University „Politehnica” of Bucharest

Faculty of Electronics, Telecommunications and Information Technology

Parallel implementation of a Memcached system using CUDA architecture

Dissertation thesis

submitted in partial fulfillment of the requirments for the Degree of Master of Science in the domain Advanced Computing in Embedded Systems

Thesis advisor Student

S.L. Dr. Ing. George Valentin STOICA *Ionuț Alexandru CUȚA*

2021

University "Politehnica" of Bucharest **Anexa 2**
Faculty of Electronics, Telecommunications and Information Technology
Master program

DISSERTATION THESIS
of student **CUȚA I. Ionuț-Alexandru , 421-ACES**

1. Thesis title: Parallel implementation of a memcached system using CUDA architecture

2. Thesis description and student's original contribution (not including the documentation part):
The project aims to create a memcache (in memory cache) application for accessing distributed databases through temporary memory data storage. The application must implement an API consisting of SET, GET, DELETE operations. Data storage is designed based on a hash table structure described by key-value pairs. The application aims to streamline and improve the performance of such a system by accelerating key extraction operations, calculating the hash function, and accessing the table to obtain data using the CUDA architecture, grouping a large number of accesses to be processed in parallel on a CUDA based GPU. In this project we will simulate and analyze the performance of a single memcache node accessed by several clients. In addition to the performance obtained from the massive parallelization using GPU, choosing a database with very high traffic, the latencies added by the grouping of several accesses will be minimal.

3. Academic courses the thesis is based on::
Parallel Computing, Distributed and High Performance Computing, Performance Analysis and Optimization

4. Thesis registration date: 2021-01-05 21:40:27

Thesis advisor(s),
Ș.L.Dr.Ing. George Valentin STOICA

Student,
[illegible]

Master program director,

Dean,
Prof. dr. ing. Mihnea UDREA

Validation code: **bfb2aae0bf**

[declarative de onestite academica]

Table of Contents

List of figures

List of tables

Abbreviations

CPU = Central Processing Unit

GPU = Graphics Processing Unit

GPGPU = General purpose programming on GPUs

LRU = Least Recently Used

UDP = User Datagram Protocol

TCP = Transmission Control Protocol

Introduction

Memcached, or more broadly in-memory key-value store, is typically used in data intensive applications that require high throughput and low latency. It is a general purpose, scalable storing paradigm and its main goal is to offload some of the traffic from traditional databases. Transactions to and out of this memory cache are done using pairs of key-value and a core interface that uses SET to write into the memory and GET to retrieve the data.

Memcached is not used as a permanent storage but as a bridge between the permanent databases and the client, storing the latest or most accessed items. This type of memory store should therefore be interpreted by the clients as a transitory one, since it does not guarantee that the stored data will be there once they need to read it back.

Storage is distributed across different servers or nodes, to which all the clients have access, and calls to the API can be done through both UDP and TCP. Many popular websites such as Facebook, Youtube or Twitter report using a Memcached implementation in their systems to provide quality service and fast access to their users.

Multiple attempts to improve performance have been done, each targeting a different area. Efforts to improve the software behind the application, such as MICA, included taking advantage of CPU parallelism, different data structure and network stacks. On the hardware side, aiming at an increase in energy efficiency, hardware accelerators have been used to replace the CPU. ASICs provide a good efficiency but do not keep up with the constantly changing applications while FPGAs provide some adaptability but are not very fast when it comes to reconfiguration or context switch.

An alternative that could solve both problems is the GPU, which is capable of achieving both high throughput and energy efficiency. Having the advantage of high parallelism and high memory throughput they can process more requests at the same time and even split the work per request, therefore being able to provide fast data access, latency reduction and general performance improvement even when working on large sets.

This thesis aims to design, test and analyse the performance of a CPU-GPU system that will do the necessary computations of an in-memory key-value store. The CPU will handle the packets, convert them to an easier to process format and send them to the GPU, which will act as an index table for the items stored in CPU memory. Packets will be batched together and launched whenever the batch limit has been reached or a certain interval of time has passed.

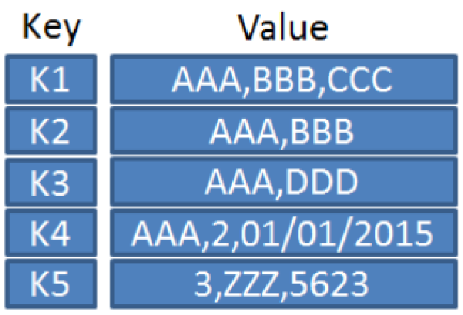
In Chapter 1 of this thesis we will briefly describe what an in-memory key-value store is. In Chapter 2 we will go over the Memcached architecture, process flow, look at the performance bottlenecks for each of its components and try to explore different approaches in which to optimize them. Chapter 3 is reserved for characterizing hash functions along with different hash table design patterns, ways to handle collisions and ways in which they could be implemented in a parallel environment. Chapter 4 goes over the CUDA architecture and programming model, chapter 5 presents the implementation and, finally, Chapter 6 is dedicated for results and performance analysis.

1. In-memory key-value store

1.1 Definition

In a *key-value* store, data is represented as a collection of *key-value* pairs and can be viewed as an associative array. *Keys* are unique identifiers (filename, URL, hash) while the values can represent any type of data (test, video, JSON document, etc.). [1] The application has complete control over what data is being written for a particular *key*, making this system general purpose.

A simple example of *key-value* pairs can be seen in figure 1.



Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Figure 1 - Key-value pairs

We can analyse the *key-value* stores considering the following parameters:

1. Concurrency

   Concurrency can only be defined per *key* and it is offered as either optimistic writes.[1] This means that although conflicts are possible i.e. different applications trying to write different data using the same *key*, they will be rare. When this happens, one application's data will be discarded.

2. Queries

   For this type of database there is only one way to query for data and that is through the *key*[1]. The user or application will not be able to query using an index.

3. Transactions

   There is no guarantee that a transaction has been completed. [1] Although such a mechanism is possible to implement, it would greatly affect performance.

4. Format

   Key-value stores do not have a predefined format. As the name implies, there are only 2 fields for each item, *key* and *value*.

5. Scaling

   Key-value stores can be scaled out by simply adding more partitions or nodes. [1] This means that the user will have to choose the endpoint by either a *hash value* or a field inside the *key*.

6. Replication

   Replication, or having a copy of the same item on different machines is theoretically possible but may prove to be a complex task depending on how conflicts are treated.

7. Portability

   Key-value stores are portable since they do not require a specific language and the interface would be very similar for different implementations

Applications that may benefit from using *key-value* stores are: user profile preferences, product recommendation, patterns and behaviour to be used for customized adds, etc. [1]

2 Memcached

2.1 Description

Memcached was designed to offload some of the traffic from traditional databases by caching the latest or most accessed items. It is now an open source, high-performance, distributed memory object caching system and is widely used by large companies to speed up their applications.

The Memcached system has 2 components: the servers and the clients.

A Memcached server is a process that is responsible for the following tasks:

- Managing memory and allocation
- Keeping track of objects stored
- Processing requests from clients [2]

It is very easy to scale up a Memcached system, increasing the capacity of stored data can be done by adding additional servers or nodes to the system.

A client is an interface through which Memcached servers can be accessed. Its responsibility is to assemble the data in a serial packet, be able to connect to different servers and choose which server will the packet be sent to, usually by hashing it. Clients have been implemented across different languages.[2]

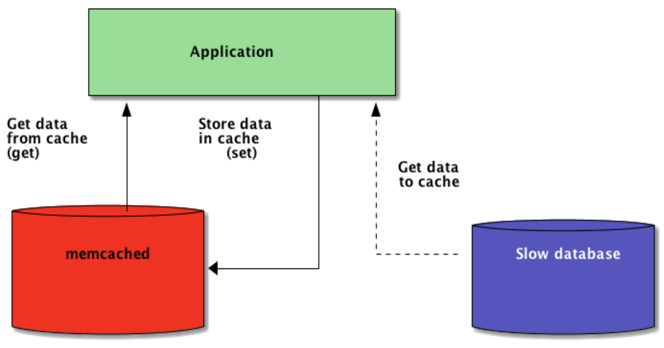


Figure 2 - Memcached system

Memcached is a store for small chunks of arbitrary data that can be represented as strings or objects that can be generated by database access or come directly from the application. [2] This means that the application has full control over what data it writes.

Since it is a distributed system, it means that the end user has a unified view over the environment, even if there are more physical machines running Memcached. [2] Servers can be identified through procedures such as hashing the key. This is done by the client and it guarantees that a particular key will always end up pointing to the same server.

Servers index data using a structure called hash table. This can be viewed as an array of buckets that can store multiple items, either in consecutive order through a linked list[2] or random access through a *hash value*. To actually store the data, instead of traditional memory allocation, a slab system is used. This is mainly done to avoid fragmentation.

Communication between server and client is done through UDP for set commands and TCP for get commands. [4]

These elements along with the list of primary commands will be detailed next.

2.2 Architecture

2.2.1 Hash table

As mentioned before, the hash table is an array of buckets, usually aligned to a power of 2. When writing an element into the hash table, a bucket is chosen by computing the hash value for the item and using it as an index. A mask hash must be used on the hash value to take into account the size of the hash table. If k is the size of the hash table then the mask would be 2^k – 1, and with a logical AND performed between the value and the mask, the result will point to a valid bucket. [5]

Items inside the bucket can generally be stored in multiple ways but the Memcached implementation chose to store them as linked lists with a NULL termination. [5]

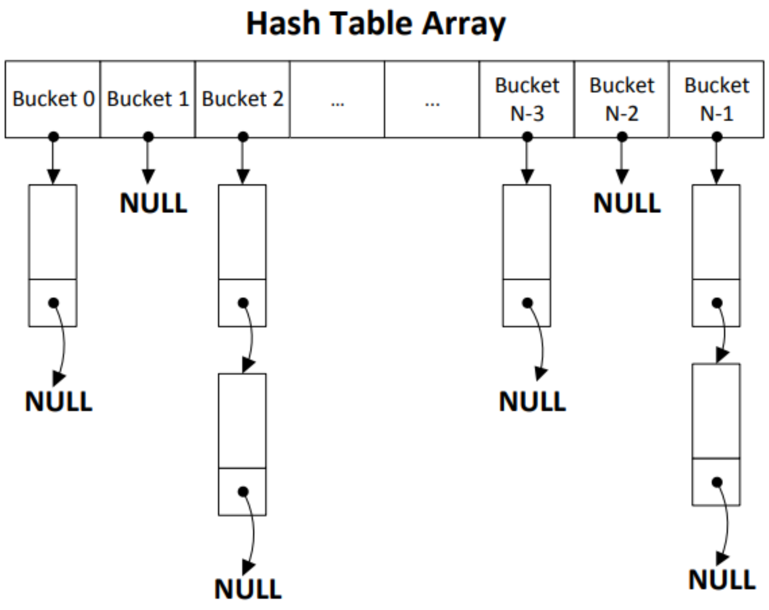


Figure 3 - Hash table

Since the hash table acts as a shared memory between all threads, thread safety is maintained using a global lock that is asserted whenever the hash table is accessed and released after the operation has been completed.

Without the usage of this lock, there would be some scenarios where items close to each other would interact in unintended ways. Since buckets are constructed as linked lists, in case of a DELETE on consecutive elements, for each element a thread would update the pointer of the previous element and in this case the list would get broken since the next element in the list would be pointed to by an element that no longer exists. This would also be true for GET since a linked list may be traverse while being updated. [5]

2.2.2 LRU Cache (Least recently used)

The LRU is a double linked list used to monitor all items inside a slab partition which was mentioned before. This structure is maintained similar to the hash table.[5] In the event that a bucket is full and an element must be evicted to free space for a new element, the LRU is used to determine which is the least used and can be evicted.

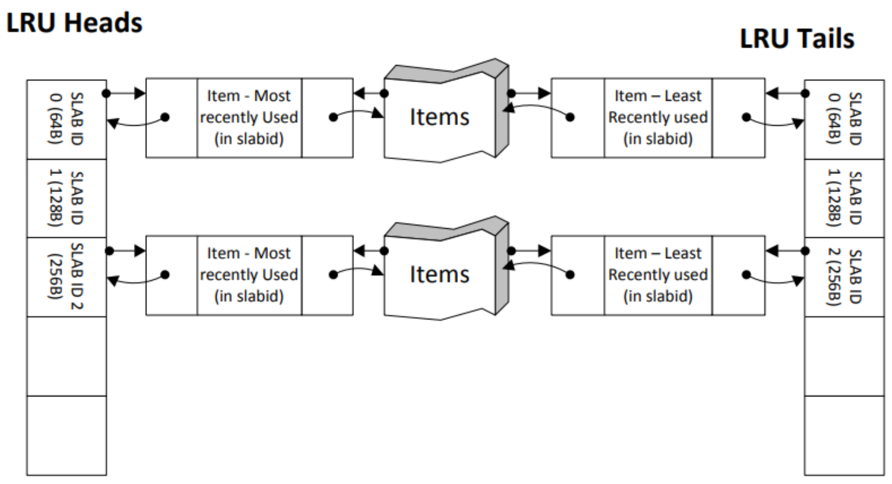

LRU Heads
LRU Tails
SLAB ID 0 (64B)
SLAB ID 1 (128B)
SLAB ID 2 (256B)
Item - Most recently Used (in slabid)
Items
Item – Least Recently used (in slabid)

Figure 4 - LRU

To find the least used item, the tail is checked. [5]

A global lock is also used for the LRU in the same way it is used for the hash table. When a GET request is performed, the position of the item inside the LRU is updated to move to the front. [5] Without a global lock 2 elements can both be moved at the head of the list at the same time, creating a conflict an creating a broken pointer similarly to the hash table linked list scenario.

2.2.3 Cache item data

This structure is responsible for holding key-value data such as: [5]

- Pointers used in the hash table and the LRU
- Key and value
- Length of the key and value

2.2.4 Slab allocator

Since traditional memory allocation calls would be inefficient, a slab system is used to manage the cache items. [5] These are pre-allocated and can hold multiple cache items depending on their size. A list with all the free locations is maintained and when a new cache item needs to be inserted the slab allocator will find a location that can fit it. [5]

From a system perspective, we can say that the hash table acts as the link between the key and the value stored in the slab system. Instead of looking through all the slabs, the index is obtained from the hash table and the value returned in constant time.

2.3 Commands

There are 3 main commands supported by the Memcached architecture: [4]

1. GET – used to retrieve data based on a key or multiple keys
2. SET – used to add data to memory, possibly overwriting existing data
3. DELETE – used to remove data based on a key, if it exists

Additionally, there are over 10 other commands ranging from different flavours of SET (APPEND, PREPEND, REPLACE) to statistics about the state of the memory. [4] Most of the effort is spent on GET since it represents the bulk of requests in the operation of Memcached. [5]

The standard protocol involves running a command against an item. Items have the following format: [4]

- A key – an arbitrary string with a length up to 250 bytes
- A 32bit “flag” value
- An expiration date – either 0 for unlimited or the number of seconds
- A 64bit “CAS” value
- Arbitrary data

Considering only the main commands, the basic flow of Memcached is as follows: [5]

```
request arrives
data packet is
        retrieved
        interpreted – data and command are extracted
a hash value is computed based on the key
cache lock is acquired to begin processing
        destination is determined
        for STORE only – value and flags are being written to a
        cache item
        hash table is accessed and command completed
        LRU is updated
cache lock is released
response is created
statistics are updated
response is sent back to client
```

Besides the parts of the flow that include a global lock, other steps can be parallelized to some degree. These include building the data for the hash table access (computing all the hashes), building the response or sending the response to the clients.

2.4 Performance and optimizations

2.4.1 Limitations

As mentioned before, Memcached cannot be viewed as a persistent cache. This means that the data can either be lost after being evicted to clear space, replaced by an access that uses the same key, or a power failure can clear the whole cache of a node.

To improve performance a method of scaling out is used instead of scaling up. [2] This means that new nodes are added to increase the cache size, which affects the whole infrastructure and clients need to be aware of the change. Another limitation is that nodes are independent and unaware of each other.[2] This means that operations for some possible commands can be restricted or statistics and monitoring may not be so efficient.

2.4.2 Performance Analysis

In a test environment prepopulated with keys the following observations have been made regarding performance:

- 15% of the time is spent on the network driver [3]. A possible optimization opportunity is to create a network driver specifically for Memcached architecture and commands
- Copying data from the user space to the kernel space takes up 3% of the time. [3] A possible solution to this would be to bypass the network frame.
- Smaller delays are inserted handling TCP packets and waiting for the lock to be released [3]

While each of the above bottlenecks come with a potential for optimization, an alternative solution would be to rethink the architecture, data structures and process flow.

2.4.3 Optimizations

A first optimization would be to replace the global lock with a stripped lock that would guard every Nth bucket inside the hash table. [5] This effectively splits the hash table into more partitions and the buckets can be opened in parallel. Inside a single partition there will still be conflicts but choosing a small value for N should ensure that their frequency is greatly reduced.

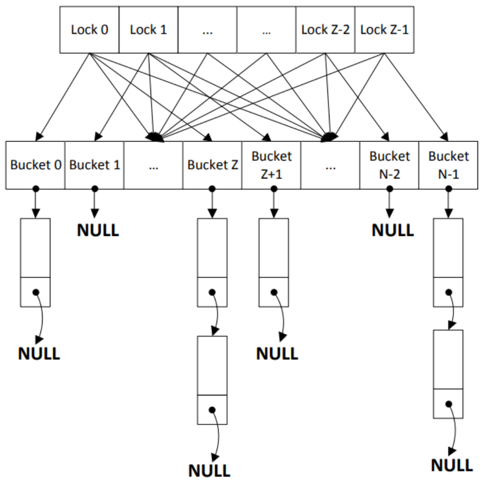


Figure 5 - Stripped lock

To remove the lock on the LRU, a concept known as “bag” LRU is used, where a single linked list replaces the double link list and items are inserted into LRU “bags” based on their timestamp. [5][6] This method guarantees that when inserting new items in the LRU it will not conflicts with older items and therefore removing the need for a lock.

The double linked list structure can be reused such that its “previous” pointer know hold information about the position of the list from oldest to newest. [5][6] This information can be used by the eviction mechanism which will always try to evict items from the oldest bags. The newest bags will be used to insert new items but this will not prevent conflicts with items from the older bags that are accessed and must be reinserted to the front of the queue into the newer bags. A solution to this would be to have alternate bags for items newly added into the LRU and items that are “refreshed”. A lock per bag is needed to avoid conflicts between cleaner threads and eviction threads. [5]

The procedure for initializing the bag LRU would look as follows: [5]

```
Create the array of bag heads
For each ID
    Initialize the first bag
    Initialize the lock
    Point the bag array head to this bag
    Increment the counter
Launch the cleaner threads
```

Cleaner threads will watch the newest bag and whenever it gets full it will create a new one and initialize it in a similar way as described above, by pointing the next pointer of the full bag to the newly added one, and then setting it as the newest bag. Cleaner threads need to continuously traverse the structure and remove expire cache items but also merge bags that fall under a certain threshold. [5]

The other issue that greatly affects performance is the serial model of execution used for the hash table. An approach where the hash table is partitioned can remove the need for a lock therefore increasing performance. [7] The main downside with this approach is that it relies on random accesses spread out across the hash table. If for some reason one or more clients perform heavy access into a specific partition all the effort will be put on the threads that belong to that partition leaving all the others one idling. If this approach is chosen a mechanism that prevents such imbalances should also be implemented.

Another approach would be to make an advantage out of the fact that over 99% of accesses will consist of GET. [7] Therefore, if a parallel hash table is designed to completely remove global locks for GET accesses, the performance should be greatly improved.

To remove the GET global lock, the main case that need to be addressed is dealing with expansions. An expansion is required when the number of cache items is much larger than the available buckets of the hash table. [5] This is because adding too many items to each bucket, although possible, it will greatly increase the time needed to traverse each bucket and lower performance. Adding more buckets to the hash table will not solve the issue since all the items will still be distributed between the initial buckets so a new hash table has to be created and all items from the old table rehashed into the new one.

The flow for a GET with no global would look as follows: [5]

```
Compute the hash to get bucket location
If not expanding
       Get the item
       If item not found
                If expansion has already started
                     Find bucket in the new table
                     Wait for the initial bucket to be empty
                     Retrieve item
If expanding
       If bucket not moved
                Check bucket chain for item
                If item not found
                     Wait for the bucket to be empty
                     Find the new bucket and retrieve the item
       If bucket moved
                Find the new bucket
Retrieve item
```

Since it was established earlier that lock removal would be focused on the GET command since it takes up most of the traffic, SET and DELETE commands can maintain the lock mechanism but the stripped lock optimization mentioned above can be used for some performance improvement.

In the case of an expansion, the process for each command needs to check if the bucket has been moved. If not, the process is locked out until the bucket is empty, only after that being only to move to the new bucket and insert or delete the cache item. [5]

3 Hash functions

3.1 Definition

A hash function is a function who takes as input a message of any length which is then transformed into a message of fixed length called hash value, message digest or checksum.

Hash functions are defined as f : D - > R, where domain D = {0, 1} *, which means that the elements of the domain are binary sequences of variable length and the elements of the results R = {0,1}^n are binary sequences of fixed length, where n can take any value greater than 1. Therefore, a hash function will process a message M of any length and produce a result h of length n. [8]

Hash functions are mainly used in cryptography. If a hash function called H meets all the criteria bellow it is called a cryptographic hash functions:

1. H accepts as input a sequence of any length
2. H produces an output of fixed length for any input message length
3. H must behave as a random function but at the same time be completely deterministic and reproductible. This means that every time it is applied to a message it will always return the same result
4. If we have a message M, it is very easy to compute its hash value. This means that its complexity must be O(n) where n is the length of the input message. Therefore, M should be very easy to implement in both software and hardware.
5. If we have the result returned by computing a hash function, it is computationally difficult to find the initial message M. This is the reason hash functions are also called 1-way.
6. If we have a message M1, it is extremely hard to find another message M2 such that H(M1) = H(M2). In cryptography this property bears the name of collision.
7. A hash function must be heavily sensitive to any change of the input message. This means that if a single digit is change in the input message then the output message will be completely different. [8]

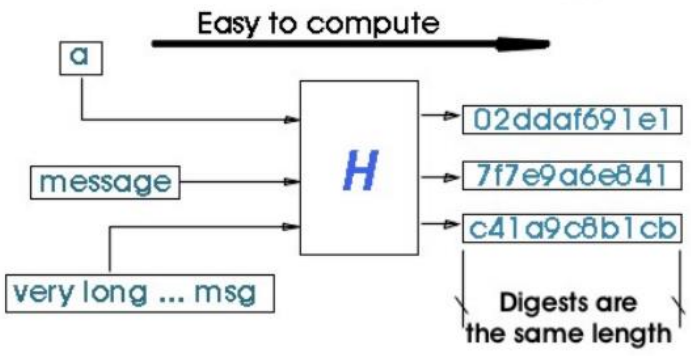


Figure 6 - Hash function

These properties are important only in a cryptographic usecase as they greatly slow down any type of attack that might make the hash function unsecure. When their scope is outside of this domain, points 5 and 6 do not have to taken into account. This leave us with the following requirements: a hash function has to return a fixed length message which is easy to compute and act as a random function even though collisions may happen.

To better understand the working principle of more complex hash functions, the Merkle-Damgard construction. This structure defines a step by step procedure of obtaining a result of fixed length out of a message of variable length.

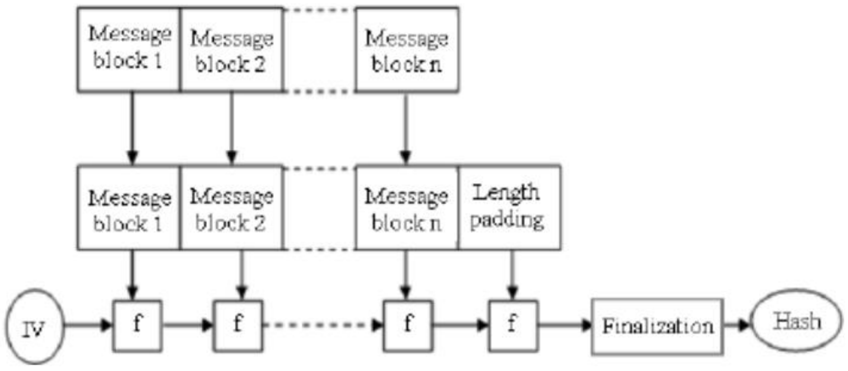


Figure 7 - Merkle-Damgard construction [9]

The main blocks of this structure are:

- IV : Initialization vector – a numeric sequence used as a startpoint for the first iteration
- F: compression function – a one way function that receives a fixed length sequence as input and generates a sequence of fixed length
- Finalization: an additional transformation which reduces even more the length of the result obtained from the iterations
- Hash : the result

As can be seen in the figure (?), the input message has to be devided into n blocks of equal length, this length depending on which function f is used. The length of the message is then modified so that it is a multiple of a certain number by adding a 1 at the end of the message, a number k of 0s and then the value k written in binary. [8]

In every iteration, the function receives 2 arguments: the corresponding block of the message and a chaining variable. In the first iteration the chaining variable is the initialization vector and together with the first block it represents the inputs to the compression function. The computed result becomes the chaining value for the next iteration. During the last iteration, the chaining value is used as a input for the Finalization block which applies a transformation, usually a decrease in length. In some cases, this block is not present. [8]

3.2 Hash table

A hash table is a data structure similar to an associative array, which maps keys to values. What is particular for this type of structures is that they use a hash function on the key to compute the index. In other words, if we have the element x and the hash function h, we will store x at position h(x) in the table.

Most hash table assume that the hash functions used in their implementation will cause collisions, meaning that for two different items the result of the hash function will be the same and they will need to be stored at the same location. A simple way to solve this is to use buckets and have each possible location in the hash table be able to store multiple items. Multiple solutions to this problem will be addressed.

3.2.1 Open addressing

In this method collisions in hash tables are resolved by looking elsewhere in the table. For this we need to have multiple hash functions that are guaranteed to map an item to different buckets of the hash table.

When searching for an item, we apply each hash function to our element and check if the hash table contains anything at that location. If yes, we return the value, otherwise we return a null or similar indication to indicate that the nothing was found. The operation for inserting is similar, only in reverse, and we need to verify if each location is empty. If yes, then we can insert the element at that location. [10]

Advantages of open addressing are: [11]

- Only one hash table is required
- It is efficient in terms of storage

Some of the drawbacks include: [11]

- Requires that the cells or location have a flag to indicate if they are empty or occupied. Alternatively this can be understood as 0 for empty and any other value as occupied
- The items that are hashed must have distinct keys.
- Proper table size needs to be chosen. This is chosen according to the number of elements that are intended to be stored and the load factor

Alternative implementations of open addressing are:

1. Linear probing – only 1 hash function is used but when a collision occurs the second location is an increment of the first. Modulo arithmetic is used to wrap around the table. [11]
2. Quadratic probing – similar to linear probing, the only difference being that successive location are computed based on an arbitrary quadratic polynomial instead of simple increments. [11]

3.2.2 Chaining

This method of resolving conflicts implies that each location indicated by a hash is a pointer to a structure such as a linked list that can hold multiple elements. When looking for an element the entire list needs to be scanned. In a similar manner for inserting, the list need to be traversed so that the end is found and the element is appended.[10]

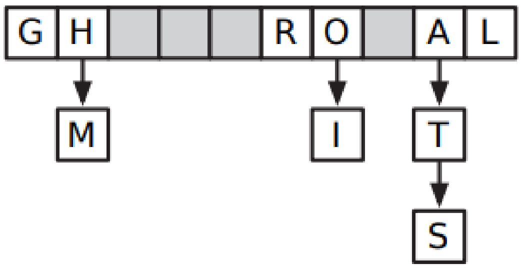


Figure 8 – Chaining [10]

Complexity is the same for both operations. Computing the hash is done in O(1) and then additional O(1) for each element of the linked list, making it O(1+l(x)) where l(x) is the length of the list. [10]

The advantages of this method are:

- Efficient collision resolution. [11] The length of the linked list can be extended indefinitely, the only downside of this being that it increases complexity and reduces performance
- Deletion is easy. [11] The standard algorithm for deleting an element of a list can be used

Some of the disadvantages are as follows:

- It involves writing the code for the separate data structure. Other and more complicated structures can be used instead of a simple linked list [11]
- If nodes are dynamically allocated, this can slow down some machines and reduce performance [11]

3.2.3 Hash table size

Hash tables have limited size and in some cases they are not large unless they need to be. As mentioned before in the case of Memcached expansion can be performed, effectively doubling the maximum size.

Prime numbers are used for table sizes because this allows easy probing in the case of the open addressing method. This guarantees that any step length is a relative to the table size and ensures that an empty spot will be found for any step length as long as there are still empty spaces in the hash table. [12] If the size and the step would not be relative primes, a state in which the index wraps back to the beginning of the table can be reached. Therefore it would get incremented and will follow the same column reaching the same value from where it started.

3.2.4 Integer hashing

The methods described above for avoiding collisions have some particularities when working with keys that have a fixed size of 32 or 64 bits.

An integer hash table can be created using the following model, using 32 bit values:

- Insert:
    - When inserting an element to an uninitialised location, we create a 12 byte array and use the 4 bytes to store the number of entries, the next 4 bytes to store the key and the last 4 bytes to store the data [13]
    - When inserting to an initialized location, we allocate 8 more bytes to the array to which we copy the key and data, making sure to increment the number of entries afterwards [13]
- Search:
    - Location is found using the hash value
    - If it exists the array is traversed and the element is returned

One of the first problems encountered with this method is that a delete operation would require that 8 bytes are deallocated and all the elements are copied down towards the first element.

The approaches mentioned above can be applied to this case.

For standard chaining integer hashing, a 12 byte node can be created which will contain the key, the data and a 4 byte pointer to the next node. When inserting an element, each additional node can be appended at the end of the list or removed and the list rerouted in the case of a delete. [13]

3.2.5 Compact Bucketing

As mentioned in the previous chapters, hash tables can be either implementing using open addressing, where all the elements are stored directly in the hash table, or closed addressing, where keys are mapped to buckets through a specific data structure.

Compact hashing offers a different implementation in which the buckets contains quotients, which, in contrast to open addressing implementation, do not need to store any information to recover a key. The problem arises with the overhead for each bucket, holding pointers and information about their size, which grows with the number of buckets. [14]

If we consider a key-value pair, we can compute the hash of this pair and assign it to the bucket pointed by the result, in the following way:

b = hash(x) mod n, where x is the key and n is the number of buckets.

Inside that bucket, we represent the key by its quotient, which can be:

q = hash(x, y) div n

Then, the key x can be recovered in this way:

X = f(-1) (qn + b) [14]

There are 2 variants to this implementation, the first being the simple compact hashing. To perform an insert operation on a simple compact hash table, the entire bucket is scanned, while for insert the existing bucket is copied to a new array and the new key is added at the end. [14]

The second variant is represented by a space efficient compact hashing method, in which each bucket is divided into N sub buckets, such as each sub bucket has a complexity of O(1) with a high probability. This way, the search operation would have an expected O(1) complexity. [14]

3.3 Cuckoo hashing

Cuckoo hashing uses two different hash tables, called T1 and T2, along with two hash functions h1 and h2. Either key can be stored either in a cell of T1 or a cell of T2, but not in both at the same time. The idea of storing in one out of two places is also called two-way chaining. [16]

When looking for a key, the key is first hashed using h1 and T1 is checked at that location. If there is no result, the same procedure is repeated for T2 using h2. [16]

Deleting a key is done in constant time, similar to the search operation. Both tables are checked and if the key is found then it deleted. Insert is more complex than the previous two operations. The approach for insertion is called “cuckoo insertion” at it implies kicking other keys away until each key has it own position. [16] More specifically, if we wanted to insert a key x into the hash table, we would compute h1(x) and look into T1, if that location is already occupied then we would have to evict that element and replace it with x, then repeat the same procedure with the evicted element but for T2. This would create an eviction chain that will eventually find a spot for each element.

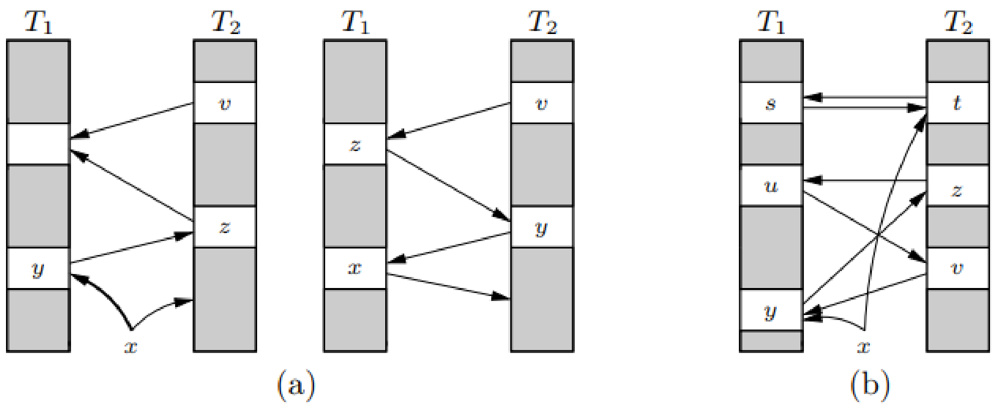


Figure 9 - Cuckoo hashing[16]

This is visible in figure (?). For case (a) x is inserted by evicting y from T1 and moving it to T2, evicting the element at that location, z, and moving it to T1. Therefore, it is not necessary to also check the hash value of x for T2. In the opposite case of (b), y cannot be evicted to free up space for x since it would create a chain that would end up with y being inserted in the same position in T1. When trying to rehash x into T2, the same behaviours occurs.

The procedure for the insert operation would look as follows:

```
Loop MaxTimes
        If T1[h1(x)] is empty then T1[h1(x)] <- x; return
        Swap x and T1[h1(x)]
        If T2[h2(x)] is empty then T2[h2(x)] <- x; return
        Swap x and T2[h2(x)]
End loop [16]
```

Alternatively, a search operation can be performed before starting the insert operation to ensure that the key is not already written. [16]

Another variant of this algorithm is to use a single hash table on which to hash the key using both hash functions. [17]

3.3.1 Insert Analysis

When inserting a new element, there are two possible scenarios.

The first possible scenario is that there is an arbitrarily long number of tries to insert an element. In can be shown that if the insertion chain is not closed then it is not possible to to find a location for all keys. [16] This means that in order to solve this issue new hash functions need to be used.

The second scenario is that an element is successfully inserted and it can be proven that the sequence is expected to end in O(1). [16]

3.3.2 Performance

To analyse the performance of cuckoo hashing, it can be compared to some of the methods that were described in the previous chapters. These methods are as follows: chained hashing, linear probing, double hashing, two way chaining.

In the case where the hash table stayed constant, that implying there is a balance stream of operations that insert and remove keys in and from the hash table, search time is the same for all methods. For unsuccessful searches, linear probing was the fastest to return that answer, having cuckoo and two-way chaining closely behind. When it comes to inserts, it can be observed that cuckoo and two-way chaining are slow for small sets of data. For deletion, when using large sets, cuckoo hash is the winner. [18]

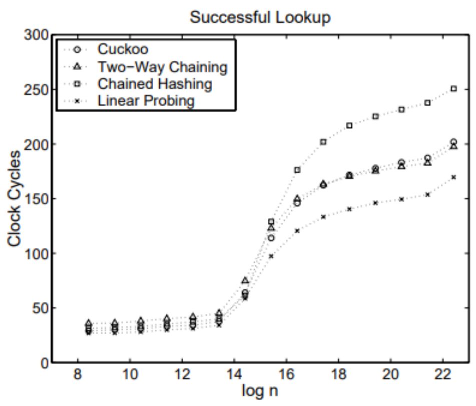


Figure 10 - Hash table performance [18]

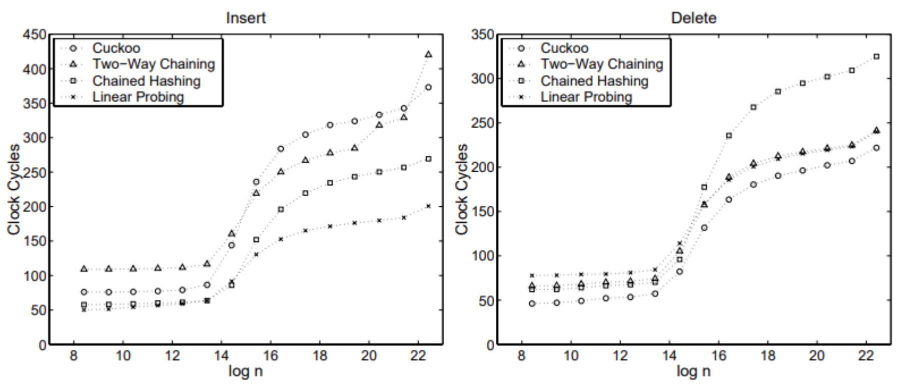


Figure 11 - Hash table performance [18]

In the case of dynamic hash tables, results for insert operations in growing hash tables were in 2% of each other while for delete operations in shrinking hash tables, linear probing came first followed by chained and cuckoo hashing. [18]

The result would be that cuckoo hashing can average the other implementations in terms of performance while being very simple to implement at the same time.

3.4 Parallel hashing

When designing a parallel hash table, 2 aspects have to be taken into consideration.

1. Synchronization between threads. All hash table design approaches and algorithms presented in the previous chapter involved some kind of sequential access, for example linked list which by definition have to be traversed sequentially. In other cases, locks were needed. When designing parallel hash tables, a data structure where all elements of a bucket have to accessed in parallel needs to be found.
2. Variable work. If we take the example listed for the previous point, traversing a list will need as much work as its length. Only the thread processing it would be affected by this, but in the case of parallel hashing it would desired to have all the threads doing approximately the same work or have some mechanism of synchronization to make sure that idle threads are waiting.

3.4.1 Perfect hashing

Perfect hashing relies firstly on having a hash table size that is much larger than the elements that need to be stored and secondly on having no collisions. To obtain this property two level hashing is used as described in the previous chapters, and an additional condition that for each level the complexity is O(1), we will expect to be able to find each item in O(1). [19]

A similar approach is taken when considering minimal perfect hash tables, but while gaining in space efficiency, they have inherently sequential mechanism making them unfit for parallelization. [19]

3.4.2 Cuckoo hashing

It has been shown that when using at least 2 hash functions and storing an item into the bucket containing the least amount of items reduces the expected size of the longest list. [19] This is an advantage when working in parallel because we can compute the two hashes, access the tables in parallel and have the threads count the elements in parallel between each other.

Standard cuckoo hashing places at most one element into each bucket while relying on multiple available locations for each item in order to be evicted and moved around until a chain that fits all items is found. Like the case before, this is sequential to some degree and would not be efficient to parallelize so an implementation in which we allow more items per bucket and have each bucket be processed in parallel is desired.

3.5 Parallel cuckoo hashing

The first obstacle when trying to implement parallel cuckoo hashing is ensure that conflicts between items are properly handled. This can be implemented by attempting to write each item at the same time and have a mechanism between threads to communicate and signal which write has succeded. If the hash table is established as having two different hash functions then the items that failed for h1 will attempt to access h2. The items that still fail after this step will need to move back to h1 and evict the items from the occupied locations. The process is then repeated for these items, until all of them are stored. [19]

This implementation however would not be efficient for large hash tables since items would have to move around using global memory. A solution would be to use the first level of hashing to find the bucket and then have the bucket divided into smaller buckets. Inside this bucket parallel cuckoo hashing could then be used. [19]

3.5.1 Hash table design

Constructing a hash table of size n for this method will consist of two phases. [19]

Phase 1. Items are distributed to buckets using a hash function h.

The goal of this phase is to distribute items across buckets. It begins by computing an estimation for the number of buckets needed based on a low risk of overfilling the buckets. It has been observed that a number of buckets of n/409 is enough to have a maximum estimated occupation of 80% for each bucket. [19]

After computing the destination for each key by hashing it, the count of each bucket has to be determined along with the offsets for each item. If any bucket has more than 512 items inside a new hash function needs to be chosen. Next step is to rearrange the data, splitting it by key and value to benefit from write coalescing. [19]

Phase2. A hash table is built in shared memory for each bucket

A block of 512 threads works to parallel cuckoo hash all items inside each bucket using previously prepared hash values. This step consists of simultaneous tries from each thread to insert its item into the value indicated by the hash value and the process is repeated for the number of hash values used. [19]

The last step of this phase is to copy the newly created hash tables from shared memory out to global memory.

For retrieval, the bucket is computed based on the hash and after that the bucket hash functions are used to look for the item. [19]

3.6 Hash function choice

When choosing a hash function to be used when accessing a parallel hash table there are three main properties it needs to have.

1. Format. Working in parallel with the GPU will most likely involve fixed key or value sizes, therefore an integer function is preferred.
2. Speed. The hash functions will most likely be computed by the CPU as part of the pre-processing, that means that a hash function suitable for the CPU is preferred
3. Distribution. When designing a collision handling mechanism the probability of collision for the specific hash functions needs to be taken into account, but nevertheless that value should be as good as we can get since it will improve overall performance

In the end choosing a suitable hash function will be a trade-off between speed and distribution efficiency.

Since it has been stated in the previous chapters that at least two hash functions are needed for the cuckoo hash implementation, a 64 bit integer hash can be chosen, and if we assume that the key length will be 32 bit, we can consider both halves of that hash value as two different functions.

If we are looking for a good quality hash (meaning good distribution and low collision probability) that are still somewhat fast then good candidates would be XXH3 and possibly Murmur3, according to automated tests run using SMhasher. [20]

If we consider speed to be the most important priority and make the assumption that the GPU will be more efficient in dealing with collisions than the CPU to compute more complex hashes then bitwise XOR hash might be a better choice. This is easy to implement as well.

4 CUDA Architecture

4.1 GPGPU Computing

GPGPU (General-purpose computing on Graphics Processing Units) represents the use of processing units, typically designed for computer graphics, to perform computation in applications that are normally handled by the CPU. Since the main advantage of this type of high-performance computing is to crunch large amounts of data, tasks that well suited to a GPGPU implementation need to be data parallel and throughput intensive. Data parallel means that a processing unit can execute the same operation on different elements at the same time and throughput implies that the task will have a continuous stream of data ready to be processed in parallel. Since the complexity of each processing unit is a lot lower than that of a CPU, GPGPU computing excels when the algorithm requires relatively simple operations to be performed on each item of the input data.

Figure (?) shows the main differences between a CPU and a GPU. Since the GPU is specialized for highly parallel computations, more resources are allocated to data processing rather than data caching and flow control. Instead of relying on large data caches to avoid memory access latencies, the GPU can hide these latencies using computation. [21]

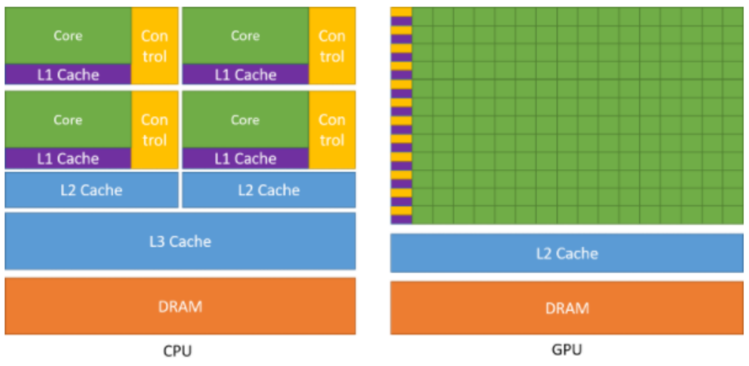


Figure 12 - CPU and GPU architecture [21]

Some of the applications that have been accelerated using GPGPU computing include a wide domain spanning from image or video processing, to deep learning, numerical analysis and computational sciences such as weather prediction.

There are multiple APIs available for GPU programming such as the heterogenous and open-source OpenCL or the proprietary NVIDIA CUDA designed to work with NVIDIA GPUs.

4.2 CUDA

4.2.1 Introduction

CUDA was introduced in 2006 by NVIDIA as a general purpose computing platform and programming model that could leverage the parallel capabilities of their GPU to solve problems in a more efficient way compared to the CPUS.

The CUDA software environment allows developers to use C++ and so it is designed to have a low learning curve for programmers already familiar with standard programming languages.

There are 3 main abstractions introduce by the CUDA environemnt: [21]

- A hierarchy of thread groups – threads are organized into warps and more broadly into blocks
- Shared memories – threads can access a chunk of memory that is shared between them
- Syncrhonization – threads can communicate or wait for each other

This enables users to target fine-grained parallelism at thread level while maintaing coarse-grained parallelism by partinioning the task and splitting the work between thread blocks. Blocks of threads are designed to run on any available MP (multiprocessor) of a GPU, either concurrently or sequentially, such as each CUDA program can be executed across different devices. [21] This concept is illustrated in figure (?) and is reffered to as automatic scalability.

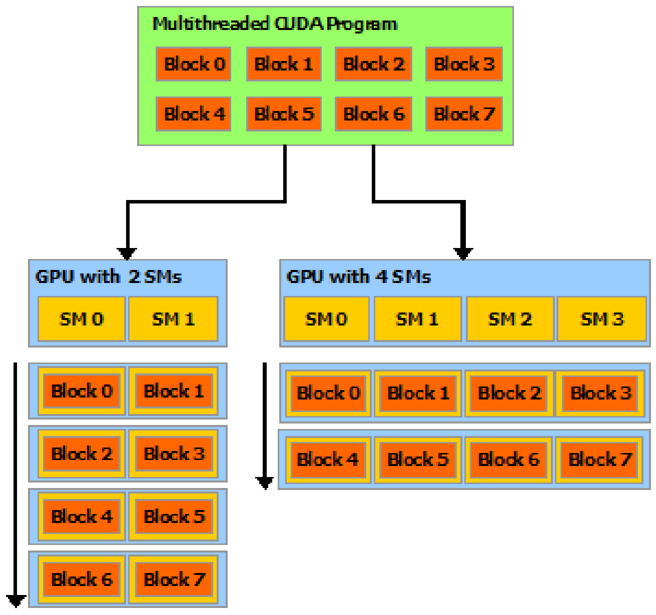


Figure 13 - Block scalability [21]

4.2.2 Programming model

Kernels

The CUDA environment allows the user to define C++ functions called kernels that, when called, are executed in parallel by different CUDA threads. Besides the usual arguments that a C++ function would have, a kernel is also launched with a set of parameters that indicate the launch configuration – how many threads to launch and how they are organized at higher levels (in threads blocks, for example). [21]

Each thread can identify its own position based on a variable called threadIdx. [21] This then enables the thread to compute what part of the task it is assigned to so it can retrieve the necessary data. threadIdx can either be 1,2, or 3-dimensional depending on the type of thread block it belongs to. This way provides a more natural approach when working with 1D, 2D, or 3D arrays.

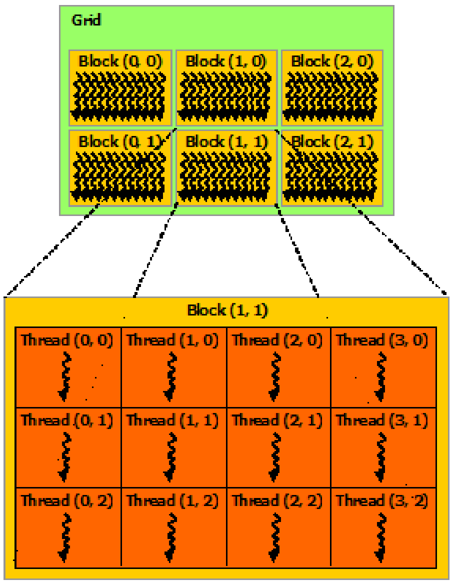


Figure 14 - Thread grid [21]

Figure (?) shows an example of how threads can be grouped into 2D thread groups which are then organized in a 2D grid of thread blocks. The number of threads inside a thread block is limited because by definition they expect to use shared memory and therefore need to run on the same core. A grid size of more than one is usually used when the number of thread blocks exceeds the capability of the device. More specifically, this happens when the number of thread blocks is bigger than the number of cores on the device. [21] As mentioned earlier, the number of threads per block and blocks per grid are specified when launching the kernel.

Thread witing a thread block can cooperate by accessing the same data through available shared memory and coordinating global memory access. Synchronization points can be added such as each thread will wait for all other threads to reach a specific point in the code before continuing. [21]

Memory structure

CUDA threads can access data from multiple sources during execution, all shown in figure (?): [21]

- Local thread memory - individual to each thread
- Shared memory – per thread block, can be accessed by all threads contained in that block
- Global memory – can be accessed by any thread

Besides global memory there are also two additional read-only memory spaces: constant and texture memory. These come with optimizations for different types of accesses. [21]

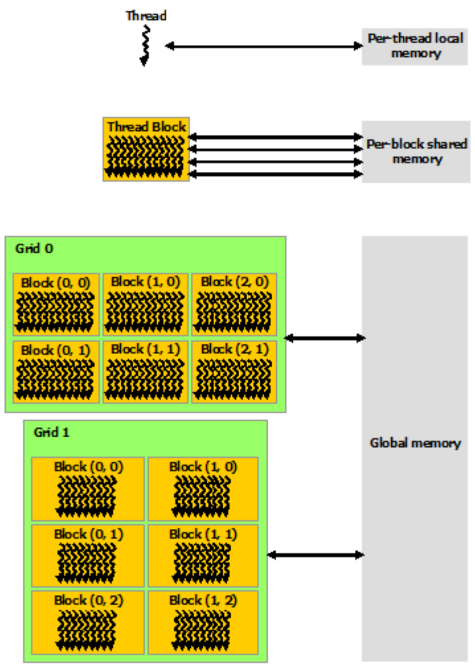


Figure 15 - GPU memory [21]

Workflow

A typicall workflow for a program using the GPU would look like this:

1. Memory is allocated for both CPU (host) and GPU (device)
2. Data is copied from host to device
3. Kernels are launched to process said data
4. Results are copied back from device to host

Hardware

NVIDIA GPU architecture is built around an array of multithreaded Streaming Multiprocessors (SMs). When a grid of thread blocks is launched, blocks are distributed to available multiprocessors. After thread blocks terminate others are created to occupate the multiprocessors. To manage this large amounts of threads an architecture called SIMT is used. [21]

The SMs creates and manages and executes threads in groups called warps. Most present day GPUs are built around a warp size of 32. Threads part of a warp start at the same time and from the same address of program memory but they are free to branch and execute different instructions from one another, although synchronization at a later point is recomanded. Besides the global ID each thread has it is also assigned an ID inside the warp. [21]

SIMT enables programmers to take two different paths when deciding the thread structure: they can either create threads that will run independentaly from one another or they can create data parallel threads designed to minmize divergence and coordinate memory access.

Threads executing the current instruction are called active threads while threads that are not on the current instruction are called inactive. Threads can be inactive for various reasons such as branching to a different line or simply being part of a warp that needs only some of the threads to be active. When multiple threads access the same memory location, atomic instruction is recommended and it will guarantee that writes or reads to that memory location are seralized but the order cannot be determined. [21]

Through a system called Hardware Multithreading, a state of the warp is maintained during the entire lifetime of a warp. [21] This makes switching between differents warps really efficient and in a scenario that the number of warps executed on an SM is smaller than the maximum possible, the SM can employ other warps to reside and be processed together but this is may be restricted by other constraints coming from registers and shared memory.

The maximum number of threads per block is computed using the following formula:

$$Ceil\left(\frac{T}{W_{size}}\right)$$ [21], where:

T = number of threads per block

Wsize = warp size

Performance

The main performance optimization goals are: [21]

- Achieving maximum utilization
- Optimizing memory access
- Optimize instruction usage
- Minimze memory thrasing

Depending on the particular bottlenecks of each application some of the strategies applied to reach these goals may yield different results. Optimizing memory access for a kernel that has poor instruction usage will not make much difference, for example.

At application level, the processing and operations that can be done serially should be assigned to the host and the device should be left with the parallel workloads. [21] This would imply that even parallel workloads that would run faster serially should be kept assigned to the CPU but in the end this comes down to the needs of the specific application.

When designing the thread structure blocks should be kept independent to each other in terms of data such that they will not be required to use the global memory as a synchronization mechanism but keep to syncrhonization features already implemented per block. [21]

When it comes to memory access optimizations, two data paths needs to be addresed:

1. Data transfers between host and device

   The main way to optimize data transfer is to group more smaller accesses into a single one. That means copying all the data to the device before the kernel runs and copying all the data back at the end. [21]

2. Memory access on device

   When a warp executes an instruction that accesses global memory, it coalesces the memory access into one or more memory transactions. In order to minimize the number of transactions and memory words transfered, data structures should be designed according to the computing capabilties of the device. [21] To make sure the memory access will transfer only the words needed for a particular data structure, allignment can be specified for each data structure

Improving instruction throughput can be done by: [21]

- using intrinsic arithmetic functions defined by the architecture (sin, pow, exp, etc)
- minimizing branching and flow control instructions (if, switch, etc)
- keeping synchronization points to a low as this instruction can impact performance by making the SM idle

For lower memory thrasing, constantly allocation and freeing of memory must be avoided. [21] That means critical sections in terms of performance should restrain from the operations mentioned above.

If the platform allows it, a type of managed memory allocation can be used which allows oversubcription, meaning that an application will be resident only when needed. [21]

4.2.3 Device parameters

The device used to run the implementation described in this thesis is GTX 1660 Ti (TU116), which is part of the Turing architecture family along with the RTX devices and has a compute capability of 7.5.

The following table describes the technical specifications that need to be taken into account when working with the programming model.

Specification	Capability
Maximum number of threads per block	1024
Warp size	32
Maximum number of blocks per SM	16
Maximum number of warps per SM	32
Maximum number of threads per SM	1024
Shared memory per MP	64KB
Local memory per thread	512KB
Constant memory	64KB

Table 1 - Device specifications [21]

5 Implementation

5.1 System design

The purpose of this application is to process Memcached-type commands, which imply storing, indexing and retrieving data. Because of this, it is important to first split and problem, define the architecture and decide tasks will be handled directly by the CPU and which tasks will be offloaded towards the GPU.

The architecture will need to support the following commands along with their specific constraints:

- SET(key, value) - Stores the value in memory at the location indicated by the key
- GET(key) – Returns the value stored earlier in memory at location indicated by the key
- DELETE(key) – Deletes value stored at the location indicated by the key

The key and the value will be formatted as strings of characters and their maximum size will be 512 bits.

Because of their size, it would be inefficient to send the packets to be stored directly on the GPU. A better approach would be to store them in CPU memory, assign a unique ID for each location and use the GPU as an index table to set and return these locations based on the key of the packet.

For constant time access and parallelization the GPU will store the data into a hash table and use Cuckoo hash to access it. The hash table will be divided into N buckets. Since Cuckoo hashing is used each element will have 2 available buckets to be assigned to and will find these buckets using 2 different hash values. The data structure used for each bucket will be a plain array that can store M(=8) elements.

A complete flow of SET-GET for a single key-value pair would look like this:

1. SET(key, value)
    a. Extract key and value from the packet
    b. Store them at the first available location on CPU memory
    c. Create an ID for the key and compute its 2 hash values
    d. Call the GPU kernel which will insert the location into the first or second bucket on the hash table and mark it using the ID
2. GET(key)
    a. Extract the key from the packet
    b. Create the ID for the key
    c. Call the GPU kernel which will search the 2 buckets for the ID and return the location
    d. Use the location to find the value stored in CPU memory

In a real workflow this would be massively parallelized, that meaning that multiple SET commands will be batched into a single GPU kernel.

The system design from a block level perspective is presented in figure (?).

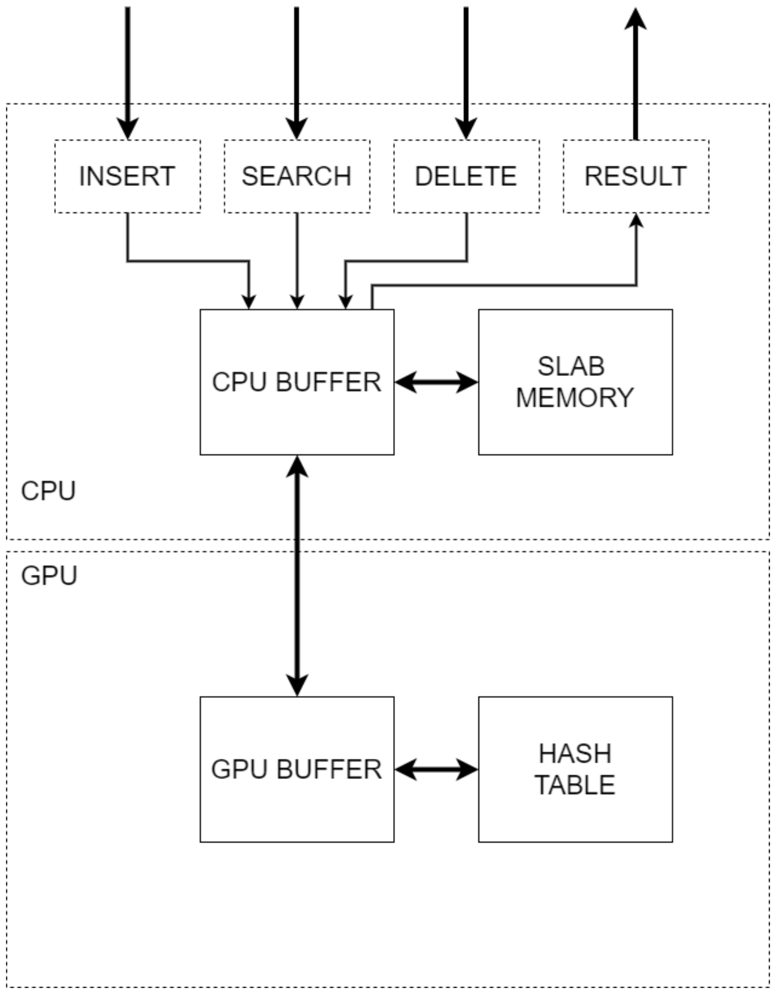


Figure 16 - System design

The input to the system is represented by INSERT, SEARCH, DELETE blocks. These are the requests that would normally be received through network, but in this implementation they are represented by text files which contain the commands and key or key and value pairs. The output of the system is the RESULT block, this too being represented through a text file.

The slab memory is used to store the values while the CPU buffer is used to create references to them. More specifically, based on the key the values of hash and signature are created and based on the location of the value in the slab memory a location descriptor is created. These are added in batches to the CPU buffer, transferred to the GPU buffer and then stored inside the hash table.

We can describe each of these blocks separately.

5.1.1 Input files

Inside the text files, key and value pairs are represented as 512 bit fields formatted as strings on two consecutive lines:

```
Key:   lavender_gray8C9a02Dcfa7E1B12aA2ff43b02CEf5BFca66b82BFECEd1BfC44
Value: 08dEbC1CDaB84E1DbfEE386b8DFf238498e1AbE9e07B72C64A788A876BB2dbB3
```

In this example the key is represented as color followed by random hex numbers while the data(value) is represented only through random hex numbers. Receiving requests is implemented as continuously reading from the input files and sending responses by continuously writing to the output file.

5.1.2 CPU/GPU Buffer

This structure is essentially the same both on CPU and GPU and is used to transfer data. It has the following definition:

```
typedef struct elem_buffer {
    key_search_t *key_search;
    key_insert_t *key_insert;
    key_delete_t *key_delete;
    location_t   *locations;
    uint32_t nr_search_keys;
    uint32_t nr_insert_keys;
    uint32_t nr_delete_keys;
} elem_buffer_t;
```

It contains array pointers for each of the key, insert and delete elements that need to be sent to the GPU to be processed along with the number of elements for each. There is a 4th array that contains either the location descriptors read from GPU or location descriptors that need to be written to the GPU. This array does not need a value to indicate the number of elements since it will be indicated by the number of search or delete elements depending on the case.

Further it is explained how the CPU buffer is filled in the case of insert jobs, although very similar mechanism happen both in the cases of search and delete.

We have the CPU buffer declared and preallocated:

```
cpu_buffer = (elem_buffer_t*)malloc(sizeof(elem_buffer_t));

cpu_buffer->key_search = (key_search_t*)malloc(MAX_SEARCH_JOBS *
sizeof(key_search_t));

. . .

cpu_buffer->locations  =   (location_t*)malloc(MAX_SEARCH_JOBS *
sizeof(location_t  ));
```

From there we can declare the function that will fill the buffer as taking pointers to the input file, the buffer itself and the slab memory. The slab memory and location descriptor generation will be described shortly after.

```
      void fill_insert_buffer(FILE** file_pointer, elem_buffer_t**
cpu_buffer, slab_block* slab[])
```

Inside the function body, a buffer is used to read a line of key and value from the input file.

```
      fgets(key_value_buffer, FILE_BUFFER_LEN, *file_pointer);
```

Then, the hash and signature values are computed using an XOR hash. It is a very simple way of hashing that requires splitting the input into N-sized chunks and sequentially applying the XOR function between the result and the next chunk. Result is initialized with the first chunk of data. In this case N is equal to 64 bits.

```
uint64_t hash_result;

hash_result = *(uint64_t*) (&key_value_buffer[0]);

for(int j = 8; j <= KEY_LEN - 8; j+= 8) {

    hash_result = hash_result ^ *(uint64_t*) (&key_value_buffer[j]);

}
```

The implementation of the hash function is shown in the code above. Since the input buffer is an array of chars each of 1 byte, we can take 8 of them and format them as an unsigned integer which would result in the 64 bits needed for each iteration of the hash function.

That is done simply by casting the base address of a chunk of 8 chars to be a pointer of type unsigned int.

Then, the 64 bit result can be split to form the hash and signature values.

```
(*cpu_buffer)->key_insert[i].hash      = (hash_t)       (hash_result >> 32);

(*cpu_buffer)->key_insert[i].signature = (signature_t)  (hash_result      );
```

This is repeated for a maximum of insert jobs which is 512 and then the number of keys is also set.

5.1.3 Slab memory system

The slab memory system is designed as a large number of slab blocks each containing a number of slab items. The format of the slab system and the location descriptor is presented in figure(?)

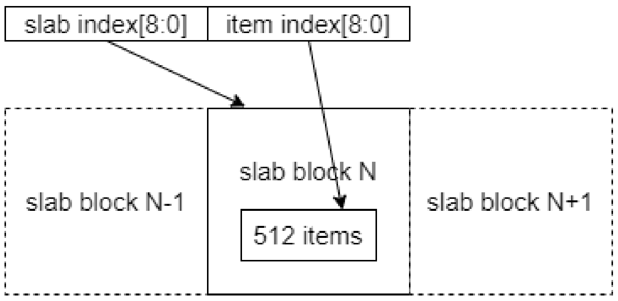


Figure 17 - Slab memory

It is defined as following:

```
typedef struct slab_item {
    char key[KEY_LEN+1];
    char value[VALUE_LEN+1];
} slab_item_t;

typedef struct slab_block {
    slab_item_t item[NR_SLAB_ITEMS];
    int occupied;
} slab_block_t;
```

For this implementation NR_SLAB_ITEMS has been chosen as 512 which is the size of an insert block and a total of 512 slabs blocks are instantiated, allowing for a maximum number of approximately 260.000 items to be stored.

We will describe how the slab memory system works for the same scenario described above in the case of the CPU buffer. In the code, they are filled as part of the same function.

For each insert block we first get the index of the first empty slab block and then we write each item:

```
int slab_index = get_free_slab(slab);
for(int i = 0; i < MAX_INSERT_JOBS; i++) {
strncpy((*slab)[slab_index].item[i].key, key_value_buffer, KEY_LEN);
}
```

After copying the whole insert block into the slab block, it is marked as occupied.

5.1.4 Data transfer

Data is transferred between the CPU buffer and GPU buffer through the following function:

```
void transfer_data(elem_buffer_t** host_buffer, elem_buffer_t**
device_buffer, direction_t dir);
```

It takes as arguments pointers to the CPU buffer and GPU buffers and based on the direction indicated by the 3rd argument it knows what data to transfer between the two buffers. If the direction is indicated as “host to device” then it will copy elements from each of the key, search or delete elements based on their associated counters and if the direction is “device to host” it will copy back the results (locations) found by the search kernel.

5.1.5 Memory allocations

In the case of the GPU, the allocation of the hash table is done at the start of the application and all its locations are initialized to 0. Additionally, the CPU and GPU buffer locations are also pre-allocated with a maximum number of item locations and then reused for each job. No other significant memory allocation is done until the application is closed.

5.2 GPU Design

5.2.1 GPU Hash table

The hash table will be defined with a size of 30 bits, meaning 1GB of total data. A Cuckoo hashing algorithm will be used to access the table with the particularity that both buckets associated to each key can be found in the same table instead of two different tables.

The hash table will be built as an array of buckets capable of holding 8 keys. Each key will have 2 hash values precomputed (called *hash* and *signature*) which will be used to point towards the two buckets in the following way:

- First bucket will be at the address indicated by the value of *hash*
- Second bucket will be at the address indicated by the value of *hash* ^ *signature*

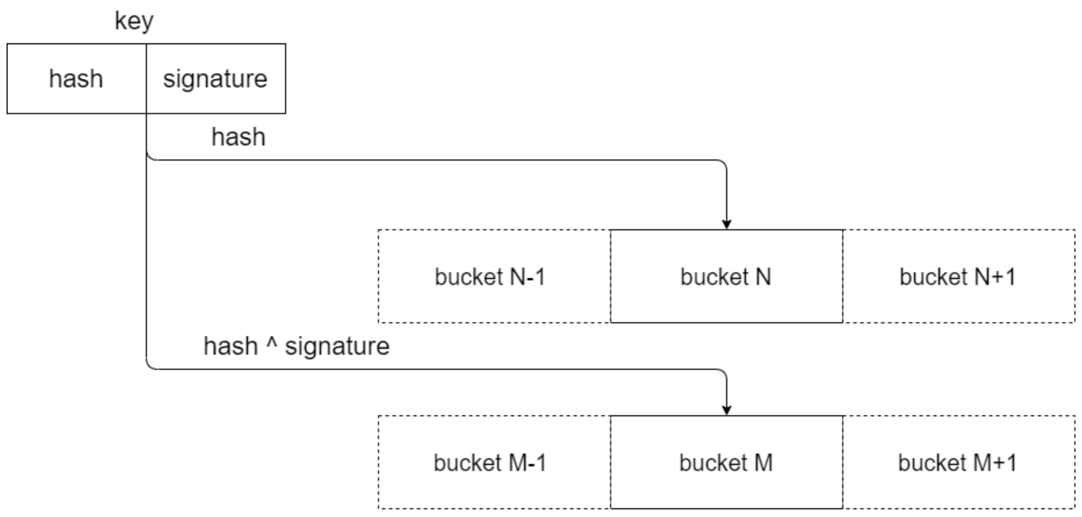


Figure 18 - Bucket addressing

The contents associated to a key that will be inserted in one of the 8 bucket slots will be the location of the key-value pair in CPU memory and an ID which is unique per key, in this case being the *signature* value.

When performing any of the search, insert or delete operations, a thread group consisting of 8 thread will be created for each key to scan each slot of the bucket. The same group will be responsible for both buckets assigned to every key, as show in figure (?).

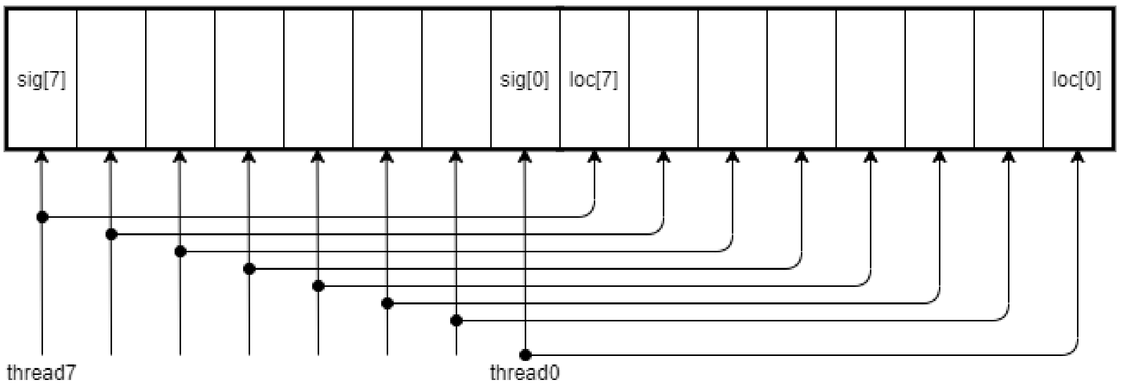


Figure 19 - Bucket access

In the case of insert, every thread will look for an empty spot and communicate with the other threads in its group to create a a list of all the threads that found an empty spot. After that a thread is chosen to insert the key (signature+location) based on a function, in this case the thread with the lowest ID.

For search and delete, every thread will look for the signature of the key in each of the 8 slots of the bucket and, if found, will either return the location in the case of search or delete the contents in the case of delete.

Since the hash table is built as an array of buckets with each holding 8 pairs of signature-location, each pair having size of 8 bytes, that means the total size of a bucket is 64 bytes and so we are left with a 24-bit address, meaning about 16 million different buckets.

Since kernels that access buckets will be called in parallel there is the possibility of conflicts. In the case of search and delete this will not be an issue since search only reads data and delete should act on different elements since, by design, there is no key duplicate. Only in the case of insert there is a possibility of write-write conflicts, meaning that a thread group could overwrite the key written by another thread group. Therefore, only one of the keys will be successfully inserted.

Since this application only acts as a cache and permanent storage is not a requirement, we can permit write conflicts as long as their probability is very low. As being described above, since we have approximately 16 million total buckets, assuming a hash function uniformity and a number of 512 items being written in parallel, we would have a 0.0032% chance of a conflict for any batch of inserts. In the context of this application this can be considered as a safe value.

5.2.2 GPU structures

For efficient memory access all values used within the hash table will be of constant size and defined as such to ensure alignment:

```
typedef uint32_t signature_t;

typedef uint32_t location_t;
```

Using these we can define the format of a single bucket inside the hash table:

```
typedef struct bucket {
    signature_t  signature[BUCKET_SIZE];
    location_t   location[BUCKET_SIZE];
}bucket_t;
```

In the current implementation BUCKET_SIZE is equal to 8, which means that it can store 8 signature – location pairs. The signature represents the unique ID that is generated based on the key and the location indicates where the value is stored inside CPU memory. Since each element can access 2 different buckets, it means that there are 16 total available locations, reducing the impact of hash collisions.

Knowing this, we can construct an insert element in the following way, where we have the hash value that will point to a bucket, we have the location that we need to store and the signature used to mark the location.

```
typedef struct key_insert {
    hash_t       hash;
    signature_t  signature;
    location_t   location;
}key_insert_t;
```

Since we are using Cuckoo hashing and need to have 2 available buckets for each element, once bucket will be accessed directly using the hash value and second one using of hash ^ signature.

Search and delete elements are constructed in a similar way:

```
typedef struct key_search {
    hash_t       hash;
    signature_t  signature;
}key_search_t;
```

```
typedef struct key_delete {
    hash_t       hash;
    signature_t  signature;
}key_delete_t;
```

5.2.3 Kernel architecture and thread communication

Since each element will need to access the 8 different locations of a single bucket to either look for the signature and return/clear the location in case of search and delete or look for an empty spot to write the location and signature in case of insert, we can assign 8 threads (a thread group) to a single element. These threads are responsible to compute which element they are assigned to and what is their offset i.e. what location they need to access inside the bucket, based on the global thread ID.

Additionally, threads inside a thread group need to be able to communicate between them to signal if they found the signature they are looking for in case of search/ delete or if they found an empty spot in the case of insert. This is done using a ballot function (__ballot_sync) which is available only inside a warp, which represents an architecturally defined group of threds that in this case is 32. Therefore, a warp will fit 4 threads groups. Since the ballot function is called for the whole warp, each thread will need to use a mask to communicate only with its threads group and this is done based on the element index and the offset which were defined earlier.

5.2.4 Search kernel

```
__global__ void gpu_search_key(
    key_search_t  *keys,
    location_t  *locations,
    bucket_t    *hash_map,
    int          no_items
    )
```

The search kernel takes as input a pointer to the search element array which will be used to access the hash table and return the locations found. As mentioned above each thread will first compute its global ID, the index of the element it needs to process together with the rest of the thread group and the offset inside its thread group. After that is done it can get a reference to the first bucket using the element's hash, check the signature and return the location if there is a match.

```
bucket_t *bucket = &(hash_map[first_hash]);
if (bucket->signature[bucket_offset] == key->signature) {
        locations[id] = bucket->location[bucket_offset];
```

After this is done each thread will call the ballot function on the same comparison evaluated above to check if any of the other threads found a match and, if not, repeat the same process for the second bucket.

```
__ballot_sync(ballot_mask, bucket->signature[bucket_offset] == key-
>signature )
```

As defined from the start, first hash value is already contained in the search element and the second is computed based on the signature.

```
int first_hash = key->hash;

int second_hash = key->hash ^ key->signature;
```

5.2.5 Insert kernel

```
    __global__ void gpu_insert_key(

        key_insert_t  *keys,

        bucket_t    *hash_map,

        int          no_items

        )
```

The insert kernel takes as input a pointer to the array of insert elements which then uses to access the hash map. After computing the necessary offsets in a manner similar to the search kernel, the ballot function is called by each thread group to determine which locations inside the bucket are available i.e. their signature is equal to 0. If there are multiple available locations the first one will always be chosen. Threads will always know what the chosen location is and what is then left is that each thread checks if it has the responsibility to write the location and signature to the chosen spot.

If there are no available spots the same procedure is repeat for the second bucket. If that one is also full, an element will be chosen to be replaced to its other available bucket for which the procedure will repeat. The element to be replaced is chosen based on the least significant bits of the current element’s signature. The procedure is repeated for a set number of tries and if no available location is found inside a bucket then the element is left evicted.

5.2.6 Delete kernel

```
__global__ void gpu_delete_key(
    key_search_t  *keys,
    bucket_t    *hash_map,
    int          no_items
    ) {
```

The delete kernel is similar to the insert one and takes as input a pointer to the key array which then uses the access the hash map. After computing the the necessary offsets in a similar manner to the previous two kernels, the first hash is computed and each thread looks at its offset inside the bucket for the signature and if there is a match both the signature and the location value are deleted.

After this first step, all threads use the ballot function to indicate if any of them deleted the element and, if not, they compute the hash for the second bucket and repeat the procedure. Then the kernel ends because there is nothing to return.

6 Results and Performance

6.1 Test setup

In order to test the application both in terms of performance and data integrity, a scenario in which a total of 32768 elements are processed is ran, this being the maximum number of search elements allowed per batch. Having an insert batch limit of 512 elements it means that a total 64 insert jobs will be launched.

The two text files used to generate the insert and search jobs are filled with random data (pairs of key-value) using a Python script. Additionally, a Python script is used to compare the output data written in another file to the original input.

Kernel setup:

Insert: 512 elements, 8 threads per element, 1024 threads per block, grid of 4 blocks.

Search: 32768 elements, 8 threads per elements, 1024 threads per block, grid of 256 blocks

6.2 Data integrity

As mentioned in chapter 5.2.1, a theoretical failure probability of 0.0032% is expected per each insert batch. Since in this test setup 64 insert jobs are launched a probablity, that probability should still be under 1%.

However, running this scenario in multiple tests with different seeds results in about 2-4 total failures per test which is a few orders of magnitude above what is expected theoretically. This can be attributed to a number of factors. First, a random input data was assumed but in reality the test used a similar key format (name of a colour + random HEX numbers) for all of the keys. Compared to the total number of 8-bit ASCII characters possible, this means that the range of the hash input could have been reduced by a maximum factor of 16. Additionally, since the hash function used(XOR) is not cryptographically secure, it means that in practice we can expect a significant number of collisions even for different input data.

Considering these factors, the behavior of the application in terms of failures is a lot closer to what we expect. Even so, the pass rate is on average above 99.99% which can be considered acceptable for a cache memory. Below is an output from one of the tests:

```
Number of matched items: 32766

Number of failed items: 2

Pass rate: 99.9939%
```

6.3 Performance

To generate an overall image of the system performance, memory allocation and memory transfer, NVIDIA Visual Profiler can be used. A snapshot of the results can be seen in figure (?).

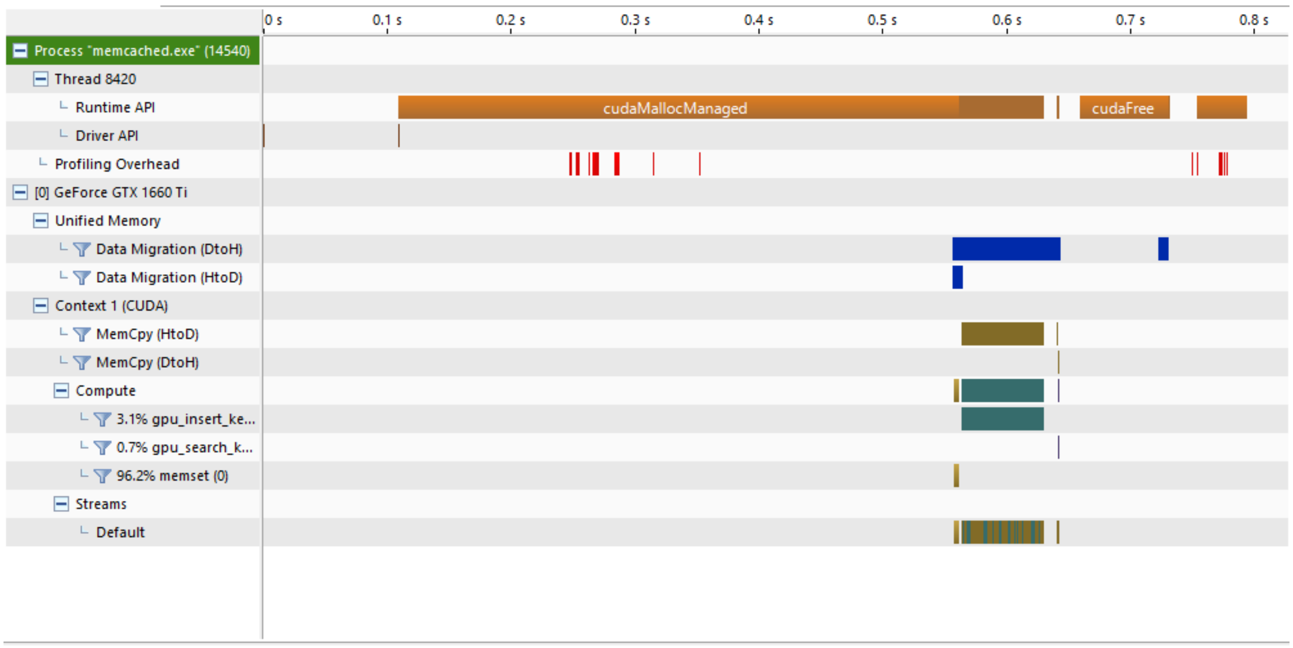


Figure 20 - Overall performance

As it can be observed in the figure, a significant portion of time is spent on allocating CPU and GPU memory – about 0.55 seconds. Since it was stated in one of the previous chapters that, based on the nature of this application, these memory allocations only happen when the application is started and should be insignificant compared to the total runtime. Therefore, they can be excluded from this analysis. This is also true for the deallocations that happen before the application is closed, which take about 0.2 seconds.

Considering these, we are left with about 1 second of active time in which we can see the 64 insert jobs followed by a single search one, as seen in figure (?).

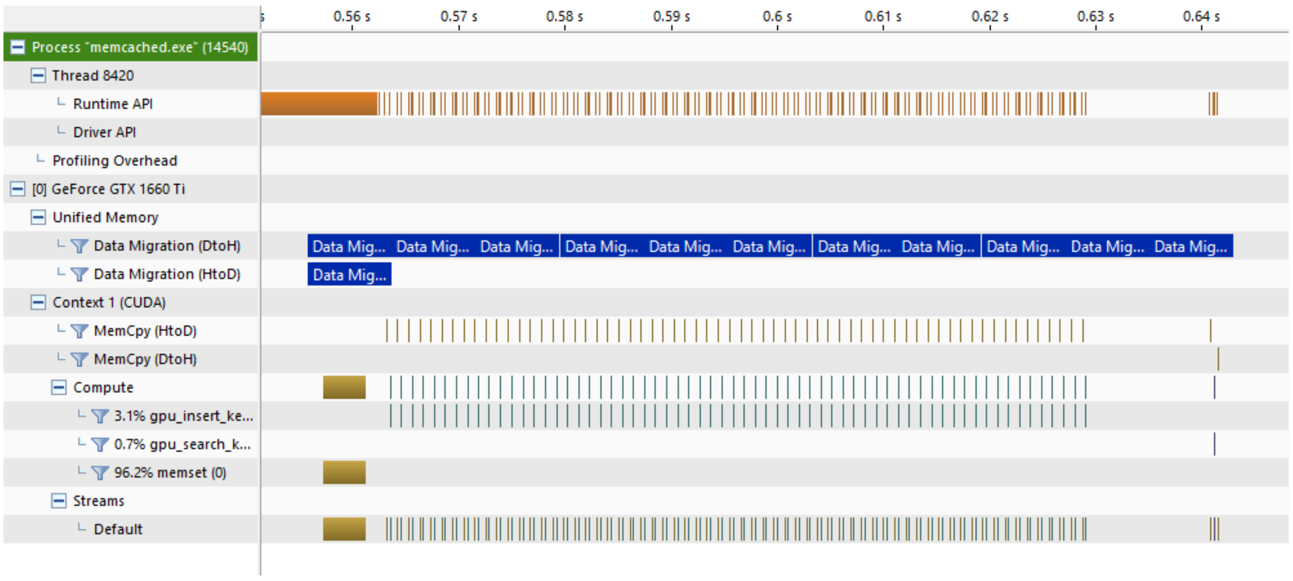


Figure 21 - Kernel performance

Insert kernel

In the case of the insert kernel, copying the buffer from CPU to GPU space takes 1.664 us according to the profiling results, while the kernel itself finishes in 3.36 us. Overall, across all 64 kernel launches, considering all the idle time and synchronization between them, there is a latency of 0.66s. This also includes the time spent on CPU to read input data and fill the buffers.

Search kernel

In the case of the search kernel, copying data is done in about 21.85 us because the buffer is significantly larger while the kernel itself runs in 30us. With the added time at the end (copying the buffer containing the locations back) at the end, we have a total of 0.73 ms spent on processing the search batch. CPU times is harder to approximate in this case since there is only one job launched but we can approximate a maximum of 0.01s of latency.

As we would expect, the search kernel, processing 64 times more data in parallel, is about 64 times faster. Again, as stated when the scope of the application was defined, about 99.6% of all accesses on this type of cache memory are GETs. This means that the data is written once and then read for significantly more times.

Therefore, the lower performance of SET commands as compared to GET is not that significant to the overall performance of the application.

The performance analysis per kernel that provides more information about how the GPU resources are used can be generated using NVIDIA Nsight Compute.

The following report was generated for the search kernel.

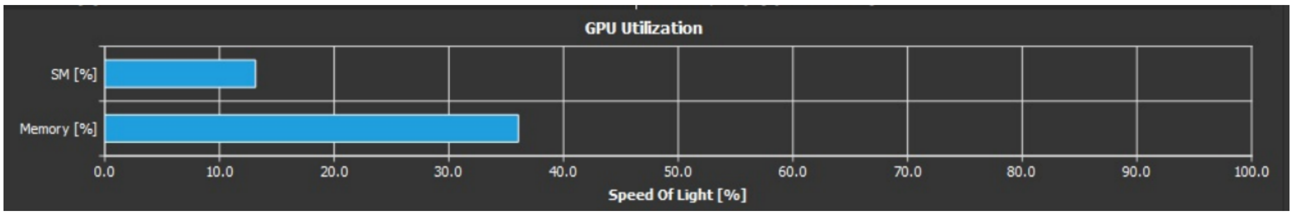


Figure 22 - Utilization

In figure (?) SMs (Streaming Multiprocessors) usage is shown as being around 15% while memory usage around 40%. As will be seen in later reports this is not actually an issue since the number of thread blocks launched simply do not need to be run on more SMs. On the other hand, increasing the size of the search batch will create more latency as the application will need to wait more time in between launching jobs.

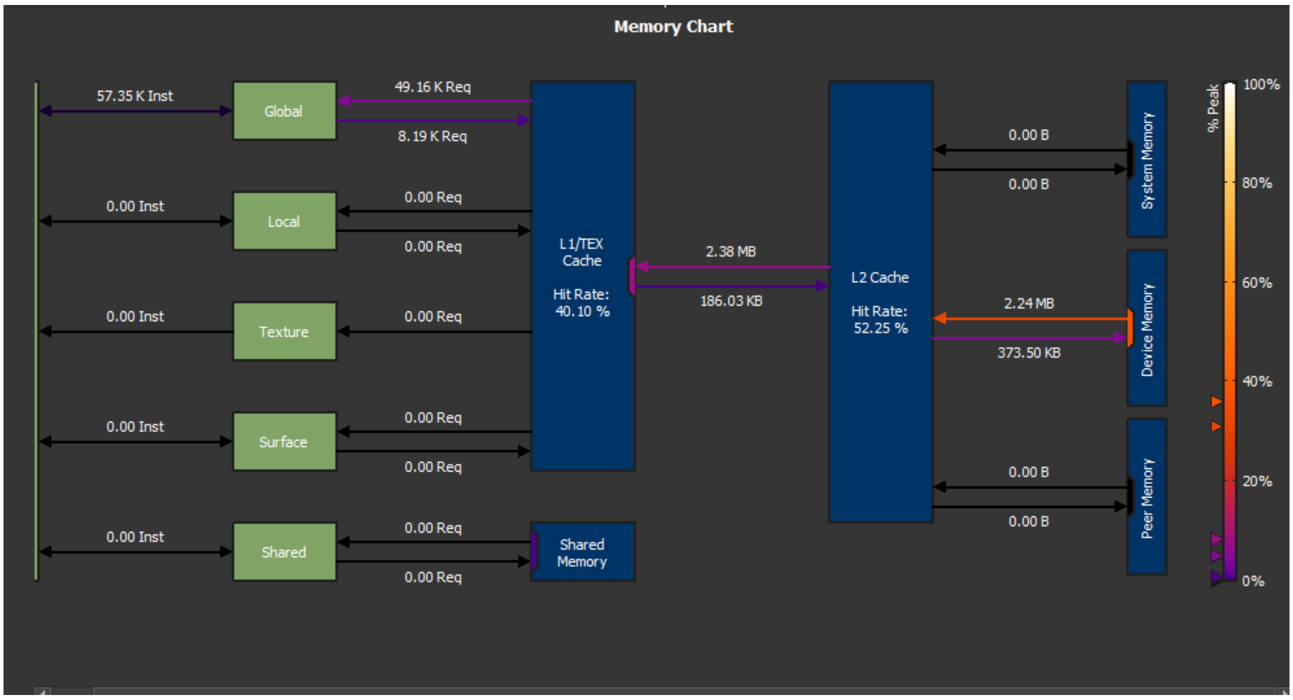


Figure 23 - Memory access

Looking at the memory chart, we can see that the kernel accessed only global memory while running. Considering the design of the hash table, it is intended for each bucket to reside fully in global memory, and thefore, as long as memory transfer does not represent a bottleneck, this is not viewed as an issue.

In numbers, memory throughput is around 95 GB/s or 18% of the maximum bandwidth.

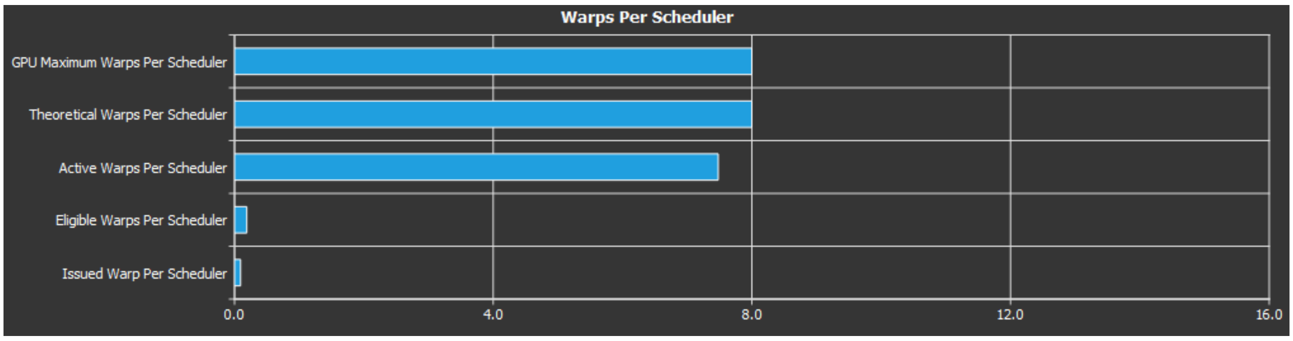
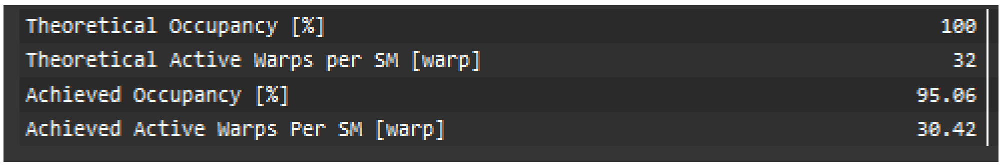


Figure 24 - Warp scheduler

In figure (?) we can see the behaviour of the warp scheduler. Although the SM utilization presented at the beginning of the chapter is low, the number of active warps per scheduler was close to the theoretical maximum possible. On the other hands, the number of eligible warps is poor. Eligible warps represent the subset of active warps that are ready to issue their next instruction. A low value can be caused by warp stalling and the possible cause in this case is the usage __syncthreads across all 4 threads groups in a warp. It is theoretically possible that a group of threads find their item in the first bucket and are forced to wait for other group of threads inside the warp to find their item in the second bucket, causing stalling.



Theoretical Occupancy [%]	100
Theoretical Active Warps per SM [warp]	32
Achieved Occupancy [%]	95.06
Achieved Active Warps Per SM [warp]	30.42

Figure 25 - Occupancy

In figure (?) we can see that the number of active threads per SM is close to the theoretical maximum of 32, so the achieved occupancy is 95%. This indicates that the workload is well balanced and latencies can be hidden more easily. Further, it can be seen that this is also the maximum possible even when varying the block size. The best theoretical block size is also the one being used – 1024 threads per block.

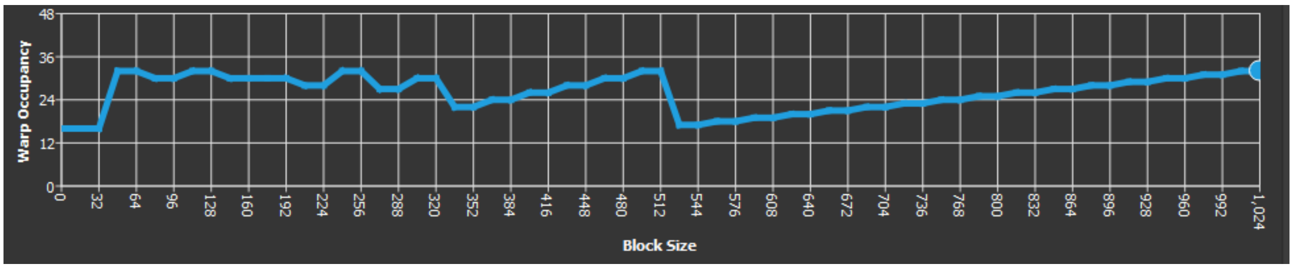


Figure 26 - Block sizes

6.4 Further work

The first possible improvement of the design of this application would be handle the write-write conflicts that can happen when running more insert jobs in parallel. A possible option would be to split the entire hash table into more partitions and add a prefix to the hash value of each item indicating what partition it is assigned to. This would also need to be taken into account when computing the second hash value either before with a similar prefix or directly inside the kernel. This is because both buckets needs to be inside the same partition.

Another possible improvement would be to redesign the hash table so buckets are no longer stored directly in global memory, but a thread block cooperates in shared memory when filling or reading a bucket and then the whole bucket is written to global memory.

An additional component that could be added to the design is a LRU (Least Recently Used cache) in order to manage evictions. In the current implementation this process would happen automatically but the eviction is random, while a component such as this one can evict the least used items.

Finally, for the case of receiving requests and sending back responses, network support could be added.

Conclusions

The initial goal of this project, that of exploring an alternative implementation of an In-memory Key-value store using the parallel capabilities of a GPU has been met.

The most important metrics that need to be considered when deciding the reliability of this kind of application, performance and data integrity, have been analysed as standalone since they cannot be compared to a real system or implementation because of the lack of network support.

Overall, the advantages of this kind of implementation have proven to be:

- High throughput
- Energy efficiency
- High memory capacity of the addressing table

On the other hand, the main disadvantage is that when trying to increase the throughput by accumulating more jobs, latency is also increased. In the end, it comes down to a trade-off between the two.

Bibliography

[1] Nikhil Dasharath Karande, “A Survey Paper on NoSQL Databases: Key-Value Data Stores and Document Stores” in *International Journal of Research in Advent Technology* no2, 02/2018

[2] Heli Shah, “A Brief Introduction to Memcached with its Limitation” in *International Journal of Engineering Research & Technology,* vol 3, issue 2, 02/2014

[3] Vijay Chidambaram, “Performance Analysis of Memcached”

[4] Memcached website – https://memcached.org

[5] James T Longston, “Enchancing the Scalability of Memcached” on software.intel.com

[6] Brian Agnes,“A High Performance Multi-threaded LRU Cache”, Code Project Article

[7] Mateusz Berezecki(Facebook), “Many-core Key-Value Store”

[8] Joseph Sterling Grah, “Hash functions in cryptography”, Universitet of Bergen, 06/2008

[9] Azman Samsudin, “Omega Network Hash construction” in *Journal of Computer Science* no 5

[10] Hash table lecture, https://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf

[11] Peter Nimbe, “An Efficient Strategy for Collision Resolution in Hash Tables”, in *International Journal of Computer Applications*, 08/2014

[12] Dapeng Liu, “Comparison of Hash Table Performance with Open Addressing and Closed Addressing: An Empirical Study” in *International Journal of Networked and Distributed Computing* 01/2015

[13] Nikolas Askitis, “Fast and Compact Hash Tables for Integer Keys”, School of Computer Science and Information Technology, RMIT University

[14] Dominik Köppl, “Fast and Simple Compact Hashing via Bucketing”, Department of Informatics, Kyushu University, Japan Society for Promotion of Science

[15] Emil Sit, “Storing and Managing Data in a Distributed Hash Table”, PhD Thesis, 06/2008

[16] Rasmus Pagh, “Cuckoo hashing”, Department of Computer Science, Aarhus University

[17] Rasmus Pagh, “Cuckoo Hashing”, IT University of Copenhagen

[18] Rasmus Pagh, “Cuckoo hashing”, in *Basic Research in Computer Science,* 08/2001

[19] Dan A. Alcantara, Real-Time Parallel Hashing on the GPU

[20] SMhasher GitHub pagehttps://github.com/rurban/smhasher

[21] NVIDIA C++ Programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html