

1 Введение.

О чём курс? Есть теоретическая база распараллеливания, и это было на курсе Елизарова. Но есть другой класс задач, в который параллельность тривиальная, и нам надо лишь быстро параллельно посчитать, без бед с теорией. Особенность в том, что для такого массового многопоточного исполнения CPU не очень подходит, потому что он создавался и оптимизировался для быстрого исполнения небольшого числа потоков. Типа 16 штук. Это очень мало. А если хочется параллельно обрабатывать десятки тысяч одинаковых данных, эти 16 потоков скорее будут выполнять ваши вычисления последовательно, а не параллельно. Поэтому тут нужны другие оптимизации и другой подход.

Что такое видеокарта? Это куча убогих процессоров, которых прям очень-очень много. И оптимизированы они под то, чтобы очень сильно параллелить. Каждый один из них медленный. Но если ваша задача бьётся на тысячи и десятки тысяч потоков, будет именно то, что хочется.

Мы будем рассматривать классический учебный пример: умножение матриц. Оно и полезное практически, и много полезных идей и техник содержит. И рассматривать мы будем его начиная с процессора, чтобы был какой-то бейзлайн. Его же мы будем использовать для проверки правильности работы, потому что в процессе оптимизации можно очень много набажить.

Как программировать видеокарточки? Изначально появились библиотеки в духе OpenGL, где были шейдеры: специальные программки в видеокартах, которые были нужны для трёхмерной отрисовки. Можно их и использовать: сгенерировать прямоугольник, загрузить входные данные в текстуры, выходные данные тоже выводить в текстуру, и потом из текстуры выгружаем. А потом стало понятно, что хочется не только рисовать, а именно что считать, и для этого хочется иметь более удобное API. Первым популярным из таковых стала CUDA. Получила она популярность не потому, что она была очень хороша, а потому, что NVIDIA влила туда кучу денег: куча учебных курсов, документации, бесплатно делали разным людям реализацию на CUDA. Зачем? Чтобы продавать лицензии. Права на исполнения CUDA-кода имеют только NVIDIA. Ну и как бы оно в целом норм, жизнеспособно. И до сих пор оно поддерживается, чтобы уже имеющийся код до сих пор работал на всех их видеокартах. Из ещё заслуживающего интереса упомянем OpenCL. У него тоже есть интересная история в том же духе. Таким же образом, как NVIDIA, поступает Apple. А ещё такие компании очень не любят, когда такое делают с ними, поэтому Apple и NVIDIA ненавидят друг друга. Поэтому Apple спонсировали создание открытого стандарта, являющегося конкурентом CUDA: OpenCL. И владеет им не Apple, а Khronos Group (те же люди, которые владеют OpenGL). И OpenCL в целом вообще более общий, чем CUDA. Он реализован не только под видеокарты, но и под процессоры и FPGA, например. FPGA — нечто среднее между железом и софтом: микросхемы, конфигурацию которых можно задать программно. Оно обычно программируется на специальных языках (например, verilog). И OpenCL позволяет программировать на FPGA на почти чистом C. Обратная сторона такой широкой поддержки: управление неповоротливым концерном. Пока NVIDIA щедро льют в CUDA новые фичи (из нового железа), в OpenCL всё происходит гораздо более медлительно. Из интересного NVIDIA не поддержала вторую версию OpenCL, и Khronos в третьей версии пошли им на встречу, сделав почти все фичи из версии 2 опциональными. Но нам, в целом, не важно, нам хватит OpenCL 1.2.

Ещё существует такая штука как HIP. Это убогая попытка AMD сделать совместимость с CUDA. Бинарная совместимость жёстко охраняется патентами, на программная совместимость запатентовать нельзя. Поэтому вы немного переименовываете функции CUDA, и получаете HIP. Меняете `cudaMalloc` на `hipMalloc`, и больше ничего. Получаете код под HIP. Но вот беда: поддержка у HIP'а в говне. Он работает только на некоторых карточках, притом необязательно на новых. И никакой логики в том, где оно поддерживается, нет. Совершенно не programmer-friendly.

Ну и не будем забывать, что уже миллион лет Intel пытаются выйти на рынок видеокарт. Они поддерживают OpenCL, но пытаются продвигать своё API, но это никем не используется, очевидно.

Ещё есть Metal от Apple, но яблочки сосут, это не будем трогать.

Ещё в новых версиях OpenGL (и в других графических библиотеках) добавили вычислительные шейдеры, и их можно использовать примерно так, как OpenCL (Vulkan Compute, например, вообще использует практически то же промежуточное представление, что и OpenCL). Но оно всё равно предназначено для графики (т.е. игрушек). Просто говоря — точность говно. Особенно точность деления. Ну и наконец Vulkan — низкоуровневое графическое API. Инициализация Vulkan — тысяча строк. Vulkan Compute — в десяток раз меньше, но это всё равно дохера много. А ещё шейдеры Vulkan Compute консервативны.

Из самого грустного — там нет указателей. Объекты передаются через сложные странных хэндлы. Передать указатель на матрицу в функцию нельзя. Удачи выразить свою мысль так, чтобы ничего не было скопировано. Это всё можно починить кучей расширений, которые для начала надо перечислить, включить и убедиться, что эти расширения у вас есть. Количество усилий для этого огромно. Ну и наконец, по сравнению с OpenCL, Vulkan нельзя запустить ни на чём, кроме видеокарт.

Кстати про OpenGL. Он говно. Его проектировали сто лет назад, и сейчас видеокарты работают совсем не так, как сейчас устроены видеокарты. Поэтому драйвер для него — огромный кусок дерьма. Vulkan появился не просто так, как бы. Игры написаны идиотами и если их запускать по стандарту, они сдохнут. Поэтому, например, NVIDIA пишет драйвера с кучей костылей по тому, как заставить игрушки работать, да ещё и быстро. Поэтому, например, NVIDIA имела более слабое железо, но FPS'ов было больше. И тогда AMD выпустили низкоуровневое API, которое соответствовало бы устройству видеокарт. Оно называлось Mantle. Это чудо тоже отдали Khronos'у, который сделал из этого Vulkan. Собственно, по той же причине, что OpenGL не соответствует устройству карт, Intel сильно получили в рожу, когда выпустили свои первые карты (они состояли из кучи первых Pentium'ов): они не умели в эту чёрную магию, поэтому им пришлось использовать DXVK — транслятор DirectX в Vulkan (который изначально был создан для запуска игр под Linux).

План курса такой: сначала потрогаем CUDA (владельцы красных карт возьмут народный конвертер из CUDA в HIP), а потом OpenCL.

Как это всё использовать? CUDA — расширение C++, которое добавляет магические заклинания. Его надо скомпилировать так: файл .cu подаётся компилятору CUDA, который разделяет его на device-код и host-код. host — это просто C++, который кормится обычному компилятору. И device-код компилируется специальным компилятором от NVIDIA, и получается специальный бинарник, который вставляется прямо внутрь вашего бинарника. Всё это линкуется с библиотекой, которая занимается как раз тем, что вычисляет device-бинарник и отправляет его на видеокарту, попутно разбираясь со временами жизни и т.п. Эти действия можно и руками делать, если очень хочется.

OpenCL делает примерно то же самое, что вы получите от CUDA, если будете делать все шаги руками. Вам придётся самим указывать, где брать бинарник, с чем линковаться и т.п.