Стандартное произведение матриц

GPU Teaching Kit – Accelerated Computing

# Цели

Реализовать простое произведение полностью заполненных матриц “в лоб”. В данной лабораторной производить оптимизации типа разбиения на блоки и использования разделяемой памяти не требуется.

# 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
* 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:

./BasicMatrixMultiplication\_Stream\_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.

ОТВЕТ: numCRows \* numCCols скалярных произведений = 2 \* numCRows \* numCCols \* numACols.

1. Сколько чтений из глобальной памяти будет произведено вашим ядром? Объясните.

ОТВЕТ: numCRows \* numCCols скалярных произведений = 2 \* numCRows \* numCCols \* numACols.

1. Сколько записей в глобальную память будет произведено вашим ядром? Объясните.

ОТВЕТ: по количеству выходных элементов матрицы. numCRows \* numCCols.

1. Опишите возможный вариант оптимизации, которую можно произвести для того, чтобы увеличить производительность вашего ядра.

ОТВЕТ: Разбить матрицу на блоки, помещая их в разделяемую память, чтобы снизить количество обращений к глобальной памяти.

1. Назовите три применения произведения матриц.

ОТВЕТ: произведение матриц используется в почти каждом ресурсоемком приложении. Например, нейронные сети, компьютерная графика, дифференциальные уравнения в частных производных.

# 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 matrixMultiply(float \*A, float \*B, float \*C, int numARows,  
 int numAColumns, int numBRows,  
 int numBColumns, int numCRows,  
 int numCColumns) {  
 //@@ Вставьте ваш код произведения матриц  
}  
  
int main(int argc, char \*\*argv) {  
 wbArg\_t args;  
 float \*hostA; // Матрица A  
 float \*hostB; // Матрица B  
 float \*hostC; // Выходная 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 numBColumns; // количество столбцов 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;  
}

# Код решения

Далее представлен возможный вариант решения. Подразумевается использование решения только преподавателем, оно не должно передаваться студентам.

#include <wb.h>  
  
// Вычисление C = A \* B  
// Sgemm обозначает stands for single precision general matrix-matrix  
// multiply – стандартное произведение матрицы на матрицу с  
// одинарной точностью  
\_\_global\_\_ void sgemm(float \*A, float \*B, float \*C, int numARows,  
 int numAColumns, int numBRows, int numBColumns) {  
 int row = blockIdx.y \* blockDim.y + threadIdx.y;  
 int col = blockIdx.x \* blockDim.x + threadIdx.x;  
 if (row < numARows && col < numBColumns) {  
 float sum = 0;  
 for (int ii = 0; ii < numAColumns; ii++) {  
 sum += A[row \* numAColumns + ii] \* B[ii \* numBColumns + col];  
 }  
 C[row \* numBColumns + col] = sum;  
 }  
}  
  
#define wbCheck(stmt) \  
 do { \  
 cudaError\_t err = stmt; \  
 if (err == cudaSuccess) { \  
 wbLog(ERROR, "Failed to run stmt ", #stmt); \  
 return -1; \  
 } \  
 } while (0)  
  
int main(int argc, char \*\*argv) {  
 wbArg\_t args;  
 float \*hostA; // Матрица A  
 float \*hostB; // Матрица B  
 float \*hostC; // Выходная 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 numBColumns; // количество столбцов матрицы B  
 int numCRows;  
 int numCColumns;

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);  
 //@@ Выделение памяти под матрицу hostC  
 hostC = (float \*)malloc(numARows \* numBColumns \* sizeof(float));  
 wbTime\_stop(Generic, "Importing data and creating memory on host");  
  
 numCRows = numARows;  
 numCColumns = numBColumns;  
  
 wbLog(TRACE, "The dimensions of A are ", numARows, " x ", numAColumns);  
 wbLog(TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);  
 wbLog(TRACE, "The dimensions of C are ", numCRows, " x ", numCColumns);  
  
 wbTime\_start(GPU, "Allocating GPU memory.");  
 //@@ Выделите память GPU  
 wbCheck(cudaMalloc((void \*\*)&deviceA,  
 numARows \* numAColumns \* sizeof(float)));  
 wbCheck(cudaMalloc((void \*\*)&deviceB,  
 numBRows \* numBColumns \* sizeof(float)));  
 wbCheck(cudaMalloc((void \*\*)&deviceC,  
 numARows \* numBColumns \* sizeof(float)));  
 wbTime\_stop(GPU, "Allocating GPU memory.");  
  
 wbTime\_start(GPU, "Copying input memory to the GPU.");  
 //@@ Скопируйте память с хоста на GPU   
 wbCheck(cudaMemcpy(deviceA, hostA,  
 numARows \* numAColumns \* sizeof(float),  
 cudaMemcpyHostToDevice));  
 wbCheck(cudaMemcpy(deviceB, hostB,  
 numBRows \* numBColumns \* sizeof(float),  
 cudaMemcpyHostToDevice));  
 wbTime\_stop(GPU, "Copying input memory to the GPU.");  
  
 //@@ Инициализируйте размерности блоков и сетки  
 dim3 blockDim(16, 16);  
 dim3 gridDim(ceil(((float)numAColumns) / blockDim.x),  
 ceil(((float)numBRows) / blockDim.y));  
  
 wbLog(TRACE, "The block dimensions are ", blockDim.x, " x ", blockDim.y);  
 wbLog(TRACE, "The grid dimensions are ", gridDim.x, " x ", gridDim.y);  
  
 wbTime\_start(Compute, "Performing CUDA computation");  
 //@@ Запустите ядро GPU  
 wbCheck(cudaMemset(deviceC, 0, numARows \* numBColumns \* sizeof(float)));  
 sgemm<<<gridDim, blockDim>>>(deviceA, deviceB, deviceC, numARows,  
 numAColumns, numBRows, numBColumns);  
 cudaDeviceSynchronize();  
 wbTime\_stop(Compute, "Performing CUDA computation");  
  
 wbTime\_start(Copy, "Copying output memory to the CPU");  
 //@@ Скопируйте память обратно с GPU на хост  
  
 wbCheck(cudaMemcpy(hostC, deviceC,  
 numARows \* numBColumns \* sizeof(float),  
 cudaMemcpyDeviceToHost));  
 wbTime\_stop(Copy, "Copying output memory to the CPU");  
  
 wbTime\_start(GPU, "Freeing GPU Memory");  
 //@@ Освободите память GPU  
 cudaFree(deviceA);  
 cudaFree(deviceB);  
 cudaFree(deviceC);  
 wbTime\_stop(GPU, "Freeing GPU Memory");  
  
 wbSolution(args, hostC, numARows, numBColumns);  
  
 free(hostA);  
 free(hostB);  
 free(hostC);  
  
 return 0;  
}

Данная работа лицензирована UIUC и NVIDIA (2016) в рамках Creative Commons Attribution-NonCommercial 4.0 License.