Федеральное агентство связи

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

Лабораторная работа №2 по курсу «Программирование графических процессоров» на тему «Работа с разделяемой памятью»

Новосибирск

2020

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

## Задание

1. Прочитайте главу из теоретического материала "Разделяемая память" и ответьте на контрольные вопросы (ответы на контрольные вопросы не нужно включать в отчёт по лабораторной работе).
2. Оптимизируйте алгоритмы, реализованные в лабораторной работе №1 при помощи разделяемой памяти.
3. Постройте графики зависимости времени выполнения алгоритма от размера матрицы и вектора (Размеры матрицы 1000x500, 1000x1000, 1500x1000, 2000x1000, 2000x1500, 2500x1500, 2500x2000).
4. Проанализируйте, реализованные алгоритмы при помощи утилиты nvprof на эффективность доступа к глобальной памяти.

## Методические указания по выполнению лабораторной работы

Разделяемая память – это своего рода кэш. Ускорения от использования разделяемой памяти можно достичь только если к каким-то данным происходит многократное обращение. Тогда переместив их в разделяемую память из глобальной можно сократить время затраченное на обращение в память за счёт высокой скорости разделяемой памяти.

Рассмотрим схему параллельного умножения матрицы на вектор.



*Рис. 1 – схема умножения матрицы на вектор*

Каждая нить берёт по одной строке матрицы и умножает попарно элементы строки на элементы вектора. Все нити используют один и тот же вектор. Скопировав вектор в разделяемую память можно получить ускорения выполнения алгоритма. Но проблема разделяемой памяти в том, что её объём очень мал по сравнению с объёмом глобальной памяти. Поэтому нужно предусмотреть ситуацию когда вектор полностью в память не помещается.

Учитывая, что разделяемой памяти может не хватить нужно модифицировать алгоритм таким образом чтобы часть вектора копировалась в разделяемую память. Нити умножали часть строки на часть вектора, расположенного в разделяемой памяти. Затем копировали следующую часть вектора в разделяемую память и умножали элементы следующей части строки и т.д. как показано на рисунке 2.



*Рис. 2 – умножение нитью строки на вектор с копированием частей вектора в разделяемую память*

Копирование частей вектора можно производить параллельно, каждая нить может копировать в разделяемую память один или несколько элементов. Таким образом псевдокод алгоритма будет выглядеть следующим образом:

//Выделяем разделяемую память
//Количество элементов будет равно количеству нитей в блоке.
\_\_shared\_\_ shared\_vector[THREAD\_PER\_BLOCK];

//k - номер части вектора. +1 потому что M не обязательно делится нацело
for(k = 0; k < (M / THREAD\_PER\_BLOCK) + 1; ++k) {
 //p - реальный номер элемента вектора
 //который будет скопирован нитью в разделяемую память
 p = threadIdx.x + THREAD\_PER\_BLOCK \* k;
 //Проверка на выход за пределы вектора
 if(p < M) {
 //Копирование из глобальной памяти в разделяемую
 //Т.к. размер разделяемой памяти равен размеру блока
 //Каждая нить может скопировать по одному элементу
 shared\_vector[threadIdx.x] = V[threadIdx.x + THREAD\_PER\_BLOCK \* k];
 }
 //Синхронизация. Нужно дождаться пока все нити скопируют свои элементы
 //Потому что каждая нить использует скопированную часть вектора полностью
 \_\_syncthread();
 //i - глобальный номер нити, N - количество строк в матрице
 //Проверка не выходим ли за границы матрицы
 if (i < N) {
 //Умножение части вектора на часть строки
 //j - порядковый номер элемента в части вектора или строки матрицы
 //чтобы получить реальный номер элемента матрицы
 //нужно прибавить j количеству полностью обработанных элементов
 for(j = 0; j < THREAD\_PER\_BLOCK; ++j) {
 C[i] += A[i][j + k \* THREAD\_PER\_BLOCK] \* shared\_vector[j];
 }
 }
 //Синхронизация нужна для того чтобы убедиться
 //что все нити закончили работу с частью вектора
 //потому что следующая операция - перезапись разделяемой памяти
 \_\_syncthread();
}

В данном коде можно сделать ещё одну оптимизацию – перенести операции над вектором C в разделяемую память. И скопировать результат в глобальную только после того как результат будет посчитан. С учётом этого алгоритм будет выглядеть следующим образом:

//Выделяем разделяемую память
//Количество элементов будет равно количеству нитей в блоке.
\_\_shared\_\_ shared\_vector[THREAD\_PER\_BLOCK];
\_\_shared\_\_ shared\_c[THREAD\_PER\_BLOCK];

shared\_c[threadId.x] = 0;
\_\_syncthread();

//k - номер части вектора. +1 потому что M не обязательно делится нацело
for(k = 0; k < (M / THREAD\_PER\_BLOCK) + 1; ++k) {
 //p - реальный номер элемента вектора
 //который будет скопирован нитью в разделяемую память
 p = threadIdx.x + THREAD\_PER\_BLOCK \* k;
 //Проверка на выход за пределы вектора
 if(p < M) {
 //Копирование из глобальной памяти в разделяемую
 //Т.к. размер разделяемой памяти равен размеру блока
 //Каждая нить может скопировать по одному элементу
 shared\_vector[threadIdx.x] = V[threadIdx.x + THREAD\_PER\_BLOCK \* k];
 }
 //Синхронизация. Нужно дождаться пока все нити скопируют свои элементы
 //Потому что каждая нить использует скопированную часть вектора полностью
 \_\_syncthread();
 //i - глобальный номер нити, N - количество строк в матрице
 //Проверка не выходим ли за границы матрицы
 if (i < N) {
 //Умножение части вектора на часть строки
 //j - порядковый номер элемента в части вектора или строки матрицы
 //чтобы получить реальный номер элемента матрицы
 //нужно прибавить j количеству полностью обработанных элементов
 for(j = 0; j < THREAD\_PER\_BLOCK; ++j) {
 shared\_c[threadIdx.x] += A[i][j + k \* THREAD\_PER\_BLOCK] \* shared\_vector[j];
 }
 }
 //Синхронизация нужна для того чтобы убедиться
 //что все нити закончили работу с частью вектора
 //потому что следующая операция - перезапись разделяемой памяти
 \_\_syncthread();
}
if ( i < N ) {
 С[i] = shared\_c[threadId.x];
}
\_\_syncthread();

Модификация кода хоста в этой лабораторной работе не требуется

Для анализа эффективности доступа к разделяемой памяти используйте утилиту nvprof. События для профилирования: shared\_ld\_bank\_conflict – количество конфликтов банков памяти при считывании данных, shared\_st\_bank\_conflict– количество конфликтов банков памяти при записи данных. И метрики: shared\_efficiency – эффективность использования пропускной способности шины данных разделяемой памяти, shared\_load\_transactions\_per\_request – количество транзакций при каждом запросе к разделяемой памяти.

По аналогии оптимизируйте алгоритм умножения вектора на матрицу.