failed to malloc:%d bytes", size);
            exit(EXIT_FAILURE);
        }

        ges_gpuDevAlloc((void **)&(devBuffer), size);

        /* Compute the average over GES_THROUGHPUT_ITERATIONS. Host->Device. */
        elapsedTime = 0;
        for (iterations = 0;
             iterations < GES_THROUGHPUT_ITERATIONS; iterations++) {
```

```
            ges_gpuEventRecord((void *)&(start));
            ges_gpuMemCpyToDev(devBuffer, hostBuffer, size);
            ges_gpuEventRecord((void *)&(end));

            ges_gpuThreadSynchronize();
            elapsedTime = elapsedTime +
                        ges_gpuEventElapsedTime((void *)&(start), (void *)&(end));
        }

        elapsedTime = (elapsedTime / GES_THROUGHPUT_ITERATIONS); /* average */
        GES_INFO("Hd time(ms)=%.3f bs=%d blocks=%d xfer=%dB, throughput=%.3fMB/s",
                elapsedTime, blockSize, (size/blockSize), size,
                GES_BYTES_TO_MB(size)/(elapsedTime/1000));

        /* Compute the average over GES_THROUGHPUT_ITERATIONS. Device->Host. */
        elapsedTime = 0;
        for (iterations = 0;
             iterations < GES_THROUGHPUT_ITERATIONS; iterations++) {
            ges_gpuEventRecord((void *)&(start));
            ges_gpuMemCpyToHost(hostBuffer, devBuffer, size);
            ges_gpuEventRecord((void *)&(end));

            ges_gpuThreadSynchronize();
            elapsedTime = elapsedTime +
                        ges_gpuEventElapsedTime((void *)&(start), (void *)&(end));
        }

        elapsedTime = (elapsedTime / GES_THROUGHPUT_ITERATIONS); /* average */
        GES_INFO("Dh time(ms)=%.3f bs=%d blocks=%d xfer=%dB, throughput=%.3fMB/s",
                elapsedTime, blockSize, (size/blockSize), size,
                GES_BYTES_TO_MB(size)/(elapsedTime/1000));

        free(hostBuffer);
        ges_gpuDevFree(devBuffer);

        size *= 2; /* Increase size by a power of two. */
    }
}
```

Listing A.2: Pinned Data Transfer Code

```
static void ges_throughputPinned(unsigned blockSize, unsigned power)
{
    cudaEvent_t start;
    cudaEvent_t end;
    unsigned index;
    unsigned size;
```

```
GES_INFO("Starting host pinned memory test...");

size = blockSize;
for (index = 0; index < power; index++) {
    unsigned char *hostBuffer;
    unsigned char *devBuffer;
    float elapsedTime;
    unsigned iterations;

    ges_gpuEventCreate(&(start));
    ges_gpuEventCreate(&(end));

    ges_gpuHostAlloc((void **)&(hostBuffer), size);
    ges_gpuDevAlloc((void **)&(devBuffer), size);

    /* Compute the average over GES_THROUGHPUT_ITERATIONS. Host->Device. */
    elapsedTime = 0;
    for (iterations = 0; iterations < GES_THROUGHPUT_ITERATIONS; iterations++) {
        ges_gpuEventRecord((void *)&(start));
        ges_gpuMemCpyToDev(devBuffer, hostBuffer, size);
        ges_gpuEventRecord((void *)&(end));

        ges_gpuThreadSynchronize();
        elapsedTime = elapsedTime +
                        ges_gpuEventElapsedTime((void *)&(start), (void *)&(end));
    }

    elapsedTime = (elapsedTime / GES_THROUGHPUT_ITERATIONS); /* average */
    GES_INFO("Hd time(ms)=%.3f bs=%d blocks=%d xfer=%dB, throughput=%.3fMB/s",
              elapsedTime, blockSize, (size/blockSize), size,
              GES_BYTES_TO_MB(size)/(elapsedTime/1000));

    /* Compute the average over GES_THROUGHPUT_ITERATIONS. Device->Host. */
    elapsedTime = 0;
    for (iterations = 0;
          iterations < GES_THROUGHPUT_ITERATIONS; iterations++) {
        ges_gpuEventRecord((void *)&(start));
        ges_gpuMemCpyToHost(hostBuffer, devBuffer, size);
        ges_gpuEventRecord((void *)&(end));

        ges_gpuThreadSynchronize();
        elapsedTime = elapsedTime +
                        ges_gpuEventElapsedTime((void *)&(start), (void *)&(end));
    }

    elapsedTime = (elapsedTime / GES_THROUGHPUT_ITERATIONS); /* average */
```

```
        GES_INFO("Dh time(ms)=%.3f bs=%d blocks=%d xfer=%dB, throughput=%.3fMB/s",
                 elapsedTime, blockSize, (size/blockSize), size,
                 GES_BYTES_TO_MB(size)/(elapsedTime/1000));

        ges_gpuHostFree(hostBuffer);
        ges_gpuDevFree(devBuffer);

        size *= 2; /* Increase size by a power of two. */
    }
}
```

Listing A.3: Serial CPU CRC32

```
static void ges_crc32CPU(unsigned blockSize, unsigned power)
{
    unsigned index;
    unsigned size;

    GES_INFO("Starting crc32 on 1xCPU...");

    size = blockSize;
    for (index = 0; index < power; index++) {
        unsigned char *hostBuffer;
        float elapsedTime;
        unsigned iterations;

        hostBuffer = (unsigned char *)malloc(size);

        /* Compute the average over GES_CRC32_ITERATIONS. CPU crc32. */
        elapsedTime = 0;
        for (iterations = 0; iterations < GES_CRC32_ITERATIONS; iterations++) {
            struct timeval start, end;
            unsigned block = 0;
            unsigned byte;
            unsigned offset;
            unsigned crc = 0xffffffff;

            if (gettimeofday(&(start), NULL)) {
                GES_ERR("gettimeofday failed");
                exit(EXIT_FAILURE);
            }

            while (block < (size / blockSize)) {
                for (offset = 0; offset < blockSize; offset++) {
                    byte = (block * blockSize) + offset;
                    crc = (crc >> 8) ^ table[(crc ^ hostBuffer[byte]) & 0xff];
                }
```

```
            crc = (crc ^ 0xffffffff);
            block++;
        }

        if (gettimeofday(&(end), NULL)) {
            GES_ERR("gettimeofday failed");
            exit(EXIT_FAILURE);
        }
        elapsedTime = (float)(elapsedTime +
                        (float)(((end.tv_sec * 1000000) + end.tv_usec) -
                                ((start.tv_sec * 1000000) + start.tv_usec)));
    }

    elapsedTime = (elapsedTime / GES_CRC32_ITERATIONS); /* average */
    GES_INFO("time(ms)=%.3f bs=%d blocks=%d xfer=%dB throughput=%.3fMB/s",
            (elapsedTime/1000), blockSize, (size/blockSize), size,
            GES_BYTES_TO_MB(size/(elapsedTime/1000000)));

    free(hostBuffer);

    size *= 2; /* increase size by power of two. */
  }
}
```

Listing A.4: Threaded CPU CRC32

```
static void ges_crc32CPUThread(unsigned blockSize, unsigned power)
{
  unsigned index;
  unsigned size;
  unsigned cpuCnt;

  cpuCnt = ges_cpuCnt();
  ges_crc32State.cpuCnt = cpuCnt;
  GES_INFO("Starting crc32 on %dxCPU...", cpuCnt);

  size = blockSize;
  for (index = 0; index < power; index++) {
    unsigned char *hostBuffer;
    float elapsedTime;
    unsigned iterations;

    hostBuffer = (unsigned char *)malloc(size);

    /* Compute the average over GES_CRC32_ITERATIONS. CPU crc32. */
    elapsedTime = 0;
```

```
        for (iterations = 0; iterations < GES_CRC32_ITERATIONS; iterations++) {
            struct timeval start, end;
            unsigned threads = cpuCnt;
            pthread_t thread[cpuCnt];
            ges_crc32ThreadInfo_st threadInfo[cpuCnt];

            if (gettimeofday(&(start), NULL)) {
                GES_ERR("gettimeofday failed");
                exit(EXIT_FAILURE);
            }

            ges_crc32State.blockSize = blockSize;
            ges_crc32State.size = size;
            for (threads = 0; threads < cpuCnt; threads++) {
                threadInfo[threads].tid = threads;
                threadInfo[threads].data = hostBuffer;
                pthread_create(&(thread[threads]), NULL,
                               ges_crc32Thread, &(threadInfo[threads]));
            }

            for (threads = 0; threads < cpuCnt; threads++) {
                pthread_join(thread[threads], NULL);
            }

            if (gettimeofday(&(end), NULL)) {
                GES_ERR("gettimeofday failed");
                exit(EXIT_FAILURE);
            }
            elapsedTime = (float)(elapsedTime +
                          (float)(((end.tv_sec * 1000000) + end.tv_usec) -
                                  ((start.tv_sec * 1000000) + start.tv_usec)));
        }

        elapsedTime = (elapsedTime / GES_CRC32_ITERATIONS); /* average */
        GES_INFO("time(ms)=%.3f bs=%d blocks=%d xfer=%dB throughput=%.3fMB/s",
                 (elapsedTime/1000), blockSize, (size/blockSize), size,
                 GES_BYTES_TO_MB(size/(elapsedTime/1000000)));

        free(hostBuffer);

        size *= 2; /* increase size by power of two. */
    }
}


void *ges_crc32Thread(void *threadInfoHandle)
{
```

```
    unsigned block;
    unsigned offset;
    unsigned crc = 0xffffffff;
    ges_crc32ThreadInfo_st *threadInfo =
        (ges_crc32ThreadInfo_st *)threadInfoHandle;

    block = threadInfo->tid;
    while (block < (ges_crc32State.size / ges_crc32State.blockSize)) {
        for (offset= 0; offset < ges_crc32State.blockSize; offset++) {
            unsigned byte = (block * ges_crc32State.blockSize) + offset;
            crc = (crc >> 8) ^ table[(crc ^ threadInfo->data[byte]) & 0xff];
        }

        crc = (crc ^ 0xffffffff);
        block += ges_crc32State.cpuCnt;
    }

    return (NULL);
}
```

Listing A.5: GPU CRC32

```
static void ges_crc32GPU(unsigned blockSize, unsigned power)
{
    cudaEvent_t tot_start;
    cudaEvent_t tot_end;
    cudaEvent_t crc_start;
    cudaEvent_t crc_end;
    unsigned index;
    unsigned size;

    GES_INFO("Starting crc32 on GPU...");

    size = blockSize;
    for (index = 0; index < power; index++) {
        float totalTime;
        float crcTime;
        unsigned iterations;


        /* Compute the average over GES_CRC32_ITERATIONS. CPU crc32. */
        totalTime = 0;
        crcTime = 0;
        for (iterations = 0; iterations < GES_CRC32_ITERATIONS; iterations++) {
            unsigned char *buf;
            int *devTable;
            int *devBuf;
```

```
    int *devCrc;

    ges_gpuHostAlloc((void **)&(buf), size);

    ges_gpuEventCreate(&(tot_start));
    ges_gpuEventCreate(&(tot_end));
    ges_gpuEventCreate(&(crc_start));
    ges_gpuEventCreate(&(crc_end));

    ges_gpuEventRecord((void *)&(tot_start));

    ges_gpuDevAlloc((void **)&(devTable),
                    sizeof(unsigned) * GES_CRC32_TABLE_SIZE);
    ges_gpuDevAlloc((void **)&(devBuf), sizeof(char) * size);
    ges_gpuDevAlloc((void **)&(devCrc), sizeof(int) * (size/blockSize));

    ges_gpuMemCpyToDev(devTable, table,
                       sizeof(unsigned) * GES_CRC32_TABLE_SIZE);
    ges_gpuMemCpyToDev(devBuf, buf, sizeof(char) * size);

    ges_gpuEventRecord((void *)&(crc_start));
    ges_crc32Kernel<<<min((size/blockSize), GES_GPU_MAX_BLOCK),
       GES_GPU_MAX_THREAD>>>(devTable, devBuf, blockSize, size, devCrc);
    ges_gpuCheckErr();
    ges_gpuEventRecord((void *)&(crc_end));

    ges_gpuMemCpyToHost(buf, devBuf, sizeof(char) * size);

    ges_gpuEventRecord((void *)&(tot_end));

    ges_gpuThreadSynchronize();
    totalTime = totalTime +
                    ges_gpuEventElapsedTime((void *)&(tot_start), (void *)&(tot_end));
    crcTime = crcTime +
              ges_gpuEventElapsedTime((void *)&(crc_start), (void *)&(crc_end));

    ges_gpuHostFree(buf);
    ges_gpuDevFree(devTable);
    ges_gpuDevFree(devBuf);
    ges_gpuDevFree(devCrc);
}

totalTime = (totalTime / GES_CRC32_ITERATIONS); /* average */
GES_INFO("total time(ms)=%.3f bs=%d blocks=%d xfer=%dB, throughput=%.3fMB/s",
         totalTime, blockSize, (size/blockSize), size,
         GES_BYTES_TO_MB(size/(totalTime/1000)));
```

```
        crcTime = (crcTime / GES_CRC32_ITERATIONS); /* average */
        GES_INFO("crc time(ms)=%.3f bs=%d blocks=%d xfer=%dB, throughput=%.3fMB/s",
                 crcTime, blockSize, (size/blockSize), size,
                 GES_BYTES_TO_MB(size/(crcTime/1000)));


        size *= 2; /* increase size by power of two. */
    }
}

__global__ void ges_crc32Kernel(int *table, int *data, int blockSize,
                                 int size, int *devCrc)
{
    int tid = threadIdx.x + (blockIdx.x * blockDim.x);
    int blocks = (size / blockSize);

    while (tid < blocks) {
        int *block = &(data[tid]);
        int crc = 0xffffffff;
        int byte;

        for (byte = 0; byte < blockSize; byte++) {
            crc = (crc >> 8) ^ table[(crc ^ block[byte]) & 0xff];
        }

        crc = (crc ^ 0xffffffff);
        devCrc[tid] = crc;
        tid += (blockDim.x * gridDim.x);
    }
}
```

Listing A.6: Serial CPU Encryption Source Code

```
void ges_cpuEncrypt(char *buf, unsigned len)
{
    uint32_t index;
    uint32_t bufSize = (len / GES_TEA_BYTES_PER_BLOCK);
    ges_teaKey_st key;

    key.keys[0] = GES_TEA_KEY0;
    key.keys[1] = GES_TEA_KEY1;
    key.keys[2] = GES_TEA_KEY2;
    key.keys[3] = GES_TEA_KEY3;

    for (index = 0; index <= bufSize; index++) {
        ges_teaBlock_st *block =
            (ges_teaBlock_st *)(buf + (index * GES_TEA_BYTES_PER_BLOCK));
```

```
        ges_cpuEncryptBlock(block, &(key));
    }
}


void ges_cpuEncryptBlock(ges_teaBlock_st *block, ges_teaKey_st *key)
{
    uint32_t index;
    uint32_t sum = 0;

    for (index = 0; index < GES_TEA_BLOCK_ROUNDS; index++) {
        sum += GES_TEA_BLOCK_DELTA;
        block->vecs[0] += ((block->vecs[1]<<4) + key->keys[0]) ^
                          (block->vecs[1] + sum) ^
                          ((block->vecs[1]>>5) + key->keys[1]);
        block->vecs[1] += ((block->vecs[0]<<4) + key->keys[2]) ^
                          (block->vecs[0] + sum) ^
                          ((block->vecs[0]>>5) + key->keys[3]);
    }
}
```

Listing A.7: Threaded CPU Encryption Source Code

```
void ges_cpuThreadEncrypt(char *buf, unsigned len)
{
    uint32_t index;
    uint32_t cpuCnt = ges_cpuCnt();
    ges_teaKey_st key;
    pthread_t thread[cpuCnt];
    ges_teaThreadInfo_st info[cpuCnt];

    key.keys[0] = GES_TEA_KEY0;
    key.keys[1] = GES_TEA_KEY1;
    key.keys[2] = GES_TEA_KEY2;
    key.keys[3] = GES_TEA_KEY3;

    for (index = 0; index < cpuCnt; index++) {
        info[index].tid = index;
        info[index].cpus = cpuCnt;
        info[index].blocks = (len / GES_TEA_BYTES_PER_BLOCK);
        info[index].buf = buf;
        info[index].key = &(key);
        pthread_create(&(thread[index]), NULL,
                       ges_cpuThreadEncryptBlock, &(info[index]));
    }

    for (index = 0; index < cpuCnt; index++) {
```

```
        pthread_join(thread[index], NULL);
    }
}


static void *ges_cpuThreadEncryptBlock(void *infoHandle)
{
    ges_teaThreadInfo_st *info = (ges_teaThreadInfo_st *)infoHandle;
    uint32_t tid = info->tid;

    while (tid < info->blocks) {
        ges_teaBlock_st *block =
            (ges_teaBlock_st *)(info->buf + (tid * GES_TEA_BYTES_PER_BLOCK));
        ges_cpuEncryptBlock(block, info->key);
        tid += info->cpus;
    }

    return (NULL);
}
```

Listing A.8: GPU Encryption Source Code

```
void ges_gpuEncrypt(char *buf, unsigned len)
{
    int blocks;
    uint32_t bufSize = (len / GES_TEA_BYTES_PER_BLOCK);
    ges_teaKey_st key;
    int *devBuf;
    int *devKey;

    blocks = (bufSize / 2) + 1;

    key.keys[0] = GES_TEA_KEY0;
    key.keys[1] = GES_TEA_KEY1;
    key.keys[2] = GES_TEA_KEY2;
    key.keys[3] = GES_TEA_KEY3;

    ges_gpuDevAlloc((void **)&(devBuf),
                    max((int)len, GES_TEA_BYTES_PER_BLOCK));
    ges_gpuDevAlloc((void **)&(devKey), sizeof(ges_teaKey_st));

    ges_gpuMemCpyToDev(devBuf, (int *)buf, len);
    ges_gpuMemCpyToDev(devKey, &(key), sizeof(ges_teaKey_st));

    ges_gpuEncryptBlock<<<min(blocks, GES_GPU_MAX_BLOCK), GES_GPU_MAX_THREAD
        >>>
        (devBuf, devKey, blocks);
```

```
    ges_gpuCheckErr();

    ges_gpuMemCpyToHost(buf, devBuf, len);

    ges_gpuDevFree(devBuf);
    ges_gpuDevFree(devKey);
}


__global__ void ges_gpuEncryptBlock(int *buf, int *key, int blocks)
{
    int tid = threadIdx.x + (blockIdx.x * blockDim.x);

    while (tid < blocks) {
        int index;
        int sum = 0;
        int bvec0 = buf[tid];
        int bvec1 = buf[tid + 1];

        if (0 != (tid % 2)) {
            break;
        }

        for (index = 0; index < GES_TEA_BLOCK_ROUNDS; index++) {
            sum += GES_TEA_BLOCK_DELTA;
            bvec0 += ((bvec1<<4) + key[0]) ^
                     (bvec1 + sum) ^
                     ((bvec1>>5) + key[1]);
            bvec1 += ((bvec0<<4) + key[2]) ^
                     (bvec0 + sum) ^
                     ((bvec0>>5) + key[3]);
        }

        buf[tid] = bvec0;
        buf[tid + 1] = bvec1;

        tid += (blockDim.x * gridDim.x);
    }
}
```