Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение высшего образования «Сибирский государственный университет телекоммуникаций и информатики»

Лабораторная работа №1 по курсу «Программирование графических процессоров» на тему «Работа с глобальной памятью»

Новосибирск

2020

Выполнение лабораторной работы поможет получить навыки, требующиеся для выполнения первого и третьего заданий контрольной работы.

## Задание

1. Прочитайте главы теоретического материала под названиями "Отличия GPU от CPU", "Первая программа на CUDA C", "Алгоритм сложения двух векторов на GPU", "События, обработка ошибок и получение информации об устройстве", "Глобальная, локальная и константная память". Ответьте на контрольные вопросы и выполните контрольные задания, предложенные в конце этих глав (ответы на контрольные вопросы не нужно включать в отчёт по лабораторной работе).
2. Реализуйте параллельный алгоритм умножения AxV, где A – матрица, V – вектор.
3. Реализуйте параллельный алгоритм умножения VxA, где A – матрица, V – вектор.
4. Постройте графики зависимости времени выполнения алгоритма от размера матрицы и вектора (Размеры матрицы 1000x500, 1000x1000, 1500x1000, 2000x1000, 2000x1500, 2500x1500, 2500x2000).
5. Прочитайте главу "Профилирование программ", ответьте на контрольные вопросы в конце главы (ответы на контрольные вопросы не нужно включать в отчёт по лабораторной работе).
6. Проанализируйте, реализованные алгоритмы при помощи утилиты nvprof на эффективность доступа к глобальной памяти.

## Методические указания по выполнению лабораторной работы

Для того чтобы распараллелить алгоритм AxV взглянем на код последовательного алгоритма. Пусть есть матрица A размером [NxM], где N – количество строк, M – количество столбцов. И вектор V размером M (для умножения нужно чтобы размер вектора совпадал с количеством столбцов в матрице). Результатом такого умножения будет вектор C размера N. Ниже представлен код алгоритма:

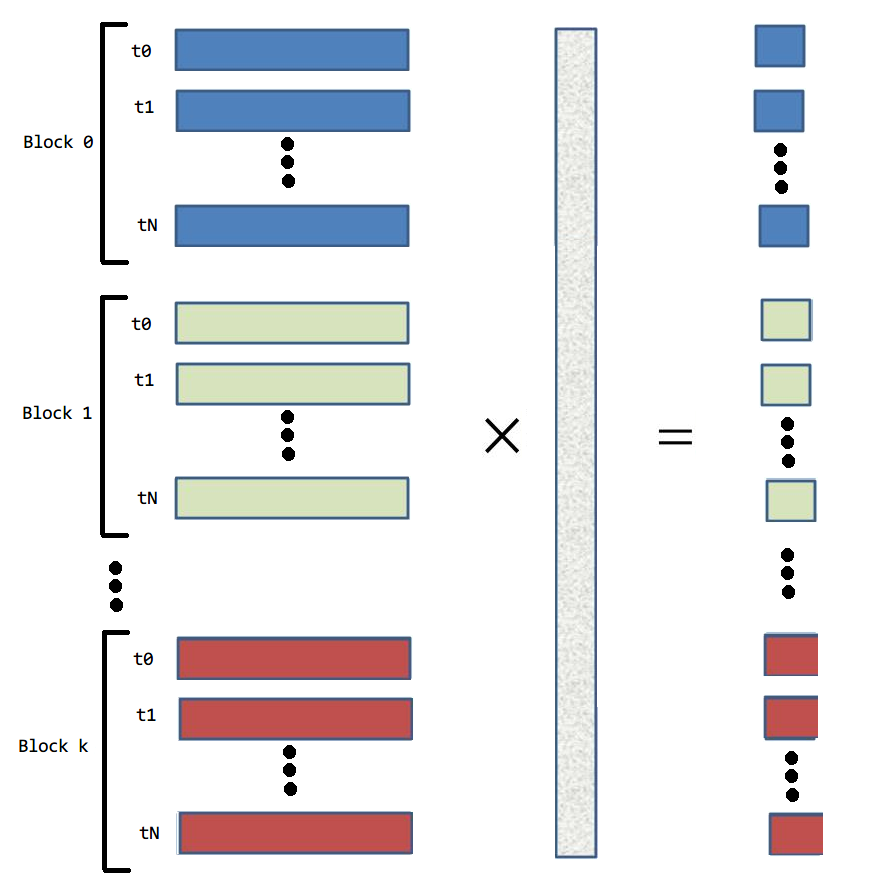
for(int i = 0; i < N; ++i) {  
 for(int j = 0; j < M) {  
 C[i] += A[i][j] \* V[j];  
 }  
}

Для того чтобы получить i-ый элемент результирующего вектора нужно взять i-ую строку матрицы A, все её элементы попарно умножить на элементы вектора и результат умножений сложить.

Теперь нужно выделить части не зависящие друг от друга. Если проанализировать работу алгоритма, то можно увидеть, что вычисление C[i] не зависит от вычислений, проводимых для расчёта других элементов результирующего вектора C. Тогда можно запустить код

for(int j = 0; j < M) {  
 C[i] += A[i][j] \* V[j];  
}

на разных вычислителях, а за индекс i взять порядковый номер вычислителя. На GPU таким вычислителем будет нить, а код нужно оформить в функцию-ядро. Важно помнить, что нити не существуют сами по себе, они группируются в блоки, а блоки вмещают не более 1024 нити. Но блоков можно запустить очень большое количество, поэтому будем считать, что их количество не ограничено и их нужно запустить достаточно, чтобы у каждой строки матрицы была своя нить. Количество блоков, которое понадобится можно вычислить по формуле N/THREAD\_PER\_BLOCK + 1, где N размер данных (в нашем случае это количество строк в матрице), а THREADS\_PER\_BLOCK – размер блока (если не знаете какой размер выбрать, возьмите 128. Важно чтобы размер был кратен 32). Единица прибавляется в конце т.к. если N не кратно размеру блока, то количество блоков будет на 1 меньше чем нужно, потому что при использовании целочисленного деления результат округлится до ближайшего меньшего целого значения. Из-за этого останется "хвост", который не распределён по нитям, но нам точно известно, что он меньше размера блока, поэтому добавляем ещё один блок для вычислений над этими данными. На рисунке 1 изображена схема распределения вычислений по нитям и блокам.



*Рис. 1 – умножение матрицы на вектор*

На рисунке видно, что каждая нить работает только с одной строкой и вектором, а результатом её работы является один элемент. После того как все нити отработают результирующий вектор будет полностью вычислен.

Ещё одна проблема – нити имеют свой номер только внутри блока. Чтобы вычислить глобальный номер нити, который будет браться вместо индекса i нужно размер блока умножить на номер блока и прибавить к нему номер нити в блоке (если забыли, где хранятся все эти величины просмотрите ещё раз теоретический материал).

Теперь опишите функцию-ядро – функцию которая будет исполняться каждой нитью GPU. Она должна принимать на вход адрес матрицы A, адрес вектора V, адрес результирующего вектора C, вычислять глобальный номер нити и в цикле вычислять элемент результирующего вектора по алгоритму, описанному выше, номер которого соответствует глобальному номеру нити.

В функции main выделите на хосте память под матрицу, вектор V и результирующий вектор и проинициализируйте их целыми числами (результирующий проинициализируйте нулями). Затем выделите память под матрицу, вектор и результирующий вектор на устройстве. Матрицу на GPU расположите в линейной памяти и используйте для её выделения функцию cudaMallocPitch. В выделенную на устройстве память скопируйте данные с хоста при помощи функции cudaMemcpy2D. Для выделения памяти на устройстве под векторы и копирования данных на устройство используйте cudaMalloc и cudaMemcpy соответственно.

После этого запустите функцию-ядро для вычислений и передайте в качестве параметра запуска размер блока и количество блоков, а в качестве параметров функции передайте адреса выделенной памяти на устройстве. После этого на хосте вызовите функцию cudaDeviceSynchronize, которая будет ожидать завершения работы всех исполняющихся нитей. Затем скопируйте при помощи функции cudaMemcpy результирующий вектор из памяти устройства в память хоста. Проверьте правильно ли сделаны расчёты.

Добавьте к коду программы замеры времени по аналогии из главы "События, обработка ошибок и получение информации об устройстве" и проведите эксперименты с указанными в задании размерами матрицы и вектора.

Главный показатель эффективности доступа к памяти – высокая пропускная способность и высокий процент попаданий в кэш (если используется кэширующий доступ к памяти). Чтобы проанализировать программу на эффективность обращения к глобальной памяти воспользуйтесь консольной утилитой nvprof и следующими метриками:

dram\_utilization – уровень пропускной способности dram относительно пиковой пропускной способности (от 0 до 10).  
dram\_read\_throughput – пропускная способность считывания из dram.  
dram\_write\_throughput – пропускная способность записи в dram.  
global\_hit\_rate – процент попаданий в L1/texture кэш.  
gld\_throughput – пропускная способность считывания из глобальной памяти  
gld\_reqested\_throughput – эффективная пропускная способность считываний из глобальной памяти.  
gld\_efficiency – эффективность считываний из глобальной памяти – отношение эффективной пропускной способности считываний из глобальной памяти к общей пропускной способности считываний из глобальной памяти.  
gst\_throughput – пропускная способность записи в глобальную память.  
gst\_requested\_throughput – эффективная пропускная способность записи в глобальную память.  
gst\_efficiency – эффективность записи в глобальную – отношение gst\_requested\_throughput к gst\_throughput.

Проанализируйте и объясните полученные результаты.

По аналогии с реализацией параллельного алгоритма умножения матрицы на вектор реализуйте параллельный алгоритм умножения вектора на матрицу. Схема доступа нитей и блоков к данным показана на рисунке 2.

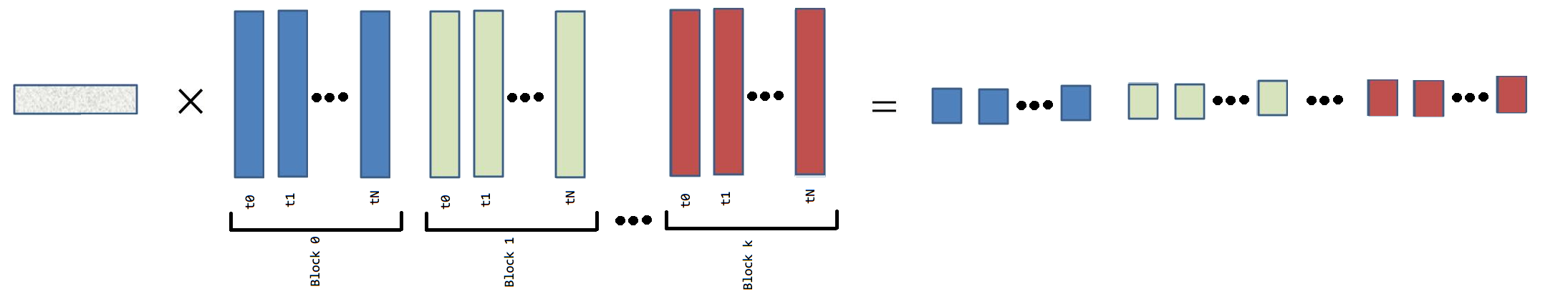


Рис. 2 – умножение вектора на матрицу

Теперь каждая нить обращается к столбцу матрицы, а не к строке.