Summary of the dissertation thesis of student: Cuța I. Ionuț Alexandru, group 421-ACES

Master program: Advanced Computing in Embedded Systems, 2021

Parallel implementation of a Memcached system using CUDA architecture

Thesis advisor: S.L. Dr. Ing. George Valentin STOICA

Project objectives: The standard Memcached architecture was analysed with an emphasis on the design of each block (hash table, slab memory), and possible performance improvements. This analysis indicated that the only and most fit component to be accelerated using a GPU is the hash table, leaving all other components to be handled by the CPU. Because of that, different hash table design patterns were studied first from the serial/sequential approach of handling conflicts, this resulting in the decision for Cuckoo hashing to be used. Then, while exploring different approaches for a parallel hash table and borrowing concepts and solution from each, a two-function parallel Cuckoo hash table with fixed bucket size was chosen. The two hash functions were generated using a variation of XOR hash. Lastly, the GPU architecture was analysed along with the specific parameters of the device used for running the implementation, NVIDIA GTX 1660 Ti.

For the implementation, the set of accepted commands (SET, GET, DELETE) together with the system architecture have been defined. The CPU would be repsonsible for collecting jobs from the input, storing values in a slab memory and filling up a buffer with the hashed keys of said values and their location in memory, in the case of set, or just the hashed keys from the input in the case of GET. Then the CPU would also be responsible for sending the data towards the GPU to be processed inside the index table/ hash table while also reading data back and sending it towards the output.

Implementation and results: The slab memory was designed to store the key/value pairs in a layout of 512 slab blocks of 512 elements, resulting in 2^18 total elements that can be stored. Based on the block index and the offset a location descriptor is generated which is added to the CPU buffer. Based on the key two 32-bit hash values are generated through the result of a 64-bit XOR hash function. The hash table then stores values of hash-location in buckets of size N=8. Since Cuckoo hashing is used, each element can be stored in either of 2 buckets pointed to by their two hash functions. Elements are launched in parallel to be processed inside the hash table by the GPU and this is done by assigning a group of N=8 threads for each element, which checks each slot of the bucket in parallel for an element of empty location. These threads use various synchronization and communication mechanism provided by the CUDA environemnt. The hash table was defined to have a size of 1 GB or 2^24 buckets.

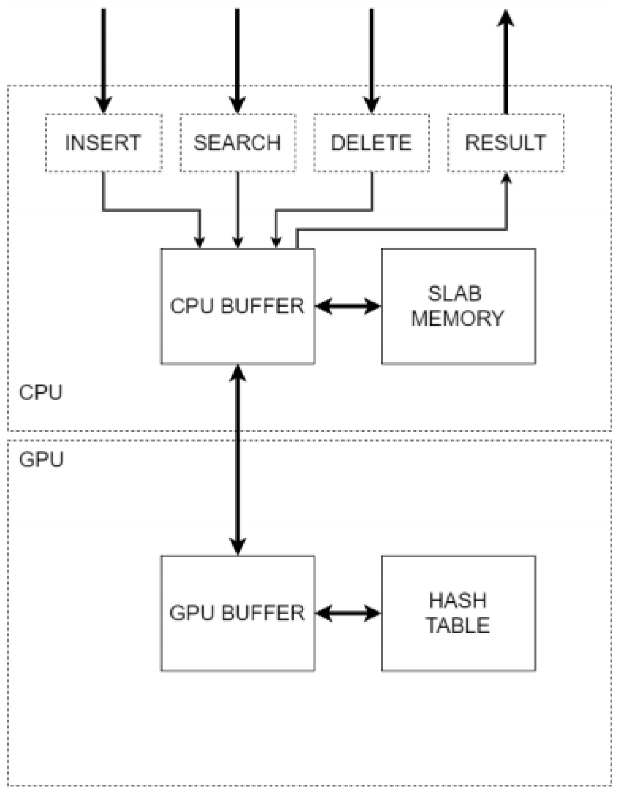
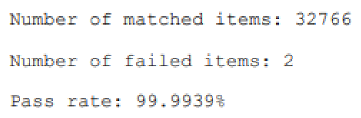


Figure 1 - System design

For a very low collision probability a number of N_i = 512 maximum insert jobs launched in parallel was chosen while for search the value can go up to N_s = 32768 jobs processed in parallel. A test setup to both check data integrity and performance analysis was built using two text files as input, both generated using a Python script. Additionally, an ouput file is filled by the system and another Python script is used to match and compare data. A kernel configuration of 8 threads per element, 1024 threads per block and maximum 4 blocks per grid was picked for insert, while a configuration of 8 threads per element, 1024 threads per block but a maximum of 256 blocks per grid was picked for search jobs.

Data integrity, while theoretically being 100% in most random cases for this scenario, resulted in an average value of over 99.99%. This can be attritubted to a number of factors but mainly to the poor characteristics (uniformity, collisions) of the hash functions and the similarities in input data.



```
Number of matched items: 32766

Number of failed items: 2

Pass rate: 99.9939%
```

Figure 2 - Data integrity results

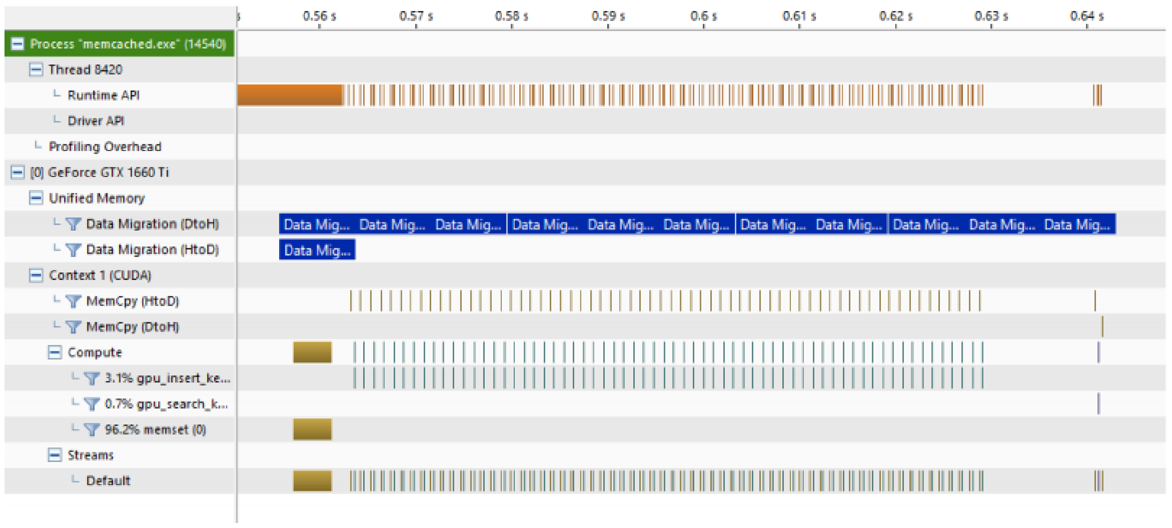


Figure 3 - System performance

In terms of performance, from a system perspective, an insert kernel finished in 3.36 us according to the profiling results, this meaning that, considering a total of 64 insert kernel launches and also accounting for data transfers, delays and synchronization there is a latency of 0.66s for a total of 32768 insert jobs. For the same number of jobs, search kernels performed, as expected, approximately 64 times faster with a 0.01s latency. Considering the estimated ratio of GET to SET in a real system is 99.6%, the poor performance of insert kernels is not significant to the overall performance of the system.

Profiling the kernels themselves has shown that the GPU utilization is around 15% for SM usage and 40% for memory usage. This shows that theoretically more thread blocks could be launched but this would also increase total latency.

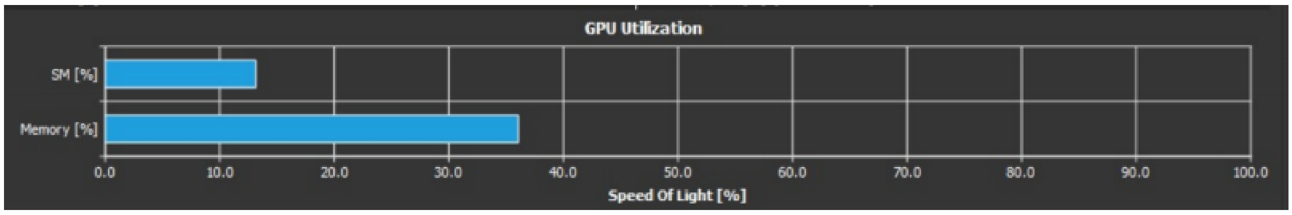
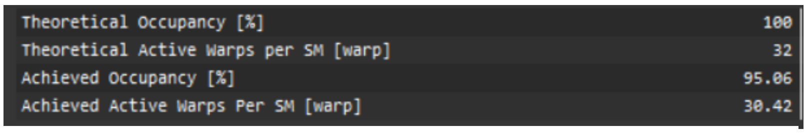


Figure 4 - GPU utilization

Per warp, the achieved occupancy was 95.06% with a number of achieved warps per SM was 30.42, not far from the theoretical maximum of 32, while the number of active warps at a given time was lower than expected. This means that the workload is well balanced but the number of active warps per scheduler is low because of the syncrhonization and thread idling.



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

Figure 5 - Occupancy

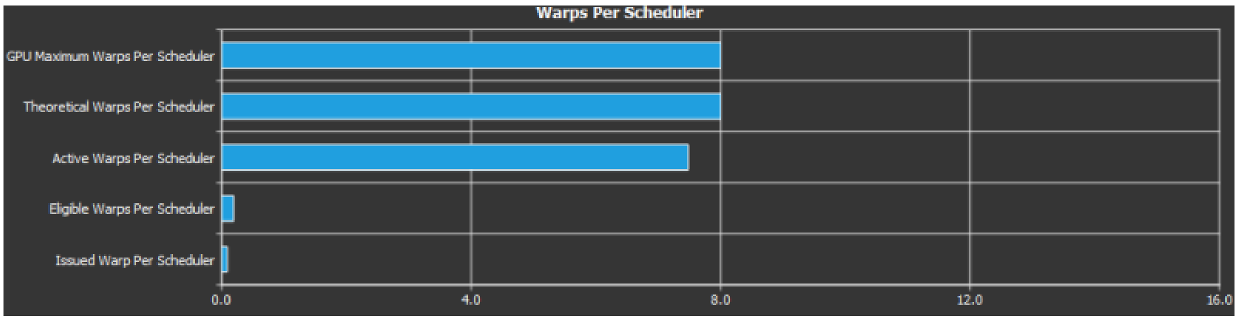


Figure 6 - Active warps

Profiling has also revealed that the block size used in this scenario results in the highest occupancy.

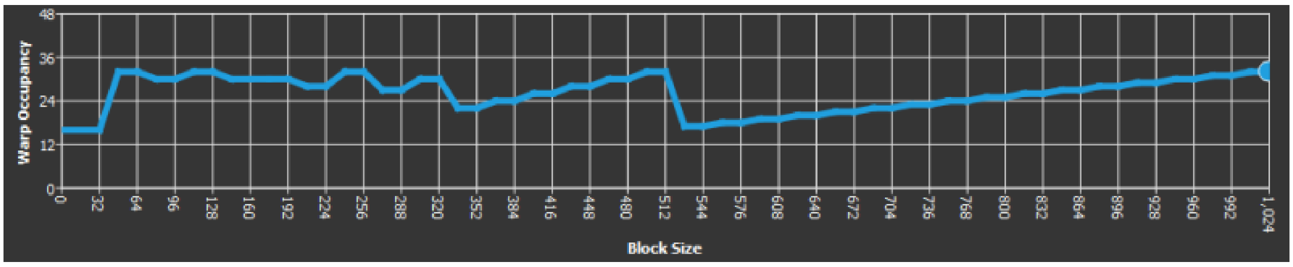


Figure 7 - Block size

The performance and data integrity analysis point to the following conclusions:

- The main advantages of accelerating a Memcached system using a GPU is high throughput achieved with a high memory capacity and energy efficiency
- The main disadvantage is the high latency inserted by waiting for jobs to fill the buffer
- Further work to improve this design can include: processing data in a pipeline, hash table redesign to lower the number of accesses to global memory, adding network support