Содержание

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 проспонсировали 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 и пытаются продвигать своё API, но это никем не используется, очевидно.
Ещё есть Metal от Apple, но яблочники 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 дайвер 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 вычленяет 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.

2 CUDA.

Установка. Просто потому, что визуально оно простое. Потому, что NVIDIA все страшные вещи замела под ковёр. А точнее, в библиотеку от которой у нас будет зависимость. Что надо для CUDA? Нужны заголовки и либы, то есть SKD. Оно называется Cuda toolkit.
После этого у нас будет вариация на тему C++. Мы уже помним, что мы пишем файлики с расширением *.cu* на C++ с расширениями, а компилятор пилит это на C++ и device-код для CUDA. Первый компилируется обычным компилятором, а второй специфичным компилятором.
Чтобы вызвать функцию CUDA надо сделать `#include <cuda.h>`.
Первое, что надо сделать:

```
int dev_num;
cudaError_t err = cudaGetDeviceCount(&dev_num);
```

Если всё хорошо, вам вернут ноль. Иначе — код ошибки. Какие там коды, можно посмотреть в <cuda.h>. Из интересного, clang-18 умеет компилировать Cuda. Версия 19 имеет с этим проблемы (как минимум под виндой). Также компилироваться можно из сред разработки, например, Visual Studio. Чтобы компилить из командной строки, делаем следующее (и вправляем пути, если надо):

```
clang++ kernel.cu -o kernel.exe -O3 -lcudart_static --cuda-gpu-arch=sm_61 \
    -Wno-unknown-cuda-version -fms-runtime-lib=dll -Wl,-nodefaultlib:libcmt \
    --cuda-path="C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6" \
    -L"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6\lib\x64"
```

Если вы получите нулевой код возврата и один девайс, это кайф. Если вам вернули ноль или вернули ошибку, которая показывает, что нет девайсов, то грустно. Что делать в таком случае? Если видеокарточка зелёная, значит криво на неё встали дрова, и их надо переставить. Если она красная и не очень древняя (архитектура RDNA и новее) и AMD о ней не забыли, можно поставить ROCm — народный транслятор CUDA в HIP, плюс подходящий рантайм. Вам надо скачать ZLUDA (https://github.com/lshqqytiger/ZLUDA). Для Windows из него надо взять nvcuda.dll и положить в WINDOWS\System32. Если карточка новая, всё хорошо, ставьте себе версию 6, и всё должно работать. Если у вас что-то старее RDNA, можно попытаться взять версию 5 и надеяться, что встанет, но надежды мало. Под Linux всё сложнее но и больше шансов на успех. Если карточка новая, оно может встать из коробки и так. Иначе надо гуглить, наверняка кто-то умный вправил так, чтобы всё работало. Если у вас синяя видеокарта, говорят, что если она новая, старая версия ZLUDA умеет транслировать в Level zero runtime, если у вас он, можете попробовать, но никто ничего не гарантирует. Если ничего из вышеперечисленного не помогает или у Вас мак, есть только сервер.

Устройства. Что дальше? Дальше для каждого девайса вызываем `cudaGetDeviceProperties`. Оно вернём кучу всего. По-хорошему ваша программа должна давать возможность выбрать девайс пользователю, если их несколько. Работать на всех девайсах сразу сложно (с точки зрения написания кода). Особенно это актуально для ZLUDA, потому что если у вас есть несколько девайсов AMD, ROCm сначала перечисляет интеграшку, а потом нормальные карты (пока все остальные делают наоборот). А ещё хорошо бы иметь приличные значения по умолчанию. Как понять, что за карты у вас? Собственно `cudaGetDeviceProperties`. Из интересного пользователю есть поле `.name`, из интересного нам — `.major` и `.minor`. Это compute capability. Это про вычислительный уровень вашего железа. Разные поколения карточек имеют разные возможности. В каких-то картах добавляют новые фичи, в каких-то даже убирают. Как правило больше — лучше, но необязательно. Для старших версий это правда, а для младших версий — вообще необязательно, потому что `.0` обычно — карты для data center'ов. Девайсы с одинаковым CC как правило отличаются тем, сколько раз вычислительные блоки скопипастили. Чем больше — тем быстрее. А вообще про CC можно открыть статью Википедии, где есть огромная таблица о том, что карточка умеет.
После этого нужно сделать `cudaSetDevice`, где вы говорите, какое устройство хочется. Он устанавливает устройство для текущего **потока**. В процессе этого всего нужно проверять возвращаемые значения.

Память. Чтобы что-то вычислить нам надо сначала иметь данные. А где у нас данные? Правильно, в оперативке. А мы хотим скопировать данные на девайс, и дальше уже запускать вычисления на этих скопированных данных. В этом основная проблема вычислений на видеокарте: время передачи данных на устройство может превышать время вычисления на процессоре. Исключением из этого являются (точнее, могут быть) интеграшки, потому что они способны работать без копирования (если это правильно накодить).
Чтобы выделить память надо вызвать `cudaMalloc`. Как скопировать данные с хоста на девайс? Во-первых, есть `cudaMemcpy`, которая помимо стандартных аргументов (куда, откуда, сколько) принимает направление. Направление — это либо «с хоста на девайс» (`cudaMemcpyHostToDevice`), либо «с девайса на хост» (`cudaMemcpyDeviceToHost`), либо «с девайса на девайс» (`cudaMemcpyDeviceToDevice`). Но ещё там есть `cudaMallocAsync`. Эта штука не останавливает процессорный тред пока не скопируется. Она ставит задачу копирования в очередь, и продолжает исполнение хост-кода. Очередей, вообще говоря, несколько, и по умолчанию вам ставят в дефолтную очередь текущего потока. Нам ничего кроме дефолтной не понадобятся, но вообще можно создавать другие очереди и указывать их.

Вычисление. Пусть мы хотим сделать $a + b$. Как сказать, что мы хотим нашу штуку на девайсе? Тут используется магическое слово `__global__`:

```
__global__ void add(const int* a, const int* b, int* c)
{
    *c = *a + *b;
}
```

Это ключевое слово значит, что это точка входа на девайсе, которую можно вызвать с хоста. Если использовать магическое слово `__device__`, то это функция, которую можно вызывать только из девайса.
Как это запустить? Вот так: `add<<<1, 1>>>(a, b, c)`. Оно не возвращает код ошибки, если вы хотите узнать, удачно ли оно поставилось в очередь, можете сделать `cudaGetLastError`. Тут единички — то, сколько вычислителей на видеокарте мы хотим. Об этом ещё потом поговорим. После этого лучше делать синхронное копирование с девайса на хост. Почему синхронное? Потому, что нам нужно подождать, что всё предыдущее посчиталось, и можно, конечно, явно ждать, но зачем, если можно подождать, пока синхронное копирование кончится. Ведь начнётся оно только тогда, когда всю очередь до него разгребут.
После этого нужно сделать `cudaFree`, чтобы освободить выделенную память.
А ещё в самом конце рекомендуют писать `cudaDeviceReset`. Во-первых, после этой команды если вы под профилировщиком, он начнёт выводить результаты. Во-вторых, есть ситуация, что если не вызвать эту функцию, видеокарта может не перейти в режим низкого энергопотребления (а остаться в режиме «мы активно вычисляем»).
Как теперь превратить это из суммы двух чисел в сумму двух массивов? Для начала надо изучить понятие SIMT — single instruction multiple threads. Одно и то же будет выполняться кучей тредов. Как эти треды создать? Для начала надо понять числа в тройных угловых скобках. Первое — сетка, второе — блок. Блок — тесно связанная группа потоков. Размер блока ограничен вашей видеокартой, и его можно получить из `cudaGetDeviceProperties` и он следует из compute capability. А первый параметр — сколько блоков запускать. Вообще индексация может быть и двухмерной, и трёхмерной, но это нам пока не надо. Пригодится это для матриц.
Так как, собственно, складывать массивы? А вот так. В нашей `__global__`-функции можно получить `threadIdx.x`. Это номер треда внутри блока. Ещё есть `blockDim` и `gridDim`. Они тоже трёхмерные, это количество тредов внутри блока и количество блоков в сетке. И ещё есть, конечно же, `blockIdx` — индекс блока внутри общей сетки блоков. Из интересного ещё есть `warpSize`, но про варпы мо поговорим потом. На NVIDIA это уже очень давно (и не собирается меняться) 32. Итого пишем следующее:

```
__global__ void add(const int* a, const int* b, int* c)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    c[i] = a[i] + b[i];
}
```

Как работает видеокарта. В видеокарте тысячи, сотни тысяч «ядер». На самом же деле это не ядра, а скорее конвейеры. А если ещё точнее, это контекст исполнения. Само исполнения и сами конвейеры в другом месте. И блоки по несколько (типа 32) объединены в исполнительный блок. Конвейер умеет считать. И всё. Он не занимается чтением команды, обработкой переходов и т.д. И настоящий исполнитель — блок конвейеров, который читает команду и исполняет её на всех конвейерах. NVIDIA называет эту конструкцию (и размер такого блока) warp. AMD это называет wavefront. В итоге команда одинаковая, но разные регистры и данные. Можно считать, что у каждого конвейера свои регистры (это не совсем правда, но близко).
Что делать с ветвлением? Проверить, какие треды куда идут. Если все в одно место, то кайф, идём куда надо, вычисляем. Если же хотя бы один поток идет не туда, куда остальные, то сначала все вычисляют ветку `if`, а потом все ветку `else`. Точнее, те, кому нужно, считают, а остальные делают ничего. Итого на процессоре стоимость `if` определялась его предсказуемостью, а тут она определяется тем, все ли в warp'е идут в одну и ту же ветку. Никакого предсказателя нет.

Ещё одна особенность: несколько warp'ов могут быть запущены на одном большом исполнителе. Это имеет некоторые бонусы. Первая — они могут очень легко синхронизироваться (т.е. все подождать пока все не придут к одному месту). Вторая — они имеют свой тип памяти (о ней позже). И это блок. Отсюда мораль: размер бока должен быть кратен размеру warp'а. А если некратен, то лучше чтобы был поменьше, чем побольше (лучше чуть-чуть необрать warp, чем взять новый почти пустой).
На NVIDIA размер warp'а 32 с самого начала и до сих пор без изменений. На AMD зависит от архитектуры. На GCN (он же CDNA) — 64. На RDNA — переключаемое, 32 либо 64. 32 нужно для совместимости с NVIDIA, если вам оно не нужно, скорее всего вам нужно использовать 64.
Но тут ещё вот какое дело. Говорили про блоки. Блоки — это то, что на самом деле исполняется. Есть большие исполнители (называются SMX на NVIDIA). И работают они так. Вы указываете размер блока, и происходит следующее. Он нарезается на варпы, и эти варпы исполняются не одновременно. У исполнителя есть очередь, и эти варпы складируются в очередь. И дальше исполняется один-два варпа за раз (сколько конвейеров у исполнителя есть). Это даёт нам вот какой прикол. Наши варпы иногда хотят обращаться в память. А это долго. И вместо того, чтобы ждать, мы суём этот варп в конец очереди и начинаем исполнять другой. А ещё есть ограничение на количество варпов в блоке. Например, некоторые видеокарты могут размер блока не больше 1024 (т.е. не больше 32 варпов). Как узнать, сколько лимит — выдаёт `cudaGetDeviceProperties` (там есть `maxThreadsPerBlock`). А вообще есть тайное знание, что если ваша карта умеет в DirectX, то 1024 треда точно можно. Из интересного, этот самый исполнитель имеет аппаратные регистры (сколько — смотрите в `regsPerBlock` от `cudaGetDeviceProperties`). И он режет их на треды. Так что если вам надо много регистров, вы проиграли. Вы либо лимита на 1024 треда не достигнете, либо на каждый тред отведут мало регистров, и локальные переменные будет сбрасываться в память, что долго. (Даже если тред не исполняется, его регистры — это его регистры.) Так что вам возможно будет лучше с блоком размера поменьше. Но это сломает вам occupancy (т.е. идею, что блоки, которые ждут память, помещаются в конец очереди). И ещё прикол: если у вам блоки маленькие и потоки жрут немного ресурсов, можно запустить несколько блоков на одном исполнителе. При желании на разных kernel'ах (т.е. функциях, которые мы запускаем). В том числе разные типы шейдеров. На одном исполнителе можно смешать блок из рисовки и блок из вычисления.
Остался один параметр, который мы не обсудили: сетка. Тут всё просто, это количество запускаемых блоков. Они встают в программную очередь на уровне драйвера, который подкидывает блоки на исполнение исполнителей. Если карточка мощная, а сетка маленькая, вам могут запустить все блоки. Но более частая ситуация иная.
Дальше мем. И блок, и сетка — трёхмерные структуры. Всё, что мы говорили сейчас — общее количество тредов в линии/квадрате/кубе. Можно считать, что у нас блок двумерный 32×32. И работать это будет как просто 1024. Просто для удобства нашей индексации. Синтаксически это записывается так:

```
dim3 grid(4, 4, 4);
dim3 block(32, 32); // z = 1
kernel<<<grid, block>>>(arg);
```

Пример. Так, как умножать матрицы? Умножение матриц — три `for`'а. И идея такая — `for`'ы по индексам выходной матрицы мы делаем потоками. И в каждом потоке делаем один `for`. Давайте договоримся, как мы будем писать. Мы будем считать, что $A \in \mathbb{R}^{M \times K}, B \in \mathbb{R}^{K \times N}$. Так вот наши потоки будут делать цикл до K.
Как измерять время? Можно попытаться мерить время на хосте, но это не ок (так мы измерим вместе с копированием данных). Лучше узнавать время на девайсе. Для этого нужно `cudaEvent_t`. Делаем так:

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
// Run kernel.
cudaEventRecord(stop);
```

```
// Synchronize (can be synchronous memcpy).
cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(&stop);
cudaEventDestroy(&start);
```

Производительность. В чём считают производительность? В FLOPS'ах (floating-point operations per second). Вот только обычно вам нужны не просто флопсы, а полезные флопсы. Их посчитать легко — посчитать количество операций руками и поделить на время. В нашем случае это $2mnk$, потому что умножение и сложение — две операции (даже если `fusedMultiplyAdd`). Так вот, если посчитать с нашим тупым умножением матриц, получится очень мало, по сравнению с тем, что на Википедии написано про вашу карту. Почему?
Помимо флопсов можно посчитать скорость передачи данных в секунду. то считать как объём делить на время. И вот по ней мы скорее всего упрёмся в максимум. Однако может быть кэш, который имеет пропускную способность больше. И если в кэш ненулевое попадание, скорость мы видим повыше. Но она в любом случае не меньше максимума у нас будет. Это значит, что мы упёрлись в память, и наша карточка ждёт. Как это починить?
Тут есть понятие разных типов памяти. Раньше мы использовали глобальную память (через что мы передаём данные с хоста, память видеокарточки) и регистры (локальные переменные внутри kernel'а по умолчанию создаются там). Если пытаться создать массив, его тоже попытаются засунуть в регистры. Если не хватит, их сбросят в глобальную память, что больно стукнет по скорости. Так что локальных переменных должно быть не очень много.
В целом всё как на процессоре, только регистров больше. Например, на современных картах есть шанс одному kernel'у получить 255 4-байтовых регистров (`double` занимает регистровую пару). В целом на видеокартах топ уровня и процессорах скорость `double` обычно в 2–3 раза меньше скорости `float`'ов. В менее дорогих картах обычно сильно медленнее. Типа 64 раза. Иногда даже программно н уровне драйвера.
Так окей, что делать? А есть ещё несколько видов памяти. Во-первых, видеокарту можно заставить обращаться к оперативке. Но этого вы не хотите: если ваша карта не интеграшка, это будет очень долго (PCI-express сильно дольше глобальной памяти). Есть это потому, что вы стремитесь к великой мечте: unified address space — правила работы с памятью общие для всех, чтобы можно было перебрасывать указатели между разными устройствами, и это будет одними и теми же указателями. Мы медленно идём в ту сторону, но пока нет. Ещё есть константная память. Синтаксически это глобальная память, но аппаратно может быть чем-то специфическим. Она доступна только для чтения, и это может быть какой-то специальный блок, который загружается перед стартом программы, либо это может быть глобальная память, доступ к которой может быть каким-то другим. Например, в NVIDIA там отдельный кэш для константной памяти. Размер его на NVIDIA небольшой, на AMD сколько глобальной памяти, столько константной, но на AMD никак от глобальной не отличается. На нашей матрице, короче, это не поможет (на NVIDIA размера не хватит, на AMD полезно не будет). Ну самая интересная — shared (разделяемая) память. Это отдельный вид памяти, который есть на исполнителе. И весь блок тредов имеет к ней доступ. И у разных блоков эта память разная (каждый блок получает свой собственных экземпляр этой памяти). По сути это ручной кэш. Вы явно говорите читать и писать именно из этой памяти, но используют её как кэш. Из забавного, на NVIDIA shared-память и кэш — это одна и та же память физически. Некоторые карточки даже позволяют распределить, какой процент этой память мы отдадим под кэш, а какой на разделяемую память. Если карточка поддерживает DirectX, у неё минимум 32КБ (на каждый блок).
Собственно, как применить это в нашем случае? Давайте в один блок засунем квадратик $x \times x$ нашей матрицы. Тогда надо будет обращаться к x строкам одной матрицы, и x столбцам другой матрицы. Перенести и то, и другое в разделяемую память, мы не можем. Во-первых, её может не хватить. Но даже если её хватит, возможно вам не нужно заполнять её полностью. И вот почему: мы помним, что один исполнитель может запустить несколько варпов в себе, если ресурсов хватает. И разделяемая память — такой же ресурс. Если ваши потоки жрут мало регистров, но варп сожрал целиком всю разделяемую память, то запустить пару варпов сразу не получится :(Чем меньше используете разделяемой памяти, тем лучше occupancy.
Так, окей, что делаем? Делаем «тайловое умножение матриц». Пилим матрицы на «тайлы» (квадра-

тики), и переносим в разделяемую памяти один квадратик из одной матрицы, и один квадратик из другой. И вычислять мы будем примерно так:

```
for (k; k += TILE) {
    a[...][...] = A[...];
    b[...][...] = B[...];
    for (k2 = 0; k2 < TILE; k2++) {
        res = a[...] * b[...]
    }
}
```

И тут `a` и `b` находятся в разделяемой памяти. Для начального понимания будем считать, что `TILE` — это размер блока.
Концептуально, поскольку все потоки параллельно исполняются, до внутреннего `for`'а у нас все скопируют своё значение в shared-память. И после этого будет нормальное обращение в shared-память. Завести себе shared-память так: `__shared__ float a[TILE][TILE], b[TILE][TILE]`. (Поскольку тут размеры константные, можно нормально завести двумерный массив. В `A` и `B` двумерное обращение придётся делать руками.)
Тут вот какая беда: внутри блока может быть несколько варпов. И они не синхронизированы. Так что после переноса в shared-память, надо подождать всех окружающих. Но что ещё нужно? Когда кто-то закончил считать, он не должен начинать копировать новые. Синтаксически так:

```
for (k; k += TILE) {
    a[...][...] = A[...];
    b[...][...] = B[...];
    __syncthreads();
    for (k2 = 0; k2 < TILE; k2++) {
        res = a[...] * b[...]
    }
    __syncthreads();
}
```

Как обрабатывать некратные вещи? Есть такой подход: когда заливаем матрицу с хоста на девайс, можно расширить матрицу нулями. Альтернативный подход: добавлять в kernel `if`'ов. Если тред попадает в матрицу, мы читаем, иначе в `a` мы пишем нолик. Посреди основных вычислений ничего не втыкаем. И в конце тоже вставлять `if`.
Что ещё важно. Важно, как именно вы обращаетесь в память. У нас в коде получается 32 запроса в глобальную память. Напоминаем: надо открыть строчку, подать команду чтения столбца и в ответ и чтение и запись выдают кучу последовательных значений. Поэтому лучше, если наши обращения — обращения к последовательным ячейкам, железка объединит обращения в одно большое (которое стои столько же, сколько одно мелкое). Причём не обязательно, чтобы соседние треды обращались прямо к соответствующим ячейкам памяти (на современных видеокартах). И важно это потому, что треды нумеруются сначала `x`, а потом `y`. Поэтому одни и те же итерации цикла внутри соседних тредов должны обращаться к соседним кусочкам памяти. То, насколько мы группируем треды, зависит от того, какого размера данные мы запрашиваем. Если мы запрашиваем `double`, вам достаточно, чтобы вместо обращались 2–4 треда. А если вы читаете по байтикам, нужно чтобы все 32 треда были рядом.
Что с shared-памятью? Это многобанковая память (т.е. есть независимые матрицы памяти, и если вы обращаетесь к одной матрице (банку), оно встаёт в очередь, иначе параллельно). И там есть чередование банков по 4 байта. Это вот что: первые 4 байта — банк 0, следующие 4 байта — банк 1. И так далее, пока банки не кончатся, следующие 4 байта —- снова 0. Пусть мы читаем `float`. Если случились запросы к адресам `0x40`, `0x48`, `0x44` и `0x3C`, мы обратимся к 4 разным банкам, и будет 4 обращения параллельно. А если мы обращаемся не к разным банкам, то нам интересует максимальное количество обращений к одному банку. Если у вас ко всем банкам 2 запроса, это идеально. Есть ещё одна фича у shared-памяти. Если вы делаете несколько запросов в одну ячейку, это один запрос. Когда мы много обращаемся к одному банку, пока остальные простаивают, это называется bank conflict. В современных архитектурах 32 банка.

Следствия отсюда. Для начала, чтобы не рисовать 32×32 на доске, будем считать, что у нас 4 банка и warp 4×4. Представим, что в нашей программе мы обрабатываем двумерный массив. И алгоритм у нас такой, что одна часть алгоритма идёт по одной координате, а другая — по другой. На первый взгляд, вне зависимости от того, что мы делаем, в одном случае мы обращаемся идеально, а во втором — всегда только к одному банку. Но на самом деле можно перепроектировать происходящее. Давайте выделим размер 5×4 shared-памяти. И тогда при обоих обращениях будет всё круто. Или можно прямо в shared-памяти перемешать элементы внутри строки при создании shared-памяти. Цена этому — вычисление, где же лежит наш элемент.
Вообще, когда вы думаете о том, что в вашем случае нужно группировать, а что нет, найдите optimisation manual к вашей карточке, и читайте, что у вас там пишут.
Так, окей. Даже вторая реализация сильно отстаёт от FLOPS'ов. Почему? Потому что один тред делает слишком много бесполезной (вспомогательной) работы относительно полезной. И если заставит один тред заставить обрабатывать несколько элементов выходной матрицы, будет лучше (и тут ещё удачи понять, какой формы будет кусочек, который считает один тред). Во-вторых, размер тайла теперь разойдётся с размером блока (теперь мы захотим перенести больше данных, чем у нас тредов). Дальше: у тайла необязательно оптимальный размер — это квадратик (спойлер, он не). И тут bank conflict'ы полезут отовсюду. И обращение к памяти не обязательно делать по одному элементу. Более это критично, если обращаемся мы по одному байту, но и для флотов тоже довольно актуально. Поэтому в CUDA есть вектора. Они пишутся как `float4` — в духе структурки из 4 элементов. В CUDA они не сильно полезные, и нужны только для того, чтобы делать запросы из памяти большего размера. AMD, кажется, умеет спрашивать до 4 32-битных значений одним запросом. Также существенное влияние может оказать разворот цикла.