ChatGPT

План полной GPU-реализации семидневной симуляции (Flame GPU)

Текущая реализация и проблемы

В текущем подходе симуляция выполняется за 7 суток с двумя «host fallback» этапами – подмешиванием данных с CPU на каждый день. А именно, **(1)** расчёт суточных часов налёта на CPU с последующей записью в переменные агентов, и **(2)** задание квот на следующий день через CPU (env property) перед каждым шагом [1]. Такая схема приводит к частым переключениям между CPU и GPU, снижая эффективность. Ниже кратко перечислены ключевые шаги текущего пайплайна (см. `sim_master.py`):

1. **Загрузка данных на CPU:** Запросы к ClickHouse для MP1, MP3, MP4, MP5 (например, `fetch_mp3`, `preload_mp4_by_day` и `preload_mp5_maps`) [2] [3]. Эти данные формируют исходные параметры агентов и глобальные плановые показатели.
2. **Инициализация агентов на CPU:** Формируется `AgentVector` и заполняются переменные каждого агента (например, psn, partseqno_i, status_id, sne, ppr и др.) на основе MP3. Также по маске типа ВС (`ac_type_mask`) выбирается порог BR (ресурс до ремонта) из MP1 и сохраняется как `br` [4] [5]. Затем популяция переносится на GPU через `sim.setPopulationData(av)` [6].
3. **Цикл по суткам (Host+GPU):** Для каждой даты D+1…D+7 выполняются:
4. *3.1 CPU:* Вычисление суточных часов налёта для каждого агента на день *D* и *D+1*. На основе MP5 формируются массивы `daily_today` и `daily_next` длиной N (число агентов) [7]. Эти массивы либо напрямую вшиваются в агентные переменные (`daily_today_u32`, `daily_next_u32`) посредством чтения/записи популяции [8], либо (в другом варианте кода) загружаются во внешнее окружение как Property Array `daily_today` / `daily_next` [9].
5. *3.2 CPU:* Задание квот на следующий день (*D+1*) в окружение модели. В текущей реализации хост устанавливает scalar property `quota_next_mi8` / `quota_next_mi17` перед шагом [10].
6. *3.3 GPU:* Выполнение одного шага симуляции (`sim.step()`), который включает последовательность RTC-функций агентов на устройстве [11]. Эти функции реализуют всю логику за сутки – обновление статусов ремонтов, начисление выработки (налёта), проверку порогов и квотирование для следующего дня. Например, функция `rtc_ops_check` проверяет ресурс агрегата с учётом завтрашнего налёта (`dn = daily_next_u32`) и при необходимости переводит агрегат в статус «списан/на ремонт» [12]; также она распределяет «билеты» (ops_ticket) на эксплуатацию в следующий день, атомарно уменьшая счётчики квот через MacroProperty [13].
7. *3.4 CPU:* Чтение результатов шага с GPU и агрегирование метрик за день. Чтение популяции (`sim.getPopulationData`) позволяет подсчитать, например, сколько агрегатов в работе (ops_current) и сколько получили билет (quota_claimed) для типов Ми-8/Ми-17 [14]. Также рассчитывается `daily_flight` (налёт) для агентов с `status_id==2` и возраст ВС в годах по `mfg_date` [15]. На этой основе формируются строки MP2 за день D [16] [17].

8. **Экспорт результатов:** После цикла 7 дней все собранные строки MP2 вставляются одним батчем в ClickHouse [18]. (Допускается выполнение этого шага на CPU.)

Проблема: Шаги 3.1 и 3.2 выполняются на CPU, тогда как их логика могла бы быть выполнена на GPU параллельно. Это создает узкое место: GPU простаивает, ожидая загрузки новых `daily_*` и квот из хоста на каждой итерации. Цель рефакторинга – **полностью перенести суточные обновления на GPU**, исключив обращения к хосту между шагами (кроме финального экспорта результатов). Ниже представлен план такой GPU-ориентированной архитектуры с использованием лучших практик FLAME GPU.

Лучшие практики FLAME GPU для данных и квот

1. Хранение исходных данных в GPU-памяти: FLAME GPU предоставляет механизм **Environment Property Arrays** для больших константных наборов данных, доступных агентам на GPU. В проекте уже реализована загрузка “MacroProperty” данных в окружение: например, MacroProperty4 (план полётов по дням) загружается как несколько Property Array в Environment [19], а MacroProperty5 (карта налёта ВС) – аналогично [20]. Эти массивы хранят всю необходимую информацию (список дат, суточные квоты, часы налёта по бортам и т.д.) в памяти GPU, избегая лишних обращений к CPU во время симуляции.

2. Использование MacroProperty для изменяемых глобальных счетчиков: Для счетчиков, которые должны изменяться на GPU (например, оставшиеся квоты), FLAME GPU 2 предлагает **MacroProperty** – глобальное свойство, поддерживающее атомарные операции внутри агентных функций. Текущий код уже применяет MacroProperty для квот: переменные `remaining_ops_next_mi8` и `remaining_ops_next_mi17` созданы как MacroProperty (размер 1) [9]. В RTC-функции `rtc_ops_check` агенты выполняют атомарный декремент этих счетчиков (`q--`) и получают *ops_ticket*, если квота ещё доступна [13]. Такой подход устраняет необходимость сериализации доступа к глобальным счетчикам и позволяет распределять квоты параллельно на GPU.

3. Сообщения между агентами для динамических данных: FLAME GPU поддерживает механизм **сообщений** для обмена данными между агентами. Это удобно, когда нужно каждый шаг доносить до агентов некоторую агрегированную информацию (например, налёт конкретного борта за день) без участия CPU. В нашем случае сообщения можно использовать, чтобы передавать суточные часы налёта от агентов-«носителей данных» (например, агентов, представляющих воздушные суда) к агентам-компонентам. Такой подход избавит от необходимости каждому компоненту самостоятельно искать свою величину в глобальном массиве, что дублирует вычисления.

4. Минимизация копирования популяции: Вместо того чтобы на каждом шаге выгружать и загружать назад всю популяцию агентов (как сейчас для установки `daily_today_u32`/`daily_next_u32` [8]), лучше обновлять агентные переменные прямо на GPU с помощью дополнительных RTC-функций или через сообщения. Это сохранит данные в виде GPU-структур на всём протяжении симуляции.

Опираясь на эти принципы, перейдём к конкретному плану рефакторинга.

Проектируемое решение: полный цикл на GPU

Ниже представлен поэтапный план перехода к полностью GPU-ориентированной симуляции. Каждый шаг сопровождается пояснениями и при необходимости – фрагментами кода, демонстрирующими реализацию.

Шаг 1. Единоразовая загрузка всех MP-данных в GPU-память
На этапе инициализации необходимо сразу загрузить на GPU все необходимые данные MacroProperty, чтобы впоследствии обращаться к ним без участия CPU. Это включает:

- **MP4 (суточные квоты):** уже загружены через MacroProperty4 Loader – в Environment содержатся массивы `macroproperty4_field_<id>` для `dates`, `ops_counter_mi8`, `ops_counter_mi17` и др. [19]. Убедимся, что эти данные охватывают весь период симуляции (как минимум D+1…D+7).
- **MP5 (карта налёта по бортам):** загружены через MacroProperty5 Loader – в Environment есть массивы `macroproperty5_field_<id>` для `dates`, `aircraft_number`, `daily_hours` и т.п. [20]. Они содержат записи “борт-дата-налёт” (возможно, для периода, покрывающего 7 дней симуляции).
- **MP1 (порог и времена ремонта):** можно хранить прямо в агентных переменных (как сейчас, поля `repair_time`, `partout_time`, `assembly_time` уже заданы каждому агенту при инициализации [4] [5]). Если эти поля понадобятся внутри GPU-логики, они уже на месте.
- **MP3 (исходное состояние агентов):** агенты уже созданы на GPU через `setPopulationData`. Каждый агент имеет необходимые поля – статус, накопленный ресурс (`sne`, `ppr`, `ll`, `oh`), идентификатор борта (`aircraft_number`), маску типа ВС и т.д. [21] [5].

Таким образом, все необходимые константные данные доступны на GPU в виде либо агентных переменных, либо Environment Property Arrays. **Дополнительно**, можно завести в Environment константу для начальной даты симуляции (например, `sim_start_date = D0` в днях от эпохи) и/или общее число дней симуляции (7) – это упростит индексирование.

Шаг 2. Добавление агентов-“носителей” для бортов (опционально)
Чтобы эффективно раздавать суточные часы налёта (MP5) без участия CPU, предлагаем ввести второй тип агентов – **агенты “Aircraft”** (по одному на каждый уникальный борт). Каждый такой агент хранит идентификатор борта (`aircraft_number`) и умеет по глобальным данным MP5 определить налёт своего борта за конкретный день. Эти агенты будут действовать как «источники» сообщений о налёте:

- **Определение типа агента:** В модели создаём новый агент, например, `agent_aircraft = model.newAgent("aircraft")`. У него минимум одна переменная `aircraft_number` (UInt), можно также хранить индекс или позицию в массивах MP5 при необходимости.
- **Инициализация агентов:** После создания всех агентов-компонентов из MP3, инициализируем агентов-борты. Из данных MP3 можно собрать список уникальных бортовых номеров (поле `aircraft_number` у компонентов) и создать соответствующее число `aircraft`-агентов. Каждому присваиваем `aircraft_number` своего борта. Эти агенты не обязательно имеют прямое соответствие 1:1 к компонентам, они моделируют сами ВС.

Если численность бортов невелика (что обычно так – например, десятки), накладные расходы от добавления такого агентного типа минимальны. Зато мы получаем возможность отправлять сообщения от каждого борта параллельно.

Шаг 3. RTC-функция для вычисления налёта на GPU (замена `daily_today` / `daily_next` на CPU)
Теперь нужно, чтобы каждый день симуляции актуальные `daily_today` и `daily_next` вычислялись на GPU. Реализуем это через дополнительные RTC-функции и, при использовании `aircraft`-агентов, через сообщения:

- **3.1. Отправка сообщения с налётом каждым бортом:** Создаём RTC-функцию, например, `flight_plan_out`, привязанную к агенту типа `aircraft`. Эта функция будет каждый шаг брать налёт своего борта на *сегодня* и *завтра* из MP5 и вещать их. Благодаря отсортированным массивам MacroProperty5 мы можем получить эти значения. Например:

```
FLAMEGPU_AGENT_FUNCTION(flight_plan_out, flamegpu::MessageNone,
flamegpu::MessageBruteForce) {
    unsigned int current_day = FLAMEGPU-
>environment.getProperty<unsigned int>("current_day_index");
    unsigned int aircraft_id = FLAMEGPU->getVariable<unsigned
int>("aircraft_number");
    // Поиск записи в MP5 для (aircraft_id, current_day)
    unsigned int today_hours = 0;
    unsigned int next_hours = 0;
    for (unsigned int i = 0; i < TOTAL_FLIGHT_RECORDS; ++i) {
        if (FLAMEGPU->environment.getProperty<unsigned
int>("macroproperty5_field_aircraft_number", i) == aircraft_id &&
            FLAMEGPU->environment.getProperty<unsigned
int>("macroproperty5_field_dates", i) == current_day) {
            today_hours = FLAMEGPU->environment.getProperty<unsigned
int>("macroproperty5_field_daily_hours", i);
            // запись для следующего дня того же борта:
            next_hours = FLAMEGPU->environment.getProperty<unsigned
int>("macroproperty5_field_daily_hours", i + N_AIRCRAFT);
            break;
        }
    }
    // Отправляем сообщение с найденными значениями
    auto out = FLAMEGPU->message_out;
    out.setVariable("aircraft_number", aircraft_id);
    out.setVariable("today_hours", today_hours);
    out.setVariable("next_hours", next_hours);
    return flamegpu::ALIVE;
}
```

Здесь `MessageBruteForce` тип сообщения без специального порядка – все `aircraft`-агенты просто рассылают свой налёт. В примере для простоты показан линейный поиск записи; в оптимизированном варианте можно воспользоваться тем, что данные отсортированы по дате, а записи каждого борта идут регулярно. Например, если в MP5 каждая дата присутствует для каждого борта, можно вычислить индекс: `i =`

`current_day_index * N_AIRCRAFT + offset_of_aircraft`. В реализации следует учесть формат хранения дат (UInt16 days since epoch). **Важно:** `TOTAL_FLIGHT_RECORDS` и `N_AIRCRAFT` могут быть вынесены как константы (либо Environment Property) при компиляции модели.

- **3.2. Получение сообщения компонентами и установка `daily_*`:** Создаём RTC-функцию `flight_plan_in` для агентов-компонентов (`component`). Эта функция принимает сообщения (тип должен совпадать с отправляющим, например, `MessageBruteForce`) и обновляет поля `daily_today_u32` и `daily_next_u32` у агентов в статусе эксплуатации. Пример логики:

```
FLAMEGPU_AGENT_FUNCTION(flight_plan_in, flamegpu::MessageBruteForce,
flamegpu::MessageNone) {
    unsigned int status = FLAMEGPU->getVariable<unsigned
int>("status_id");
    unsigned int tail = FLAMEGPU->getVariable<unsigned
int>("aircraft_number");
    unsigned int today_h = 0;
    unsigned int next_h = 0;
    if (status == 2) { // агент в эксплуатации
        for (auto msg : FLAMEGPU->message_in) {
            if (msg.getVariable<unsigned int>("aircraft_number") ==
tail) {
                today_h = msg.getVariable<unsigned int>("today_hours");
                next_h = msg.getVariable<unsigned int>("next_hours");
                break;
            }
        }
    }
    // Устанавливаем (неэксплуатируемые останутся с 0)
    FLAMEGPU->setVariable<unsigned int>("daily_today_u32", today_h);
    FLAMEGPU->setVariable<unsigned int>("daily_next_u32", next_h);
    return flamegpu::ALIVE;
}
```

Таким образом, **каждый компонент** сам получает нужное значение налёта своего борта напрямую от соответствующего сообщения. Если борта нет в сообщениях (например, борт не летает в этот день и не имеет записи), переменные остаются 0. Этот механизм полностью заменяет заполнение `daily_today_u32` / `daily_next_u32` на CPU и передачу через `setPopulationData` [8] – всё происходит внутри GPU.

Примечание: Если добавлять новый тип агента нежелательно, альтернативой является сделать функцию, где *каждый компонент сам ищет* в массивах MP5 свой налёт. Однако это дублировало бы поиск для всех агентов на одном борту. Использование сообщения от одного агента-борта устраняет дублирование и

считается более эффективным, особенно если много компонентов привязано к одному BC.

- **3.3. План слоёв для суточного обновления:** Включаем созданные функции в модель как отдельные слои перед основными вычислениями суток. Например, можно вставить:

```
lyr1 = model.newLayer()
lyr1.addAgentFunction(agent_aircraft.getFunction("flight_plan_out"))
lyr2 = model.newLayer()
lyr2.addAgentFunction(agent_component.getFunction("flight_plan_in"))
```

Эти слои должны выполняться **до начисления ресурса и проверки порогов**, чтобы поля `daily_today_u32` и `daily_next_u32` уже были установлены к моменту основных расчетов. В итоговом порядке слоёв будет: ремонт (`rtc_repair`), сброс квоты (`rtc_quota_init`), **обновление налёта (новые** `flight_plan_out/in`**)**, затем начисление ресурса (`rtc_main`) и проверка порогов/квот (`rtc_ops_check`). Такой порядок гарантирует, что на момент `rtc_main` каждый агент знает свой `dt` (налёт за текущий день), а на момент `rtc_ops_check` – `dn` (налёт за следующий день) для оценки остаточного ресурса.

Шаг 4. GPU-инициализация квот следующего дня без host

В текущем коде хост перед каждым шагом записывает в Environment scalar `quota_next_mi8/mi17`, после чего на GPU агент с `idx==0` считывает эти значения и присваивает MacroProperty-счётчикам через `exchange()` [22]. Чтобы убрать и эту CPU-операцию, сделаем следующее:

- **Получение квот D+1 на GPU:** Модифицируем RTC-функцию `rtc_quota_init` так, чтобы она сама извлекала нужные значения квот из массива MacroProperty4, вместо чтения хост-переменных. Агенту-инициатору (сейчас условие `if idx == 0`) нужно найти в массиве `macroproperty4_field_dates` индекс следующего дня и считать по этому индексу значения `ops_counter_mi8` и `ops_counter_mi17`. Поскольку массивы MP4 были отсортированы по дате при загрузке [23], можно осуществить поиск линейно (7 записей – затраты незначительны) или воспользоваться тем, что шаг симуляции известен. Например, если в Environment есть `current_day_index` (0 для D+1, 1 для D+2, ...), то индекс в массивах квот будет совпадать с `current_day_index`. Тогда код внутри `rtc_quota_init` будет примерно таким:

```
// Предположим, current_day_index = 0 для D+1, 1 для D+2, ...
unsigned int day_idx = FLAMEGPU->environment.getProperty<unsigned
int>("current_day_index");
// Считываем квоты по индексу day_idx
unsigned int q8 = FLAMEGPU->environment.getProperty<unsigned
int>("macroproperty4_field_ops_counter_mi8", day_idx);
unsigned int q17 = FLAMEGPU->environment.getProperty<unsigned
int>("macroproperty4_field_ops_counter_mi17", day_idx);
// Обмениваем значения в MacroProperty (atomic counters)
auto mq8 = FLAMEGPU->environment.getMacroProperty<unsigned
int>("remaining_ops_next_mi8");
mq8.exchange(q8);
auto mq17 = FLAMEGPU->environment.getMacroProperty<unsigned
```

```
int>("remaining_ops_next_mi17");
mq17.exchange(q17);
```

Этот код выполняется единым агентом (например, с `idx==0` или можно назначить это агенту типа `aircraft` с индексом 0 — не столь важно, важно лишь, чтобы выполнить ровно один раз). Таким образом, квоты загружаются напрямую из заранее загруженного MP4 **на GPU**, вместо передачи их с хоста через `quota_next_*`. Мы избавляемся от вызовов `sim.setEnvironmentPropertyUInt()` для квот [24].

- **Обновление счётчика дня:** Поскольку каждый шаг мы будем смещать индекс дня, нужно хранить и увеличивать `current_day_index` на GPU. Можно сделать это Environment-свойством и увеличивать через HostFunction, но мы стремимся избежать host-вычислений. Решение – использовать агент как счетчик: например, тот же агент `idx==0` в `rtc_quota_init` после установки квот может сделать `FLAMEGPU->environment.setProperty("current_day_index", day_idx+1)` (если среда допускает изменение). Однако прямого метода setProperty в RTC, скорее всего, нет. Альтернатива – хранить `current_day_index` как агентную переменную у специального агента (например, у одного агент-инициатора) и каждый раз инкрементировать её. Тем не менее, учитывая малость проблемы, допустимо установить `current_day_index` изначально в 0 и затем рассчитывать индекс как `(initial_index + step)` зная номер итерации. В FLAME GPU 2 номер текущей итерации внутри RTC-функции можно получить через `FLAMEGPU->getStepCounter()` (если доступно). Если NVRTC не предоставляет его, можно завести MacroProperty-счётчик шагов и инкрементировать его атомарно (но это избыточно). **Проще**: считать, что MacroProperty4 загружен ровно для нужных 7 дней в порядке, тогда *итерация 0 использует индекс 0, 1 -> 1, ...*. В таком случае в коде выше вместо `day_idx` можно подставить `FLAMEGPU->getStepCounter()` (начинается с 0). Это уберёт необходимость изменять что-либо в Environment, и будет работать корректно для фиксированного числа шагов.

- **Сброс билетов перед шагом:** Функция `rtc_quota_init` уже сбрасывает `ops_ticket` для всех агентов (`FLAMEGPU->setVariable("ops_ticket", 0)`) [25]. Это необходимо сохранить. При добавлении новых слоёв (налёт) убедимся, что `rtc_quota_init` идёт *после* ремонта, но *до* выдачи билетов и расчётов, что у нас соблюдается (она стоит перед `rtc_main` и `rtc_ops_check`).

Шаг 5. Запуск симуляции без CPU-вмешательств в цикле
После внесения изменений шаги симуляции будут выполняться полностью на GPU. Конфигурация слоёв (порядок действий за сутки) станет примерно такой:

1. **Repair Layer:** агенты в ремонте увеличивают счётчик дней ремонта, выходят из ремонта по достижении `repair_time` (тот же `rtc_repair`, без изменений) [26].
2. **Quota Init Layer:** один агент (или специальный) обновляет MacroProperty квот на основе данных MP4 для следующего дня, и обнуляет всем `ops_ticket` (наш доработанный `rtc_quota_init`) [22].
3. **Flight Plan Out Layer:** каждый агент-борта отправляет сообщение с налётом своего ВС за текущий и следующий день (`flight_plan_out` – новый).
4. **Flight Plan In Layer:** каждый компонент получает сообщение своего борта и устанавливает собственные `daily_today_u32` и `daily_next_u32` (`flight_plan_in` – новый).
5. **Main Layer:** агенты в эксплуатации (`status_id == 2`) начисляют себе наработку за текущий день `dt` (увеличивают `sne` / `ppr` на `daily_today_u32`) [27]. (Функция

`rtc_main` остаётся практически без изменений, кроме того что `daily_today_u32` теперь проставлено на GPU).

6. **Ops Check Layer:** все агенты в эксплуатации проверяют пороги ресурса с учётом завтрашнего налёта `dn` (`daily_next_u32`). Если ресурс после сегодняшнего дня недостаточен для ещё одного полёта, агент переводится в соответствующий статус (списание = 6 или потребность в ремонте = 4) [12] . Далее, каждый агент, кто сегодня летал (`dt > 0`), пытается получить слот на завтра: атомарно декрементирует `remaining_ops_next` счётчик своего типа и, если тот был >0, получает `ops_ticket` = 1 [13] . (Эта часть `rtc_ops_check` не меняется, кроме того, что мы убеждаемся – MacroProperty квот уже правильно инициализирован на GPU шагом 2).

В таком подходе **нет ни одного обращения к host** между итерациями шага. GPU сам «знает», сколько часов налёта дать каждому агрегату и сколько квот доступно, за счёт загруженных заранее массивов и описанных RTC-методов.

Шаг 6. Финальный экспорт и проверка результатов
После завершения 7 итераций можно собрать результаты для MP2. Этот шаг допускает использование CPU, поэтому возможны два варианта:

- **Вариант A:** Как и сейчас, считать метрики и формировать строки на CPU после каждого шага (или всех шагов сразу). Мы можем выполнять `sim.getPopulationData` после полного цикла и на основе итогового состояния агентов восстановить данные за каждый день, но это затруднительно (нужна история). Проще – продолжить сбор данных по дням, но *без вмешательства в ход симуляции*. Например, можно реализовать отдельный Logging-слой на GPU, который в конце каждого шага сохраняет нужные агрегаты в MacroProperty2 (таблица результатов). Однако это усложняет модель. Допустимо оставить агрегирование на CPU сразу после шага, **но не вмешиваться в данные агентов** – то есть, просто чтение состояния и запись в буфер. Такие действия не влияют на корректность симуляции и не требуют изменений агентских данных, поэтому удовлетворяют условию «без выхода на CPU, кроме экспорта». В идеале же – использовать GPU для логирования.

- **Вариант B (полностью GPU-логирование):** Создать глобальный буфер результатов в окружении (MacroProperty2) и по завершении каждой итерации с помощью специальной агентной функции записывать туда агрегированные показатели. Например, агент с idx=0 мог бы подсчитать через агентские переменные сколько сейчас `status_id==2` по группам и т.д., используя встроенные reductions, и сохранить в Environment Array. Однако, поскольку MP2 у нас включает запись по каждому агенту за каждый день, такой вывод проще делать на CPU. **Оптимальное решение:** использовать уже готовый экспорт после цикла, как сейчас: собрать `all_rows` и вставить батчем [18] . Главное, что на ход симуляции CPU уже не влияет.

После реализации GPU-версии следует провести **верификацию**. Критерий готовности – совпадение результатов MP2 (и финальных состояний агентов) с точностью до порядка строчек с прежней (CPU-включающей) версией. Необходимо протестировать, что при одинаковых исходных данных и начальном генераторе случайных чис (если используется) новая модель выдаёт те же `status_id`, `daily_flight`, `ops_current`, `quota_claimed` и т.д. за каждый день. Особое внимание – граничным ситуациям: распределение квот должно совпадать с прежней логикой (наш подход с MacroProperty и `q--` идентичен оригинальному [13] , так что распределение билетов сохранится). Также проверим, что **пороговые срабатывания** совпадают – т.е. агенты, выведенные из эксплуатации из-за ресурса, совпадают по дням с предыдущей реализацией.

Когда результаты совпадут, можно **удалить костыли**: убрать из `sim_master` вызовы `build_daily_arrays`, `setPopulationData` для налёта и `setEnvironmentProperty(quota_next*)` – они больше не нужны. По сути, `sim_master`/`sim_runner` сведётся к инициализации модели, запуску `sim.step(iterations=7)` и экспорту финальных данных.

Вывод

Предложенная архитектура переводит суточный цикл симуляции полностью на GPU, используя **FlameGPU 2** возможности: глобальные Property Arrays для справочных данных (MP4/MP5), MacroProperty для квотирования и межагентные сообщения для распространения динамических данных налёта. В результате, после начальной загрузки данных, 7-шаговая симуляция проходит без синхронизации с CPU – все расчёты статусов и ресурсов выполняются на устройстве, а CPU занимается лишь финальной сборкой результатов. Такой рефакторинг устранит узкие места текущей реализации и, как ожидается, ускорит симуляцию за счёт полного распараллеливания суточных обновлений на GPU. Importantly, the logical outcomes of the model remain the same – мы получаем эквивалентные результаты [12] [13], но с гораздо более эффективным исполнением. После тщательного тестирования можно окончательно убрать условные ветки и комментарии, связанные с “host fallback”, сделав код чище и готовым к дальнейшему усложнению модели (например, расширению периода симуляции или добавлению новых агентных типов) без потери производительности.

---

[1] [4] [5] [6] [7] [8] [10] [14] [15] [16] [17] [18] [21] [24] sim_master.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/sim_master.py

[2] [3] sim_runner.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/sim_runner.py

[9] [11] [12] [13] [22] [25] [26] [27] repair_only_model.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/repair_only_model.py

[19] [23] flame_macroproperty4_loader.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/flame_macroproperty4_loader.py

[20] flame_macroproperty5_loader.py
https://github.com/albud1978/Helicomponents/blob/b0fa9ef0f9c792b6559428a41d659a1f321b8b24/code/flame_macroproperty5_loader.py