ChatGPT

План рефакторинга на полностью GPU

В этом разделе представлен подробный поэтапный план, как избавиться от текущих «костылей» с возвратом на CPU при моделировании 7-дневного цикла. Предлагаемые изменения опираются на **лучшие практики** использования **FLAME GPU** (PyFLAMEGPU) и устраняют ручную загрузку суточных данных (налёта и квот) на каждом шаге. В итоге вся логика будет выполняться на GPU, а CPU будет задействован только один раз для начальной загрузки данных и финального экспорта результатов.

1. Загрузка данных в GPU память (MacroProperty) единым блоком

Текущее положение: Сейчас данные MacroProperty1/3/4/5 загружаются из ClickHouse на хост (CPU) и частично копируются в GPU вручную перед каждым шагом. В частности, суточные налёты (MP5) и квоты (MP4) на каждый день берутся с CPU. Это видно в коде: перед каждым шагом модель извлекает из `mp5_maps` и `mp4` значения и через `sim.getPopulationData` / `setPopulationData` и `sim.setEnvironmentProperty` переносит их на устройство [1] [2].

Решение: воспользоваться встроенными средствами **Environment Property Arrays** FLAME GPU для хранения **всех записей MP4 и MP5 целиком** в глобальной памяти GPU. Лучший подход – сразу при инициализации модели загрузить все необходимые макропараметры в окружение модели (`model.Environment()`), чтобы потом агенты могли считывать их напрямую:

- **MP4 (ежедневные целевые квоты эксплуатации)**: загрузить таблицу *flight_program_ac* как массивы среды. В документации Transform отмечено, что MacroProperty4 содержит 4 000 записей (4000 дней) и 8 полей, загружается как Environment Property Arrays [3] [4]. В коде загрузчика MacroProperty4 это реализовано: через `model.Environment().newPropertyArray...` создаются массивы `macroproperty4_field_<id>` длиной ~4000 элементов каждый, заполненные данными MP4 [5] [6]. Необходимо убедиться, что поля `ops_counter_mi8` и `ops_counter_mi17` (field_id 74 и 75) доступны в окружении как, например, `macroproperty4_field_74` и `macroproperty4_field_75`.

- **MP5 (ежедневные часы налёта)**: аналогично, загрузить *flight_program_fl* (MacroProperty5) в виде Environment Property Arrays. Согласно документации, MP5 содержит ~1 116 000 записей (комбинации борт × дата) и 4 поля: `dates`, `aircraft_number`, `ac_type_mask`, `daily_hours` [7]. Загрузчик MacroProperty5 читает все записи с `ORDER BY dates, aircraft_number` и создаёт массивы среды `macroproperty5_field_81` (даты), `..._82` (бортовой номер), `..._84` (суточный налёт) и т.д. длиной 1 116 000 элементов [8] [9]. Таким образом, **все суточные налёты для всех бортов сразу находятся в памяти GPU**.

  Примечание: Подобный подход уже заложен в проекте – в сводке компонентов указано, что MacroProperty4 и 5 загружены как Environment Property Arrays [10]. Наша задача – действительно использовать эти массивы на GPU, вместо того чтобы каждый день передавать отдельные значения с CPU.

Преимущества: Единоразовая загрузка всех необходимых данных на GPU позволяет обращаться к ним на каждом шаге без участия CPU. Время единственной загрузки ~10 с для ~1.13 млн записей [11], что значительно эффективнее, чем 7 раз вычитывать и копировать части данных на CPU. Кроме того, мы устраняем риск расхождений, так как GPU будет использовать единый **источник правды** данных, загруженный в начале [12].

2. Инициализация агентов с доп. переменными для быстрого доступа к данным

Текущее положение: При создании агентов в `sim_master.py` каждому агенту задаются базовые поля (psn, partseqno_i, status_id, и т.д.) [13] [14]. Однако в текущей реализации отсутствуют связи агента с его суточными записями налёта, кроме как через `aircraft_number`. В результате для обновления `daily_today_u32` / `daily_next_u32` на GPU агенту пришлось бы выполнять поиск по всей таблице MP5.

Решение: Добавить при инициализации **индексные переменные**, чтобы агент сразу "знал", где искать свои часы налёта в глобальных массивах. В частности:

- **frame_rank / индекс борта:** Для каждого планера (группы 1 или 2) определим порядковый номер среди всех бортов. Например, отсортируем уникальные `aircraft_number` всех планеров и присвоим им индексы 0,1,2,... (такой порядок совпадает с сортировкой записей MP5 по aircraft_number внутри каждого дня). Этот `frame_rank` сохраним в агентную переменную (например, `agent.newVariableUInt("frame_idx", 0)`). Для агентов-планеров значение `frame_idx` будет уникальным, а для агентов-агрегатов (деталей) – равным индексу борта, на котором они установлены (если `aircraft_number != 0`). Если у детали нет привязки к борту (например, хранится на складе), `frame_idx` можно оставить 0 или пометить особым значением, но в расчётах он не понадобится, так как статус не «2» (не в эксплуатации).

- **frame_count / общее число бортов:** Также сохраним общее количество бортов (планеров) – либо как константу в коде, либо как свойство окружения. Например, добавим `env.newPropertyUInt("frame_count", Nframes)`, где Nframes – число уникальных `aircraft_number` среди групп 1 и 2. В нашем примере Nframes ≈ 279 (из 1 116 000 / 4000 суток) [15]. Это значение потребуется для индексирования по дням.

- **Переменные daily_today_u32 / daily_next_u32 в агентах:** Они уже объявлены в модели [16], и мы сохраним их (по-прежнему UInt). Также оставляем `ops_ticket` и другие поля, которые уже есть. Новый код GPU будет обновлять эти поля **внутри симуляции**, а не через CPU, поэтому они должны быть проинициализированы нулями перед запуском (что и делается сейчас).

Пример реализации при инициализации (на Python): После создания списка уникальных бортов и их индексов, присвоим агентам frame_idx:

```
# Предполагаем, что frames_index_map: dict {aircraft_number -> frame_index}
заранее вычислен
agent_desc = model.model.getAgent("component")
av = pyflamegpu.AgentVector(agent_desc, n_agents)
```

```
for i, r in enumerate(mp3_rows):
    ai = av[i]
    ac_num = int(r[idx['aircraft_number']] or 0)
    ai.setVariableUInt("aircraft_number", ac_num)
    # ... заполнение других переменных ...
    frame_idx = 0
    if ac_num and ac_num in frames_index_map:
        frame_idx = frames_index_map[ac_num]
    ai.setVariableUInt("frame_idx", frame_idx)
# ... после цикла:
sim.setPopulationData(av)
```

Таким образом, каждый агент «знает», под каким индексом в массивах MacroProperty5 хранится информация о его борте.

3. Добавление счётчика дня в среде и GPU-циклирование по суткам

Текущее положение: В старой реализации цикл по дням организован на Python: `for D in days_list: ... sim.step() ...` [17] [18]. GPU-модель при этом не знает номера текущего дня – он учитывается на CPU при подготовке входных данных. Чтобы перенести этот цикл внутрь GPU, нужно отслеживать прогресс дней на уровне модели.

Решение: Ввести **глобальный счётчик дня** в среде модели и соответствующие GPU-функции для его обновления:

- **Свойство среды** `current_day`**:** создадим в `model.Environment()` макропеременную `current_day` типа UInt, начальное значение 0 (соответствует D0 – начальная дата версии) [19]. Объявляем её через `env.newMacroPropertyUInt32("current_day", 0)`. Выбираем MacroProperty (а не просто Property), чтобы её можно было изменять на устройстве – макропеременные поддерживают атомарные операции и обмен (exchange) [20] [21].

- **Функция** `increment_day` **на GPU:** добавим агентную RTC-функцию, которая будет запускаться **в конце каждого шага** и увеличивать `current_day` на 1. Например:

```
FLAMEGPU_AGENT_FUNCTION(increment_day, flamegpu::MessageNone,
flamegpu::MessageNone) {
    if (FLAMEGPU->getVariable<unsigned int>("idx") == 0u) {
        // Увеличиваем global current_day на 1 (atomic)
        auto day = FLAMEGPU->environment.getMacroProperty<unsigned
int>("current_day");
        day++;
    }
    return flamegpu::ALIVE;
}
```

Здесь проверка `idx == 0u` гарантирует, что только один агент (первый) выполнит инкремент, чтобы избежать повторения. Использование `MacroProperty` позволяет безопасно изменить значение [19]. После выполнения этого слоя в конце шага все агенты «увидят», что `current_day` увеличился к началу следующего шага.

- **Количество шагов симуляции:** вместо внешнего цикла Python мы можем задать `sim.step(n_days)` или просто вызывать `sim.step()` в цикле без дополнительной логики. Так как теперь все обновления происходят внутри модели, **нет необходимости на каждом шаге возвращаться на хост**. Например, можно вызывать `sim.step()` 7 раз подряд в Python без вставки данных – модель сама будет переключать дни. (Альтернативно, можно настроить `CUDASimulation` на фиксированное число шагов, но явный цикл читабельнее.)

Инициализация current_day: Перед запуском симуляции (перед первым `sim.step()`) убеждаемся, что `current_day = 0` – то есть начнём с D0. Если нужно, можно выполнить отдельный шаг для D0 (как сейчас), либо сразу перейти к D1. В текущем коде D0 уже симулируется одним шагом `sim.step()` перед циклом [22]. Мы можем сохранить эту логику: выполнить первый шаг (D0) отдельно, а затем запустить цикл GPU для D1..D7. **Важно:** на шаге D0 *не будет* квоты на D1, так как она ранее устанавливалась на CPU. В новой модели мы можем либо: - Либо не выдавать билеты на D1 в шаг D0 (т.к. `dt`=0 у всех агентов, это не повлияет – никто не запросит билет). - Либо перед первым шагом вручную прописать `current_day = 0` и `remaining_ops_next_* = 0` (или не инициализировать квоты). После шага D0, `current_day` станет 1 и далее все пойдёт по общей схеме.

Таким образом, цикл суток будет полностью контролироваться на GPU: `current_day` будет меняться автоматически, что устраняет «ручное» управление днями с хоста.

4. GPU-расчёт суточных часов налёта для агентов

Текущее положение: Ранее расчёт массива `daily_today`/`daily_next` выполнялся на CPU функцией `build_daily_arrays` с использованием данных MP5 [23]. Затем эти значения **копировались в каждого агента** через `sim.getPopulationData` / `setPopulationData` [24]. Это дорогостоящая операция (двойной перенос ~7k агентов * 7 дней) и главный источник CPU-фолбэка.

Решение: Реализовать **агентную функцию на GPU**, которая для каждого агента берёт нужные ему часы налёта из глобальных массивов MP5 и записывает в `daily_today_u32` / `daily_next_u32`. Мы можем использовать две альтернативные стратегии – приведём обе, и выберем наиболее эффективную:

- **(A) Прямой доступ агента к Environment Property Arrays:** Благодаря подготовленным индексам `frame_idx` и `frame_count`, каждый агент знает положение своей записи. Напомним, что MacroProperty5 загружена отсортированной по датам. Поскольку данные `flight_program_fl` отсортированы по (дата, борт), все записи за один день идут подряд, а порядок бортов внутри дня фиксирован (по возрастанию номера). Таким образом, если всего N_bortов (frame_count) и каждый день все борты имеют запись, индекс нужной записи можно вычислить как:

$$ \text{index} = \text{current_day} \times N_{bort} + \text{frame_idx}.$$

Этот индекс указывает на позицию суточного налёта данного борта в массивах `macroproperty5_field_*`. Соответственно, для следующего дня:

$$ \text{index}_{next} = (\text{current\_day} + 1) \times N.$$ + \text{frame_idx

Используя эти формулы, пишем RTC-функцию, пробегающуюся по агентам:

```
FLAMEGPU_AGENT_FUNCTION(update_flight_hours, flamegpu::MessageNone,
flamegpu::MessageNone) {
    unsigned int status = FLAMEGPU->getVariable<unsigned int>("status_id");
    if (status == 2u) {
        unsigned int frameIndex = FLAMEGPU->getVariable<unsigned
int>("frame_idx");
        unsigned int day = FLAMEGPU->environment.getMacroProperty<unsigned
int>("current_day").get();
        // Вычисляем базовый индекс для текущего дня в массивах MP5
        unsigned int N = FLAMEGPU->environment.getProperty<unsigned
int>("frame_count");
        unsigned int base = day * N + frameIndex;
        // Берём часы налёта за сегодня и за завтра
        unsigned int hours_today = FLAMEGPU-
>environment.getProperty<unsigned int>("macroproperty5_field_84", base);
        unsigned int hours_next  = FLAMEGPU-
>environment.getProperty<unsigned int>("macroproperty5_field_84", base + N);
        FLAMEGPU->setVariable<unsigned int>("daily_today_u32", hours_today);
        FLAMEGPU->setVariable<unsigned int>("daily_next_u32", hours_next);
    } else {
        // Для неактивных агентов налёт = 0
        FLAMEGPU->setVariable<unsigned int>("daily_today_u32", 0u);
        FLAMEGPU->setVariable<unsigned int>("daily_next_u32", 0u);
    }
    return flamegpu::ALIVE;
}
```

Здесь мы используем `environment.getProperty<Type>(name, index)` для доступа к элементу массива среды по индексу. PyFLAMEGPU позволяет так получать значения из объявленных Property Array (уже загруженных через loader) – аналогично тому, как мы получаем скалярные env-переменные, но с указанием смещения. В результате каждый агент читает **ровно один элемент** массива `daily_hours` для своего борта (и ещё один для next day). Все 7k агентов делают это параллельно, что на GPU быстро. Нет необходимости собирать массив целиком на хосте.

Пояснение корректности: Поскольку мы включили всех бортов в симуляцию с самого начала (все агенты-планеры созданы на D0), массив MP5 содержит для каждого из них запись на каждую дату (при отсутствии налёта – с нулём). Это гарантирует, что формула индекса правильна для всех `current_day` до конца доступного периода (4000 суток). Если в данных предусмотрены появления новых бортов (события рождения), тогда при отсутствии записи до появления

`hours_today` просто будет 0, что тоже корректно (агент скорее всего будет не статус2 до ввода в эксплуатацию).

- **(B) Обмен сообщениями между планерами и компонентами:** Другой вариант – использовать механику сообщений FLAME GPU, чтобы **распространить** информацию о суточном налёте. Идея: вместо каждому агенту искать свои часы, пусть **каждый планер** (группа 1 или 2) отдаст сообщение, содержащие налёт за день, а все компоненты ссылающиеся на этот планер получат его. Реализация:
- Завести тип сообщения, например `MessageBruteForce`, с полями `aircraft_number` и `daily_hours`.
- Создать агентную функцию `output_daily_hours`, исполняемую только для планеров (условие `if (group_by == 1 or 2) && status_id == 2`). В ней агент-планер получает свой `hours_today` аналогично шагу (A) – либо из глобального массива через формулу индекса (или более прямым способом, см. ниже) – и затем публикует сообщение:

```
FLAMEGPU->message_out.setVariable<unsigned int>("aircraft_number",
FLAMEGPU->getVariable<unsigned int>("aircraft_number"));
FLAMEGPU->message_out.setVariable<unsigned int>("hours", hours_today);
```

- Другой агентный RTC-функцией `input_daily_hours` подписать **всех агентов** на чтение этих сообщений. Каждый агент читает входящие сообщения и, найдя сообщение с совпадающим `aircraft_number`, берёт оттуда `hours` и записывает себе в `daily_today_u32`. (Если агент – планер, он просто найдёт своё собственное сообщение; если агент без `aircraft_number` (detached), он не найдёт ничего и оставит 0.)
- Значение `daily_next_u32` также можно передать аналогично, либо каждый планер отправит **два дня** – но удобнее обновлять `daily_next_u32` на следующем шаге как `daily_today_u32` (или отправлять "завтрашний" налёт тоже, увеличив сообщение на поле или второй проход).

Такой дизайн тоже избавляется от CPU: **планеры один раз находят свои часы через быстрый индексный доступ, а прочие агенты просто читают их из сообщений**, не сканируя глобальные массивы. Однако он более сложен из-за двух функций и системы сообщений. В большинстве случаев вариант (A) проще и не перегружает GPU (т.к. объемы не так велики).

Выбор стратегии: Рекомендуется вариант (A) – **прямое индексирование глобального массива** – как более прямолинейный. Данные MP5 уже структурированы для быстрого доступа, и у нас есть `frame_idx`. Линейный доступ к двум элементам массива на агента – это очень мало (порядка 14 тысяч чтений на шаг, GPU легко справляется). Вариант с сообщениями целесообразен, когда логика сложнее (например, поиск соседей), но здесь overhead не оправдан.

Итог: Добавляем новый слой в модель, например `layer_daily`, **перед основными слоями** `main` / `ops_check`, чтобы сперва обновить `daily_today_u32` для текущего дня. Логика последовательности будет такой: - `layer_repair` (как было), - **новый** `layer_daily_update` с функцией `update_flight_hours`, - `layer_quota_init` (см. следующий пункт), - `layer_main`, - `layer_ops_check`, - `layer_increment_day` (из шага 3).

Это гарантирует, что в `rtc_main` и `rtc_ops_check` агенты уже имеют корректные `daily_today_u32` (и `daily_next_u32` на случай принятия решений о будущем). Точность расчёта на GPU будет идентична прежней, так как используется тот же набор данных MP5, только без промежуточных копий.

Примечание о проверке: Можно добавить в режим отладки сравнение суммарного налёта. Например, агрегировать на GPU сумму всех `daily_today_u32` и сравнить с суммой по `mp5_maps` за день D на CPU (для одного шага) – ожидается полное совпадение. Но исходя из того, что MP5 загружен корректно, расхождений быть не должно.

5. GPU-инициализация квот на эксплуатацию (D+1) через MacroProperty

Текущее положение: Квоты на следующий день (MP4) сейчас передаются в среду **из хоста** перед каждым шагом. Видно, что перед `sim.step()` устанавливаются `quota_next_mi8` и `quota_next_mi17` [25], которые потом используются функцией `rtc_quota_init` на GPU для заполнения `remaining_ops_next_*`. Этот подход помечен как временный ("fallback до фикса NVRTC") [25].

Решение: Полностью перенести установку квот D+1 на устройство, используя **данные MP4, загруженные в среду**, и уже имеющуюся логику `remaining_ops_next_*` с atomicDec. План такой:

- **Отказ от хост-свойств** `quota_next_mi8/mi17`: Можно удалить переменные `quota_next_mi8` и `quota_next_mi17` из среды [26], либо просто перестать их использовать. Вместо них будем напрямую брать значения квот из массивов MacroProperty4.
- **Модификация функции** `rtc_quota_init`: В текущем виде она обнуляет `ops_ticket` каждому агенту и затем для агента `idx == 0` делает `q8.exchange(seed8)` и `q17.exchange(seed17)`, где seed8/17 берутся из скалярных env-свойств [19] [27]. Мы заменим получение `seed` на чтение из массивов MP4:

```
FLAMEGPU_AGENT_FUNCTION(rtc_quota_init, flamegpu::MessageNone,
flamegpu::MessageNone) {
    // Сбрасываем билет допуска всем агентам
    FLAMEGPU->setVariable<unsigned int>("ops_ticket", 0u);
    // Один агент выполняет инициализацию квоты
    if (FLAMEGPU->getVariable<unsigned int>("idx") == 0u) {
        unsigned int day = FLAMEGPU-
>environment.getMacroProperty<unsigned int>("current_day").get();
        // Берём квоту для ДНЯ + 1 (следующего дня) из MP4
        unsigned int idx = day + 1;
        unsigned int target8  = FLAMEGPU-
>environment.getProperty<unsigned int>("macroproperty4_field_74", idx);
        auto remaining8 = FLAMEGPU-
>environment.getMacroProperty<unsigned int>("remaining_ops_next_mi8");
        remaining8.exchange(target8);
        unsigned int target17 = FLAMEGPU-
>environment.getProperty<unsigned int>("macroproperty4_field_75", idx);
        auto remaining17 = FLAMEGPU-
>environment.getMacroProperty<unsigned int>("remaining_ops_next_mi17");
        remaining17.exchange(target17);
```

```
    }
    return flamegpu::ALIVE;
}
```

Здесь мы используем тот факт, что записи MP4 упорядочены по дням с индексом, совпадающим с номером дня от `version_date`. Если `current_day` – номер текущего дня D (начиная с D0 = 0), то `idx = day + 1` даст индекс записи следующего календарного дня (D+1) в массивах MP4. Например, при D=1 (второй день симуляции) возьмём запись с индексом 2 – это соответствует дате D2 (третьему дню от начала) [28], что эквивалентно прежнему поведению (при моделировании дня D агенты бронируют квоту на D+1).

В результате две макропеременные `remaining_ops_next_mi8/mi17` инициализируются полностью на GPU, без участия CPU. Поскольку `remaining_ops_next_*` – MacroProperty, декремент квоты (atomicDec `q--`) в `rtc_ops_check` продолжит работать так же, как раньше [20]. Это как раз реализует требование задачи «квота D+1 через MacroProperty и atomicDec» [29] – с той разницей, что теперь *seed* для MacroProperty приходит не с CPU, а из загруженных данных.

- **Убедиться в синхронизации с счётчиком дня:** Нужно правильно связать `current_day` и выбор квоты. Как обсуждалось, если мы увеличиваем `current_day` **после** шага (в `increment_day`), то на момент выполнения `rtc_quota_init` `current_day` ещё указывает на текущий день D. Тогда `idx = day + 1` и берётся квота на D+1 – именно то, что нужно. Например, в первый день симуляции (D1, если D0 уже пройден), `current_day = 1`, `idx = 2` → берётся квота на D2, как раньше на CPU [28]. При переходе к следующему шагу `current_day` станет 2 и т.д. Это синхронизовано.

Итого: мы устранили «узкое место» с ручной установкой квот. Теперь **каждый шаг** симуляции сам подтягивает нужные лимиты D+1 из среды. В исходном коде уже проверена логика, что квота применяется только для агентов с налётом (`dt>0`) и выдаётся атомарно [30] [31]. Эти части кода не изменяются. Мы лишь обеспечиваем их актуальными данными без CPU.

6. Сбор результатов и экспорт без промежуточного CPU вмешательства

После выполнения всех шагов (7 суток) данные MacroProperty2 (выходной лог) нужно сохранить в ClickHouse. Ранее это делалось путём чтения всей популяции агентов на каждый день и накопления строк на хосте [32] [33]. Теперь, когда симуляция идёт автономно, мы можем сократить обращение к CPU. Возможны два подхода:

- **Единичное считывание после последнего шага:** Поскольку вся информация по агентам на конец 7-го дня доступна в GPU, можно один раз вызвать `sim.getPopulationData` после цикла. Это даст состояния агентов на D7. Однако нам нужны метрики за каждый день D1–D7 (включая квоты и суточный налёт) для MP2. В принципе, можно сохранить только финальное состояние и вычислить за 7 дней, но это сложно, т.к. потеряется информация о промежуточных днях (например, `ops_current` ежедневно).

- **Логирование на GPU на каждом шаге:** Лучше всего предусмотреть механизм логирования результатов внутри модели. Например, добавить **Logging Layer** или использования вспомогательной MacroProperty2 в среде:

- Завести в `model.Environment()` набор массивов для выходных метрик MacroProperty2 (структура указана в документации [34] [35] – даты, борт, статус, налёт, квоты и пр.). Размер таких массивов будет 7 дней * кол-во планеров (в нашем примере 7 * ~279 ≈ 1953 записей).
- Добавить агентную функцию `log_daily_metrics`, выполняемую в конце каждого шага (например, перед `increment_day`). Она выполняется для **каждого планера** (group_by 1 или 2) и записывает в соответствующий элемент массивов среды значения:
  - Индекс записи = `current_day * Nframes + frame_idx` (похожий принцип блоков по дням).
  - Поля: `dates` (можно использовать `current_day` + `version_date` для вычисления или напрямую индекс, так как MacroProperty4 у нас есть), `status_id`, `daily_today_u32` (для planers это и есть налёт борта за день), `ops_counter_mi8/mi17` (целевые квоты на день – их можно взять из MP4 массива по индексу `current_day`), `ops_current_mi8/mi17` (фактическая укомплектованность – это число планеров в статусе 2 по группам; его можно посчитать в один проход: например, агент-планер при логировании делает atomicInc счетчика в среде по своей группе), `quota_claimed_mi8/mi17` (сколько билетов выдано – аналогично, агент с ops_ticket=1 делает atomicInc по своей группе). Остальные поля (возраст, triggers, metadata) при необходимости тоже можно рассчитать на GPU: возраст – на основе сохранённой `mfg_date` и `current_day` (годовые расчёты), метаданные – можно заполнить константами или считать на CPU после.
- Таким образом, за один GPU-проход по планерам мы запишем полную информацию за день D в массив среды MacroProperty2. Затем инкрементируем день и идём дальше.

После завершения 7 шагов массивы MacroProperty2 в среде будут содержать 7*Nframes готовых записей. Останется вызвать экспортёр (похожий на FlameMacroProperty2Exporter) на CPU, который одним батчем заберёт эти массивы и вставит в ClickHouse. CPU в этом случае участвует только один раз*, без цикла на каждый день. Размер данных невелик (<= 2000 записей на 7 дней), вставка пройдет быстро.

> **Упрощение:** Если реализация полноценного логирования на GPU кажется избыточной, допустимо и оставить сбор на CPU, но **с одним обращением после цикла**. То есть, выполнить `sim.getPopulationData` единожды на D7, а затем восстановить по сохранённым env-массивам некоторую информацию. Например, зная `daily_today_u32` за последний день и `ops_ticket` (который отражает квоту на *следующий* день, не нужный уже), можно вычислить итоговые метрики. Однако для точного соответствия MP2, где каждая строка – дневное состояние, лучше применять первый подход.

Контроль корректности: Сопоставляем с тем, что делалось на CPU: - `ops_current_mi8/mi17` – это количество планеров в статусе 2 каждого типа. Мы можем убедиться, что сумма инкрементов совпадёт с подсчётом на CPU [36]. - `quota_claimed_mi8/mi17` – число выданных билетов: на GPU считаем через счётчик, на CPU считали перебором `ops_ticket` [36]. - `daily_flight` – у планеров это значение `daily_today_u32` (для status_id==2) или 0 иначе [37] – мы как раз его и логируем. - `status_id`, `aircraft_age_years`, `mfg_date` – берём из агентских переменных (возраст можно посчитать и на CPU финально, т.к. D не велик).

7. Структура обновлённой модели

Объединим всё вышесказанное в итоговую последовательность слоёв GPU-модели **RepairOnlyModel**:

1. **Инициализация (host):** загрузка MacroProperty1,3,4,5 в Environment (см. шаг 1), создание агентов с назначением `frame_idx` (шаг 2), установка `frame_count` и начального `current_day` = 0.
2. **Шаг D0 (опционально):** выполнить `sim.step()` один раз, чтобы обновить, например, `repair_days` (как и раньше) [38]. В этом шаге `update_flight_hours` поставит всем `daily_today_u32=0` (так как `current_day=0` и налёты D0 можно считать нулевыми), квоты не потребуются. После шага D0 `increment_day` установит `current_day = 1`.
3. **Цикл D1–D7 (на GPU):** запускаем 7 шагов:
4. **Layer: rtc_repair** – без изменений (ремонт увеличивает счётчик или завершает ремонт) [38].
5. **Layer: update_flight_hours** – новый. Каждому агенту проставляет `daily_today_u32` и `daily_next_u32` из MP5 на основе `current_day` (шаг 4.A). *Пример:* в день D3 (current_day=3) для планера с `frame_idx=5` возьмётся запись с индексом = 3*Nborts + 5, что эквивалентно его налёту в таблице flight_program_fl за дату D3.
6. **Layer: rtc_quota_init** – модифицирован. Обнуляет `ops_ticket` и один агент загружает квоты D+1 из MP4 (шаг 5). Например, на день D3 возьмётся запись квоты D4 и поместится в `remaining_ops_next_*`.
7. **Layer: rtc_main** – без изменений (начисляет SNE/PPR текущего дня) [39]. Важно, что он идёт **после** установки `daily_today_u32`, поэтому использует актуальные `dt`.
8. **Layer: rtc_ops_check** – без изменений (проверяет лимиты `ll/oh` на следующий день и при `dt>0` пытается занять слот квоты через `remaining_ops_next_*`) [30] [31]. Макропеременные квоты уже инициализированы нашим `quota_init`, так что `q--` сработает корректно.
9. *(Optional)* **Layer: log_daily_metrics** – если реализуем, каждый планер пишет свою строку в MacroProperty2 (шаг 6). Можно объединить с `ops_check` при желании, но логичнее отдельным слоем.

10. **Layer: increment_day** – новый. Один агент увеличивает `current_day++` (шаг 3). После этого шаг завершается, GPU готовится к следующему дню.

11. **Окончание цикла:** После 7 итераций симуляция завершена на GPU. Все необходимые данные (состояния агентов, env-макропеременные и, возможно, массивы логирования) находятся в памяти устройства.

12. **Экспорт результатов (host):** Теперь можно собрать результаты:

13. Если настроили MacroProperty2 как массивы среды – воспользоваться аналогично MacroProperty4/5: вызвать экспортёр или вручную `sim.getEnvironmentPropertyArray` для каждого поля и вставить в ClickHouse. Это один батч INSERT ~ 2000 строк.
14. Либо выполнить `sim.getPopulationData` для агентов и сформировать строки MP2, проходясь по агентам в Python. При этом для значений квот и фактических показателей можно использовать либо сохранённые env-свойства, либо перерассчитать кратко. Например, суммарные `ops_current` и `quota_claimed` можно вычислить в Python за финальный день, но чтобы получить их для каждого дня – лучше всё же предварительно собрать на GPU, как описано.

Сопоставление с исходным пайплайном: Новый порядок исключает все “host-fallback” точки, обозначенные в описании:

- **Расчёт суточных часов (daily_today/next):** теперь происходит на GPU (функция `update_flight_hours`), используя загруженный MP5 вместо CPU `build_daily_arrays`.
- **Установка квот следующего дня:** вместо `sim.setEnvironmentProperty` на хосте используется код внутри `rtc_quota_init`, берущий данные из MP4 на GPU.
- **Сбор агрегатов:** опционально перенесён на GPU (через log layer), либо выполняется одной операцией после цикла (что значительно лучше, чем 7 раз).

Проверим **критерии готовности к снятию host-fallback**: 1. **Идентичность результатов.** Выходные значения (MP2) должны точно совпадать с прежней версией. Поскольку мы не меняли формулы, а только источник данных, все расчёты (status изменения, начисление sne/ppr, выдача билетов, смена статусов 4→5 или 2→6 и т.д.) происходят так же, как раньше [38] [40]. Отличаться могут только порядок операций (напр., мы ставим daily_hours чуть раньше, но это не влияет на логику) и отсутствие задержек между шагами. 2. **Производительность.** Устранив копирование данных 7 раз, мы заметно ускорим цикл. Собственно, `sim.step()` теперь выполняет больше работы на устройстве, но это пара сотен тысяч операций, что для GPU не критично. В Tasktracker указано, что уже были успешные 7-суточные прогоны RTC-функций [41] – наш вариант углубляет эту идею, убирая CPU из процесса. Можно замерить время шагов: ожидается снижение общего `step_ms` за 7 дней по сравнению с суммой CPU+GPU времени до рефакторинга. 3. **NVRTC/JIT compile вопросы.** Ранее была проблема с компиляцией RTC, из-за которой env-квоты временно задавали с CPU [25]. Наше решение добавляет ещё RTC-функции (update_flight_hours, increment_day), но они достаточно простые. При регистрации их в модели (например, `agent.newRTCFunction("update_flight_hours", src)`), нужно учесть порядок: `update_flight_hours` должен идти *после* `rtc_repair` и до `rtc_main`. NVRTC должен справиться с компиляцией – если же будут ошибки, возможно, придётся включить их поэтапно (есть механизм `FLAMEGPU_PROBE` в проекте для отладки RTC [42] [43]). Однако, поскольку логика независима, проблем быть не должно. 4. **Готовность данных.** Мы предполагаем, что **все нужные MacroProperty загружены**. Согласно документации, на 24-08-2025 всё это сделано и протестировано [44]. Значит, у нас есть гарантия, что, например, MacroProperty5 содержит полный диапазон дат подряд [45], и MacroProperty4 тоже покрывает все дни.

Таким образом, **после внедрения предложенных изменений** симуляция `sim_master.py` станет полностью GPU-ориентированной. Роль CPU сведётся к запуску инициализации и получению готового результата. Это упростит код (уберутся циклы с get/setPopulationData) и повысит скорость. Главное – строго проверить соответствие результатов: итоговый MP2 должен быть эквивалентен версии до рефакторинга (сравнить на небольшом прогоне). После этого «костыли» можно снять, и система будет готова к масштабным прогонам 4000 суток, используя только мощность GPU.

Ссылки на исходный код и документацию:

- Описание текущего цикла с CPU-вставками [1] [46].
- Объявление environment свойств в модели (до изменений) [47].
- Логика RTC-функций ремонта, квотирования и начисления налёта [30] [39].
- Документация по загрузке MacroProperty4/5 в окружение GPU [48] [6] [8].
- Tasktracker – подтверждение работы квот через MacroProperty и целей полного GPU-цикла [49] [29].

---

1 2 13 14 17 18 22 23 24 25 28 32 33 36 37 46 sim_master.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/sim_master.py

3 4 7 10 11 12 15 34 35 45 transform.md
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/docs/transform.md

5 6 48 flame_macroproperty4_loader.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/flame_macroproperty4_loader.py

8 9 flame_macroproperty5_loader.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/flame_macroproperty5_loader.py

16 19 20 21 26 27 30 31 38 39 40 47 repair_only_model.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/repair_only_model.py

29 41 44 49 Tasktracker.md
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/docs/Tasktracker.md

42 43 last_chat_export_20-08-2025.md
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/docs/last_chat_export_20-08-2025.md