Cuda – это язык программирования на графических видеокарточках NVidia. Почему графических? Потому что для игр (компуктерных) требовались всегда вычислительные мощности. А потом люди сообразили, что их можно использовать не только под игры, но и под счёт в науке.

Почему NVidia? Так исторически сложилось. Есть попытка создать единый язык графических процессоров OpenCL, который не только под NVidia. Моя (т.е. Силаева) позиция такова – если не с ножом к горлу, то не лезьте. Та программа, которая на Куде занимает 10 строчек, на OpenCL 80, да ещё мы теряем в производительности. Если на Куде вызов функции с 10 параметрами – это одна строчка, то на OpenCL – это 11.

На графической памяти можно сделать несколько блоков (мы это число будем обозначать как NB). Каждый блок содержит несколько (обозначим NT) ниток. Всего таким образом NВ\*NT ниток. NB и NT вы задаёте вручную – какими их выбирать? Крайне желательно степенями двойки по понятным причинам, NT необходимо 32 или больше. На современном железе лучше ставить NT=128, а далее попробовать NT 64 и 256 – если выигрыша по производительности нет, то пусть 128 дальше и фурычит.

Откуда взялось ограничение на 32? Из-за warpов – в одном warpе строго 32 нитки. И один warp может быть только в одном блоке.

Терминология жуткая! Центральный процессор CPU называется хостом host, а графический, GPU – девайсом device.

Разберём типовую задачу: подсчёт интеграла от какой-то очень сложной функции методом трапеций. Тогда каждая нитка в GPU будет считать значение в своей точке, а затем мы все их сложим. Давайте разобьём отрезок на 8192 точек.

Выделение памяти под массив выполняется командой

cudaMalloc((void\*\*)&d,8192\*sizeof(double));

Конечно же, мы ленимся проверять, а выделилась ли память. Но если возникла какая-то это ошибка, это можно проверить – с помощью команды cudaSuccess().

Распространённая ошибка: ну нам же хочется написать что-то в духе

d[7]=13

Но массив-то был создан не в CPU, а в GPU. Поэтому число «13» запишется не в массив (который на графической карточке), а в место оперативки под номером 7. Если там были ваши данные – мы их потеряем, а если программа не имеет доступа к куску номер 7 – выведется ваша любимая ошибка Segmentation fault.

Итак, память под массив на GPU выделена, теперь надо его заполнить. Сперва нужно, чтобы каждая нитка узнала, где она. Сперва вычислим её глобальный индекс:

int idx = threadIdx.x+blockIdx.x\*NT;

Номер блока blockIdx.x и номер нитки внутри блока threadIdx.x нитка знает с рождения.

Затем нитка узнаёт, какую точку х ей нужно сосчитать:

И кладёт подсчитанное ею значение функции:

d[idx]=sin(х); - если нам нужно проинтегрировать синус.

Если нам нужно проинтегрировать сложную функцию f(), которая у нас где-то описана, то надо помнить, что каждая нитка должна получить её экземпляр (что приводит к резкому возрастанию времени компиляции (до 2 мин у меня доходило!) и размеру файла).

И затем массив нужно скопировать на центральный процессор, чтобы просуммировать. Можно его просуммировать его и на графическом, но про это мы скажем позже.

Копирование массива осуществляется функцией

cudaMemcpy(h,d,N\*8,cudaMemcpyDeviceToHost);

где вместо h и d вы указываете указатели на массивы на хосте и девайсе. Обратите внимание, что сперва указывается массив, В КОТОРЫЙ копируется.

Как вы понимаете, аналогично копируется массив с hostа на device.

Первая типовая ошибка: сделать так, чтобы функция

f<<<NB,NT>>> считала вообще всё.

Каждый поток будет долго считать вашу функцию f. С точки зрения Windows – раз поток долго не отвечает, значит, он завис. В таком случае он радует поток своими драйверами… как вы понимаете, после этого всё, что вы там подсчитали, будет похоронено в гробнице Ивана Грозного. Uniх не такая наглая ОС, она просто начинает орать «караул, поток завис, надо что-то делать».

Вторая типовая ошибка: не придерживаться концепции кинологов: каждая собака может жрать только из своей миски.

Давайте представим, что мы напишем в коде нечто в духе

d[7]+=1

Что будет?

Студенты: ясень пень, каждая нитка добавит по единичке, будет +NВ\*NT.

П.К.: Черта с два. Будет что угодно. Рассмотрим ситуацию:

нулевая нитка взяла d[7]

первая нитка взяла d[7]

нулевая нитка добавила единицу у себя

первая нитка добавила единицу у себя

нулевая нитка переписала d[7]

первая нитка переписала d[7]

В этом случае результат увеличится на 1.

А возможно так:

нулевая нитка взяла d[7]

нулевая нитка добавила единицу у себя

нулевая нитка переписала d[7]

первая нитка взяла d[7]

первая нитка добавила единицу у себя

первая нитка переписала d[7]

А в этом случае результат увеличится на два.

Третья типовая ошибка:

Ветвление потоков.

Пример:

if (ti%4==0) d[ti]=7;

if (ti%4==1) d[ti]=33;

if (ti%4==2) d[ti]=13;

if (ti%4==3) d[ti]=-5;

замедлит вашу программу ровно в 4 раза.

Сначала каждый поток проверит, не выполнено ли для него if (ti%4==0), 25% присвоят своей ячейке семёрку. 75% будут в это время спать. Потом снова 25% будут присваивать, а 75% будут спать.

Как этого избежать? Созданием служебного массива.

double ff=(7, 33, 13, -5};

d[ti]=ff[ti%4];

Нитки в одном warpе согласны делать одну работу, пусть и с разными данными.

Виды памяти на графической карточке:

Глобальная \_\_global\_\_. Её видят все блоки и все нитки.

Создать массив там можно двумя способами:

cudaMalloc(…);

или

\_\_device\_\_ double a[7], b[13][66][5];

Преимуществом первого способа является то, что такой массив мы можем впоследствии скопировать cudaMemcpy-ем на хост. Массивы же, выделенные как \_\_device\_\_ double a[7], обречены жить на девайсе навсегда. Но зато вторым способом можно создавать двумерные массивы. Если ну очень необходимо вывести двумерный массив назад, то доходит до смешного: создаётся cudaMallocом буферный массив, который затем и выводится.

Локальная. Семь казней египетских. Её быть не должно!

А как она может возникнуть? Предположим, напишите вы пожадничаете и напишите

double a[600];

Сколько памяти потребуется?

Студенты: 600 элементов по 8 байт => 4800 байт.

П.К.: А вот и нет. 8\*600\*NB\*NT!

У каждой нитки есть своя память – регистровая. По умолчанию все переменные создаются именно там. Т.е. если мы напишем

double a = idx\*idx;

то каждая нитка выделит в своём регистре 8 байт под переменную а и запишет туда idx\*idx.

Когда мы до этого писали

int idx = threadIdx.x+blockIdx.x\*NT;

каждая нитка снова выделила 4 байта в своём регистре под переменную idх.

Но регистры маленькие – порядка 32 Кб. Разумеется, 8\*600\*NB\*NT байт туда не влезут, и эта память пойдёт в общую память карточки – локальную, расположенную там же, где глобальная. Скорость тут уже упадёт катастрофически.

Постоянная.

Я ей ни разу не пользовался, но коллега, который программирует на CUDе значительно больше меня, её любит и утверждает, что за ней будущее.

\_\_constant\_\_ double b[6];

как вы уже поняли, двойные подчёркивания они очень любят

cudaMemcpyToSymbol(d,h,N\*8);

и я не знаю, почему именно Symbol!

Так вот, выполняется быстрее, чем обычная. Но менять её может только центральный процесс и только про инициализации, т.е. как раз при команде cudaMemcpyToSymbol.

Текстурная.

Лет 8 назад я бы сказал, что она полезная… а сейчас уже нет.

По сути, это артефакт тех времён, когда графические видеокарточки использовались для игр.

Shared-память. Наиболее адекватный перевод на русский – общая память для каждого блока.

Представляет собой среднее арифметическое глобальной памяти (для всей карточки) и регистров (для каждой нитки). По скорости так же – чуть медленней регистров и гораздо быстрее глобальной.

С её помощью реализуем операцию суммирования элементов массива двоичным деревом. Сначала просуммируем все элементы одного блока:

\_\_shared\_\_ double s[NT];

s[tr]=d[idx];

\_\_syncthreads();

Необычная команда синхронизации ниток. Если вы читали библиотеку С, то заметили, что с двух подчёркиваний обычно начинаются служебные функции для разработчиков, а не для юзеров. Эта функция, видимо, тоже сперва писалась для разработчиков, а потом оказалось, что она и юзерам нужна.

for (i=NT/2; i>=1; i/=2)

if (idx<i) { s[idx]+=s[idx+i];

\_\_syncthreads(); };

Далее в s[0] у нас лежит сумма. Её нужно из shared-памяти доставить в глобальную – например, пусть там лежит массив res[bi], куда мы будем класть суммы блока №bi. Как осуществляется перенос?

if (ti==0) res[bi]=s[0];

т.е. перенос осуществляет только первая нитка. Остальные пусть отдыхают – помним, что перенос в глобальную память и обратно занимает время, так что нужны делать его NT раз вместо одного нет.

Строго говоря, нам нужно ещё просуммировать суммы в каждом блоке! Но тут уже гораздо меньше значений, и, как правило, тут уже проще всё скопировать на центральный процесс и чтобы он подсчитал.

Поговорим про измерение времени.

Казалось бы,

c1=clock();

f<<<NB,NT>>>(…);

c2=clock();

И мы таким образом (казалось бы!) подсчитаем время выполнения функции f. Что происходит?

Всё это выполняется на центральном процессоре. Он дал команду графическому процессору «считай». А далее уже сразу с2. Досчиталась ли функция f на GPU – неизвестно.

Поэтому нужна команда синхронизации CPU с GPU:

Аж два раза:

c1=clock();

cudaDeviceSynchronize();

f<<<NB,NT>>>(…);

cudaDeviceSynchronize();

c2=clock();

Зачем нужна первая? Потому что к моменту создания с1 GPU может ещё что-то досчитывать.

Заметим, что cudaMemCpy, т.к. затрагивает и CPU, и GPU, автоматически синхронизирует их, так что обычно cudaDeviceSynchronize() не нужно.

Чудесный способ разочароваться во всех кудах, вместе взятых – наличие нескольких видеокарточек. Причём среди них может быть материнская, которая для счёта не предназначена.

Команда

cudaGetDevice(&nc);

выведет число видеокарт на вашем ПК.

Если 1, то всё ОК.

А вот если 2 и больше, то надо понять, какая под номером 0. Если материнская, то…

создаётся переменная

cudaDeviceProperties cp;

и выводится её данные

cudaGetDeviceProperties (&cp, n);

где n – номер карточки - 0, 1, …