Avi Kapadia, Aadit Trivedi
April 3, 2025
CS 6423

# MVCC Optimization Through GPU Utilization

GitHub Repo Link: https://github.com/kappavi/gpu_enhanced_mvcc/

**Context** - Based on feedback we have received from the initial proposal, we maintain the following objective: We plan to use the parallel computing capabilities of GPUs to accelerate MVCC-related operations, such as garbage collection and transaction visibility checks. We aim to explore extensions to MVCC using GPUs, which can provide insights into how GPU architectures may benefit concurrency control in databases.

**Milestones**

We have broken our project down into accomplishable steps, or milestones:

1. For our first milestone, we will design a basic multi version concurrency control implementation (note: at first, this will not have any CUDA-related optimizations).
2. The next step is utilizing GPU's parallel processing abilities to enhance multi version concurrency control. We will optimize the insertion and updates of multi-transactions as opposed to single transactions.
3. Finally, we will connect our multi version concurrency control implementation to a simple database and allow for transaction I/O.

**(Completed) Milestone 1**

We went ahead and implemented a basic multiversion concurrency control system.

```
struct MVCCVersion {
    int value;
    int begin_ts;
    int end_ts;
    bool is_committed;
};
```

We created a struct for MVCCVersion which includes an int (object id), begin and end timestamp, and a boolean for whether or not the version is committed. We derived this from our understanding from class.

```
class MVCCStore {
private:
    std::unordered_map<int, std::vector<MVCCVersion>> versions;
    std::mutex mutex;
    int current_timestamp;
```

Additionally, we created a MVCCStore class that saves the versions across multiple different objects, saved in the versions table. The MVCCStore class includes functions such as write, read, read_at_ts (reads value at a given timestamp, commit, and rollback).

We tested our MVCC implementation on single transactions for a single object, proof of which can be found in the main() section of our C++ code.

**(In Progress) Milestone 2**

To enhance our system with CUDA-based optimizations, we plan on doing a few things.

1. Kernel-oriented development: kernels can help execute transactions concurrently which can help reduce runtime especially when a lot of different objects are being inserted/updated to our database.
2. Parallel garbage collection: a garbage collection system that runs on kernels in parallel would be a significant improvement to reducing runtime for cleaning up old/outdated versions. We begun planning this out and writing pseudocode but need more time to properly develop our kernel for this.

```
void garbage_collection(int window) {
    // 1. get number of versions
    int num_versions = d_versions.size();
    // 2. create a vector of booleans to store which versions to delete
    thrust::device_vector<bool> d_to_delete(num_versions, false);
    // 3. launch the garbage collection kernel
    // 3.1 calculate the number of blocks needed

    // 4. wait for threads to finish
    // 5. delete the versions with delete flag set to true

}
```

**(Not yet started) Milestone 3 -** We will focus on connecting our MVCC protocol to a simple database (like a text file or CSV) after its implementation is mostly finished in Milestone 2.

**Revisions**

Initially, we intended to have some basic CUDA implementation completed at our 75% milestone, but we took some time properly setting up the MVCC structure and did some basic framework for the kernel setup. We will be engaging in our CUDA implementation in the next few steps.

Right now, we are around **45%** complete with our project–the setup was a bit slower than expected, but we have high hopes for what the future project will turn out to be.

**Conclusion**

As outlined, our next step is finalizing the CUDA kernel framework setup, after which we can begin actively implementing GPU optimizations to enhance multiversion concurrency control (MVCC), specifically targeting parallel processing of insertions and updates across multiple transactions.

Achieving successful CUDA implementation for transaction visibility checks and garbage collection operations will represent our key milestone, and it will point to how we are reaching our initial optimization goals. Additionally, we plan to incorporate performance measurement metrics to clearly quantify CPU and GPU efficiency differences, drawing from similar frameworks we've previously used in our respective GPU-focused projects from last year.

So far, we've maintained alignment with the original scope proposed, despite an extended setup phase.