Блочное перемножение матриц на CUDA

GPU Teaching Kit – Accelerated Computing

# Цели

Реализовать блочное произведение полностью заполненных матриц с использованием разделяемой памяти.

# Предварительные требования

Прежде чем начать, убедитесь, что:

* вы выполнили лабораторную "Сложение векторов"
* вы прошли все лекции текущего модуля

# Ход выполнения

Отредактируйте код, чтобы выполнить следующие действия:

* выделить память на устройстве
* скопировать память хоста на устройство
* инициализировать размерности блока нитей и ядра
* запустить ядро CUDA
* выполнить блочное произведение матриц, используя разделяемую память
* скопировать результат на хост
* освободить память устройства

Места, куда следует вставить код, помечены комментариями //@@.

# Инструкция по установке

Последнюю версию исходного кода лабораторной, наряду со скриптами сборки, можно найти в репозитории [Bitbucket](https://bitbucket.org/hwuligans/gputeachingkit-labs/src/master/Module4). Инструкции к Cmake и сборки лабораторной можно найти в файле [README](https://bitbucket.org/hwuligans/gputeachingkit-labs/src/master/README.md) в корневой директории репозитория.

Исполняемый файл, являющийся результатом компиляции лабораторной, можно запустить следующей командой:

./TiledMatrixMultiplication\\_Template -e <expected.raw> \  
 -i <input0.raw>,<input1.raw> -o <output.raw> -t matrix

где <expected.ppm> – имя ожидаемого результата, <input.ppm> – входной набор данных и <output.ppm> – необязательный путь для результатов. Набор данных можно сгенерировать, используя генератор, собранный во время компиляции.

# Вопросы

1. Сколько операций с плавающей точкой будет произведено вашим ядром? Объясните.
2. Сколько чтений из глобальной памяти будет произведено вашим ядром? Объясните.
3. Сколько записей в глобальную память будет произведено вашим ядром? Объясните.
4. Опишите, какие дальнейшие оптимизации помогли бы вам улучшить производительность ядра.
5. Сравните сложности реализации этой и предыдущей лабораторной. С какими сложностями вы столкнулись в данной лабораторной?
6. Представим: имеются матрицы, размерность которых превышает максимальную размерность нитей. Набросайте алгоритм, который выполняет перемножение этих матриц.
7. Представим: имеются матрицы, которые не умещаются в глобальной памяти. Набросайте алгоритм, который выполняет произведение таких матриц.

# Шаблон кода

Представленный код предлагается как отправная точка. Импорты, экспорты и проверка решения уже представлены в коде. Требуется вставить свой код в области, обозначенные //@@. Остальной код трогать не нужно. Руководство описывает функционал методов с префиксом wb\*.

#include <wb.h>  
  
#define wbCheck(stmt) \  
 do { \  
 cudaError\_t err = stmt; \  
 if (err != cudaSuccess) { \  
 wbLog(ERROR, "Failed to run stmt ", #stmt); \  
 wbLog(ERROR, "Got CUDA error ... ", cudaGetErrorString(err)); \  
 return -1; \  
 } \  
 } while (0)  
  
// Вычисление C = A \* B  
\_\_global\_\_ void matrixMultiplyShared(float \*A, float \*B, float \*C,  
 int numARows, int numAColumns,  
 int numBRows, int numBColumns,  
 int numCRows, int numCColumns) {  
 //@@ Вставьте ваш код произведения матриц  
 //@@ В этой лабораторной работе используйте разделяемую память  
}  
  
int main(int argc, char \*\*argv) {  
 wbArg\_t args;  
 float \*hostA; // Матрица A  
 float \*hostB; // Матрица B  
 float \*hostC; // Матрица C  
 float \*deviceA;  
 float \*deviceB;  
 float \*deviceC;  
 int numARows; // количество строк матрицы A  
 int numAColumns; // количество столбцов матрицы A  
 int numBRows; // количество строк матрицы B  
 int numCRows; // количество строк матрицы C (установите  
 // это значение сами)  
 int numCColumns; // количество столбцов матрицы C (установите   
 //это значение сами)  
  
 args = wbArg\_read(argc, argv);  
  
 wbTime\_start(Generic, "Importing data and creating memory on host");  
 hostA = (float \*)wbImport(wbArg\_getInputFile(args, 0), &numARows,  
 &numAColumns);  
 hostB = (float \*)wbImport(wbArg\_getInputFile(args, 1), &numBRows,  
 &numBColumns);  
 //@@ Установите numCRows и numCColumns  
 numCRows = 0;  
 numCColumns = 0;  
 //@@ Выделение памяти под матрицу hostC  
 wbTime\_stop(Generic, "Importing data and creating memory on host");  
  
 wbLog(TRACE, "The dimensions of A are ", numARows, " x ", numAColumns);  
 wbLog(TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);  
  
 wbTime\_start(GPU, "Allocating GPU memory.");  
 //@@ Выделите память GPU  
  
 wbTime\_stop(GPU, "Allocating GPU memory.");  
  
 wbTime\_start(GPU, "Copying input memory to the GPU.");  
 //@@ Скопируйте память с хоста на GPU  
  
 wbTime\_stop(GPU, "Copying input memory to the GPU.");  
  
 //@@ Инициализируйте размерности блоков и сетки  
  
 wbTime\_start(Compute, "Performing CUDA computation");  
 //@@ Запустите ядро GPU  
  
 cudaDeviceSynchronize();  
 wbTime\_stop(Compute, "Performing CUDA computation");  
  
 wbTime\_start(Copy, "Copying output memory to the CPU");  
 //@@ Скопируйте память обратно с GPU на хост  
  
 wbTime\_stop(Copy, "Copying output memory to the CPU");  
  
 wbTime\_start(GPU, "Freeing GPU Memory");  
 //@@ Освободите память GPU  
  
 wbTime\_stop(GPU, "Freeing GPU Memory");  
  
 wbSolution(args, hostC, numCRows, numCColumns);  
  
 free(hostA);  
 free(hostB);  
 free(hostC);  
  
 return 0;  
}