Использование numba cuda jit

Москаленко Роман

1 Введение

Игра Новака и Мэя в классическом виде, при синхронной игре и подсчёте очков агентов, позволяет проводить вычисления для агентов отдельно, независимо друг от друга. Значит моделирование игры можно проводить на GPU для большей эффективности.

Были написаны две версии функции использующих numba.cuda.jit и соответственно работающих на GPU. Эти функции и будут разобраны ниже

2 GPU и cuda

Перед объяснением работы написанных функций, есть несколько основных понятий и принципов, на которых основывается cuda код.

2.1 Блоки и потоки

При каждом запуске cuda функции мы обозначаем количество блоков и потоков в блоках, на которых будут производиться вычисления. Количество блоков и потоков может быть как числом, так и двумерным или трёхмерным массивом, соответственно массивы будут задавать форму сетки блоков и потоков. Каждый поток в блоке будет выполнят весь код написанный в функции, уникальность действий каждого потока достигается за счёт возможности получить номер потока (как внутри блока, так и во всей сетке). Аналогично в случае, когда количество потоков и блоков задаётся массивом, можно узнать координаты потока в заданной сетке. Количество потоков в одном блоке ограничено ≤ 1024

2.2 Память

У GPU есть несколько доступных видов памяти, однако нас интересуют глобальная память(global memory) и разделяемая память (shared

memory).

Глобальная память позволяет нам хранить данные необходимые для вычислений на самом девайсе. Что значительно уменьшает время обращения к переменным, по сравнению с памятью хоста. Желательно перекинуть всё необходимое для вычислений в Глобальную память, это можно сделать вне cuda функции используя cuda.array.

Разделяемая память выделяется отдельно для каждого блока, все потоки в блоке имеют к ней доступ. Её очень мало (от 16 до 40 килобайт.). Время обращения к разделяемой памяти меньше чем к глобальной примерно в 100 раз. Поэтому если поток обращается к чему-то в глобальной памяти несколько раз, или если потоки в блоке обращаются к одним и тем же переменным в глобальной памяти, их стоит подгрузить в разделяемую память и дальше работать с ней.

3 Функция для простой кубической решётки

3.1 Описание алгоритма

Есть главная функция `evolve3D_5`, из которой потом вызываются cuda функции. Тут же происходит выделение глобальной памяти

Две вспомогательные cuda функции нужны для избежания лишних действий с памятью хоста. Первая обнуляет переданный массив, вторая копирует элементы из первого массива во второй. эти функции используются на глобальных массивах. Без них пришлось бы использовать `cuda_to_device` что сильно замедлит программу.

Основной код разбит на две функции `culc_scores3D` и `new_strategies3D`. Они написаны так, чтобы работать с трёхмерной сеткой блоков и потоков. Основная идея заключается в том, чтобы каждый блок подгружал, кубический кусочек поля в разделяемую память, затем мы работаем со всеми агентами, кроме крайних, так как для них в разделяемую память загружены все соседи. Блоки смещены так, чтобы соседние блоки пересекались по двум слоям агентов, в итоге крайние агенты каждого блока обрабатываются соседними блоками. Двумерная иллюстрация этого разбиения показана на рис. 1

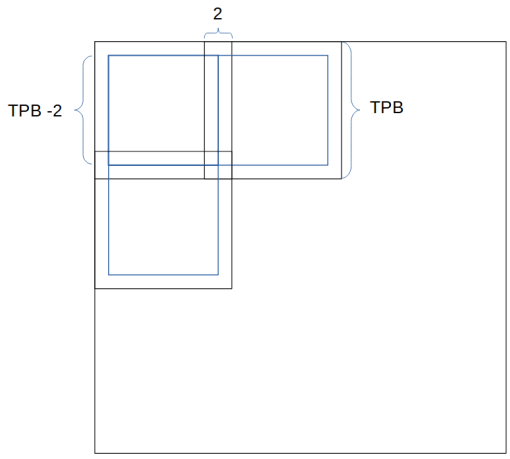


Рис. 1. Пример расположения блоков на двумерном поле. TPB - threads per block количество потоков в блоке.

Подгрузка происходит в начале каждой функции (рис. 2). Каждый поток загружает один, соответствующий элемент массива в shared memory. Затем следует команда cuda.sincthreads() которая означает, что потоки продолжат выполнение программы только, когда все потоки дойдут, до этой метки (то есть когда все потоки подгрузят свой кусочек массива). Далее мы останавливаем все потоки, отвечающие за края блока. Остальной код аналогичен коду в функциях без использования cuda.

```
# создаём массив
grid_sample = cuda.shared.array(shape=TPB, dtype=int32)
# Определяем координаты потока
tx = cuda.threadIdx.x
ty = cuda.threadIdx.y
tz = cuda.threadIdx.z

L = grid.shape

# определяем координаты агента, соответствующего потоку
x = (cuda.blockIdx.x * (cuda.blockDim.x - 2) + tx) % L[0]
y = (cuda.blockIdx.y * (cuda.blockDim.y - 2) + ty) % L[1]
z = (cuda.blockIdx.z * (cuda.blockDim.z - 2) + tz) % L[2]

# Каждый поток подгружает соответствующего агента
grid_sample[tx, ty, tz] = grid[x, y, z]

cuda.syncthreads()
```

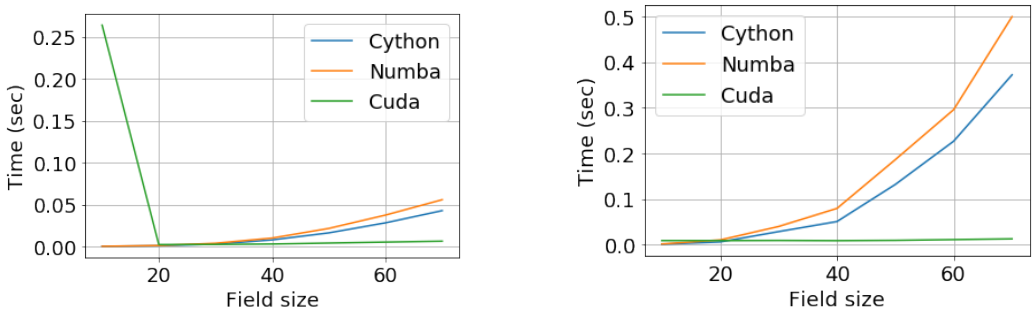
Рис. 2. Часть кода в которой происходит загрузка данных в разделяемую память

В `culc_scores3D` мы подгружаем часть массива grid, а в `new_stratagies3D`

часть массива scores. Так как для обработки одного агента, каждому потоку в функциях нужно обратиться к соответствующим массивам 27 раз. Остальные массивы подгружать не имеет смысла, так к ним потоки обращаются 1 раз.

3.2 Время работы

Данная функция показала наилучшее время. Она превосходит Cython и Numba.



(a) Врема расчёта одного шага эволюции

(b) Врема расчёта десяти шагов эволюции

Рис. 3. Графики времени работы различных реализаций функции

При увеличении количества шагов эволюции время работы меняется незначительно и не линейно. Это связанно с тем, при каждом вызове функции массив с полем переносится на девайс и обратно, и это занимает большое количество времени. Для частых замеров можно переписать функцию так, чтобы перенос массива происходил вне функции, что позволит загрузить поле на девайс один раз, и затем загружать его с девайса на хост, чтобы получить результаты. Это должно сократить время работы на маленьком количестве шагов примерно в 2 раза.

4 Функция с таблицей соседей

Без использования GPU наилучшие время работы достигалось функциями, использующими заранее подготовленные таблицы соседей: массивы в которых для каждого агента записаны номера всех его соседей. Эти таблицы позволяли избежать лишних арифметических операций и заменить их одним обращением к памяти. Так же использование таблиц

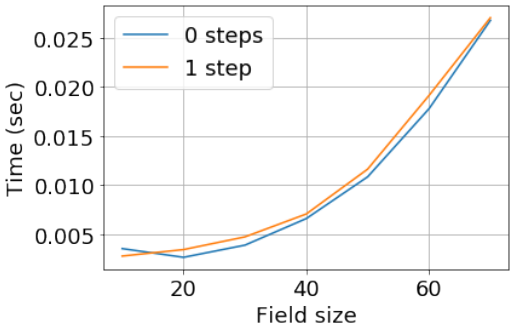
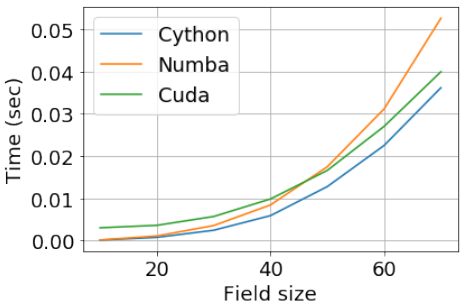


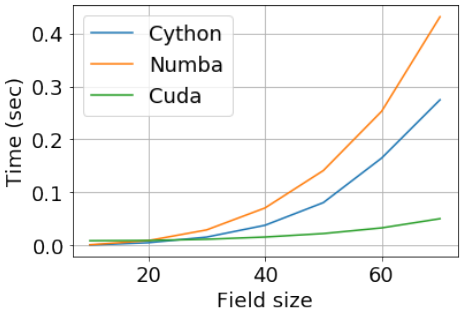
Рис. 5. Время работы `evolve3D_4` на одном шаге и 0 шагах эволюции. Во втором случае получается время загрузки массивов на девайс

делало функцию универсальной, позволяя использовать её на полях любой формы.

Вторая вариация функции evolve3D_ 4 с numba.cuda так же основывалась на таблицах соседей. Функция должна была быть универсальной и работать с полями другой формы или с другим расположением соседей. По структуре она идентична evolve3D_ 5 с одной главной функцией и четырьмя cuda функциями. Но в данном случае использование таблицы соседей не даёт выигрыша в скорости.



(a) Врема расчёта одного шага эволюции



(b) Врема расчёта десяти шагов эволюции

Рис. 4. Графики времени работы различных реализаций функции

Замедление функции связано с тем, что теперь помимо массива агентов, на девайс нужно загрузить таблицу соседей. Это занимает более 90% времени работы (рис. 5). Для конкретных замеров на малых временах это можно так же как и в предыдущем случае, производя загрузку вне функции.

Так же если рассмотреть время работы функции на большом количестве шагов эволюции (рис. 6), где загрузка массивов уже не составляет большую часть времени работы, видно что `evolve3D_4` Всё ещё медленнее чем `evolve3D_5`.

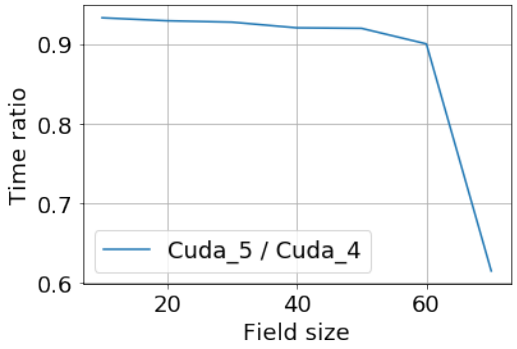


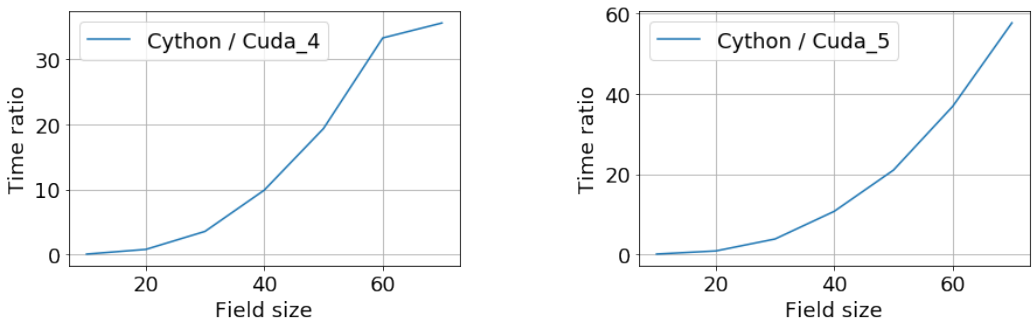
Рис. 6. Отношение времени работы функции для кубической решётки к времени работы функции с таблицей соседей на 1000 шагов

Это происходит из-за более частых обращений к глобальной памяти девайса. Помимо того, что в данном решении, чтобы узнать стратегию или счёт соседа нам необходимо 2 обращения, но и из-за того, что функция должна быть универсальной, нет простого способа использовать разделяемую память.

В `evolve3D_5` благодаря известной форме поля и расположению соседей мы можем использовать информацию, подгруженную потоками из одного блока. Однако для поля общего вида с таблицей соседей очень сложно разбить агентов на блоки так, чтоб соседи находились в одном блоке, и разделить подгрузку информации об агентах между потоками внутри блока. В итоге в то время как в функциях исполняемых на CPU, используя таблицу соседей, мы заменяли 6 арифметических операций одним обращением к памяти, на GPU мы ,избавляясь от тех же 6 операций, получаем 26 обращений к памяти.

5 Заключение

В готовом виде простая замена функции с cython на любую из разобранных функций, на большом количестве шагов уменьшит время работы примерно в 50 раз (рис. 7). Для замеров на малых временах необходима небольшая доработка кода вне функции, для предварительной загрузки информации о поле на девайс.



(a) `evolve3D_4` (b) `evolve3D_5`

Рис. 7. Отношение времени работы cuda функций к времени работы cython функции на 1000 шфгов эволюции