МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №4**

**по курсу «Программирование графических процессоров»**

**Работа с матрицами. Метод Гаусса.**

Выполнил: М.А. Бронников

Группа: 8О-407Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2020

**Условие**

**Цель работы:** Использование объединения запросов к глобальной памяти.

Реализация метода Гаусса с выбором главного элемента по столбцу. Ознакомление с

библиотекой алгоритмов для параллельных расчетов Thrust.

**Вариант 4.** LU-разложение матрицы.

**Программное и аппаратное обеспечение**

Device: GeForce GT 545

Размер глобальной памяти: 3150381056

Размер константной памяти : 65536

Размер разделяемой памяти: 49152

Регистров на блок: 32768

Максимум потоков на блок: 1024

Количество мультипроцессоров : 3

OS: Linux Mint 20 Cinnamon

Редактор: VSCode

**Метод решения**

Поскольку в задании требуется получить объединенную матрицу C = L + U - E, то будем иттеративно от 0 до n делать шаг, состоящий из 4 частей:

1. Нахождение максимального по модулю элемента в i-ом столбце ниже главной диагонали, а также номер строки j в котором находиться этот элемент.
2. Замена i-ой и j-ой строки местами в матрице.
3. Деление всех элементов i-ого столбца на найденный максимальный элемент.
4. Применение ко всем оставшимся элементам матрицы ниже главной диагонали формулы: uk, m -= uk, i \* ui, m

Результат описанных действий — искомая матрица C.

**Описание программы**

Я написал 3 различных реализаций, позволяющих решить эту задачу. Рассмотрим

самую простую из них:

На CPU мы выделяем память, считываем данные в **транспонированном виде** для упрощенного поиска минимума по столбцу с помощью иттераторов thrust после чего запускаем цикл от 0 до n-1, в котором последовательно выполняем выполняем описанные выше действия:

double\* h\_C = (double\*) malloc(n \* n \* sizeof(double));

unsigned\* h\_p = (unsigned\*) malloc(n \* sizeof(unsigned));

double\* d\_C;

throw\_on\_cuda\_error(cudaMalloc((void\*\*) &d\_C, sizeof(double) \* n \* n));

for(unsigned i = 0; i < n; ++i){

h\_p[i] = i; // init of permutation vector

for(unsigned j = 0; j < n; ++j){

cin >> h\_C[j\*n + i];

}

}

throw\_on\_cuda\_error(cudaMemcpy(d\_C, h\_C, sizeof(double) \* n \* n, cudaMemcpyHostToDevice));

for(unsigned i = 0; i < n - 1; ++i){

auto it\_beg = thrust::device\_pointer\_cast(d\_C + i\*n);

auto max\_elem = thrust::max\_element(it\_beg + i, it\_beg + n, abs\_comparator());

unsigned max\_idx = max\_elem - it\_beg;

double max\_val = \*max\_elem;

if(i != max\_idx){

swap\_lines<<<BLOCKS, THREADS>>>(d\_C, n, i, max\_idx);

h\_p[i] = max\_idx;

cudaThreadSynchronize();

}

gauss\_step\_L<<<BLOCKS, THREADS>>>(d\_C, n, i, max\_val);

cudaThreadSynchronize();

gauss\_step\_U<<<BLOCKS, THREADS>>>(d\_C, n, i);

throw\_on\_cuda\_error(cudaGetLastError());

}

Функция cudaThreadSynchronize нужна для синхронизации потоков, во избежание гонки потоков из разных функций. Код самих kernel не имеет никаких особенных деталей, кроме того, что для оптимизированного доступа к памяти соседние потоки обращаются к соседним элементам массива.

Однако при размере массива не кратном 256 данная реализация не раскрывает в полной мере преимущество объединения потоков, поэтому напишем другую реализацию, в которой дополним каждую строку до размера, кратного 256:

unsigned size = get\_allign\_size(n);

host\_vector<double> h\_C(size \* n);

device\_vector<double> d\_C;

host\_vector<unsigned> h\_p(n);

Заметим, что для удобства можно использовать предоставляемые thrust контейнеры host\_vector и device\_vector, упрощающие работу с памятью в cuda.

Тогда в kernel потоки будут обращаться к таким участкам памяти, чтобы работало объединение запросов:

\_\_global\_\_ void gauss\_step\_U(double\* C, unsigned n, unsigned size, unsigned col){

unsigned i\_idx = threadIdx.x;

unsigned j\_idx = blockIdx.x;

unsigned i\_step = blockDim.x;

unsigned j\_step = gridDim.x;

for(unsigned jndex = j\_idx + col + 1; jndex < n; jndex += j\_step){

unsigned idx0 = jndex\*size;

double C\_jc = C[idx0 + col];

unsigned index = i\_idx + col + 1 - ((col + 1) & 255);

if(index > col && index < n){

//printf("[%d, %d] = %f\n", index, jndex, C[idx0 + index]);

C[idx0 + index] -= C[size\*col + index] \* C\_jc;

//printf("[%d, %d] = %f\n", index, jndex, C[idx0 + index]);

}

for(index += i\_step; index < n; index += i\_step){

C[idx0 + index] -= C[size\*col + index] \* C\_jc;

}

}

}

Также можно использовать держать матрицу не в транспонированном виде, что ускорит swap двух линий, однако усложнит поиск максимального элемента столбца с помощью средств cuda. Для решения этой задачи я использовал класс из официального github nvidia: straded\_range и использовал следующий итератор:

strided\_range<thrust::device\_vector<double>::iterator> range(

d\_C.begin() + i,

d\_C.end(),

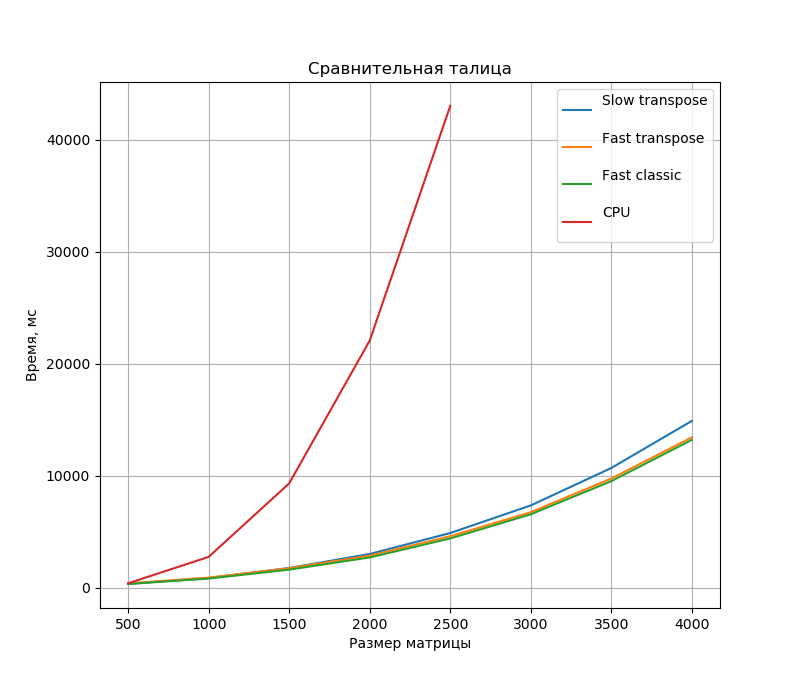
align

);

Такое размещение в памяти не только ускоряет перестановку на GPU, но и более кэш-дружелюбно на CPU.

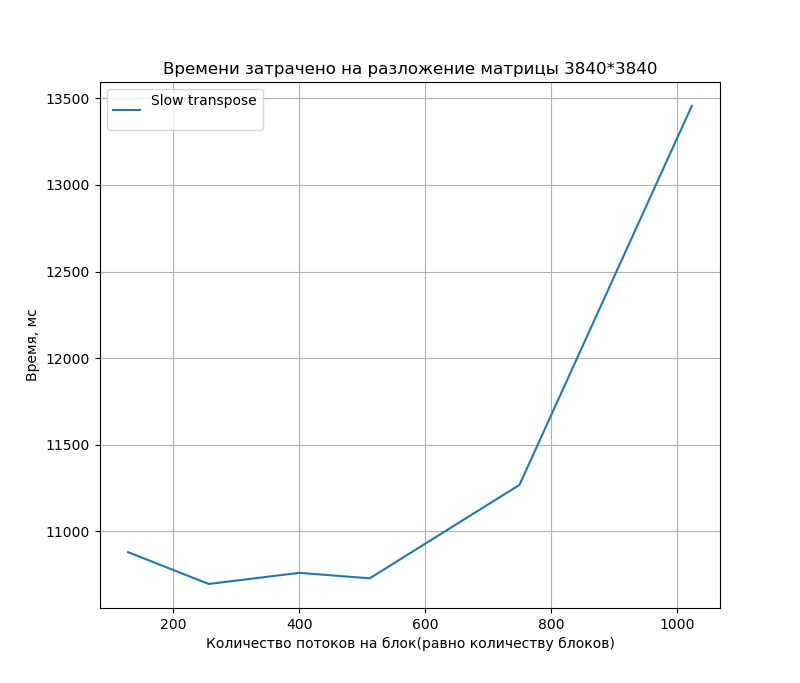
**Результаты**

Рассмотрим как зависит время выполнения от порядка матрицы в 3 разных реализациях:



Лучшей реализацией оказалась ускоренная реализация с транспонированной схемой хранения матрицы в памяти с небольшим отрывом от третей реализации. При этом аналогичная реализация на CPU быстро расходиться на графике с реализациями на GPU, поскольку имеет другой порядок сложности алгоритма.

Покажем как размерность сетки потоков влияет на производительность. В силу специфики алгоритма количество блоков равно количеству потоков в блоке:



На графике заметны небольшие падения производительности при количестве потоков, не кратном 256, поскольку в таком случае объединение запросов перестает работать, что увеличивает общее время выполнения.

Однако самое странное, это резкий рост времени выполнения программы при количестве потоков, больше 512, поскольку в таком случае размер массивва перестает делиться нацело на количество потоков, что заставляет простаивать часть потоков. В тоже время уходит больше времени на инициализацию потоков.

**Выводы**

В данной лабораторной работе я встретил снова знакомый мне с курса численных методов алгоритм LU разложения с перестановками, однако мне пришлось написат ь его реализацию в непривычном, но эффективном виде на GPU. Реализованный мной алгоритм является одним из самых эффективных методов для неиттерационного решения СЛАУ.

Помимо этого я познакомился с тем, как эффективно обращаться к глобальной памяти с использованием объединения запросов, а также узнал про то, как пользоваться библиотекой thrust, которая предостовляет возможность эффективных и безопасных вычислений.

В ходе выполнения столкнулся с массой проблем, однако практически все они были следствием моей невнимательности и несмотря на все трудности я выполнил работу, которая наглядно показала мне, насколько большое преимущество может дать использование графического процессора вместо центрального в задачах работы с матрирцами.