МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №1**

**по курсу «Программирование графических процессоров»**

**Работа с матрицами. Метод Гаусса.**

Выполнил: Н.А. Зайцев

Группа: 8О-408Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2016

**Условие**

Описание задачи:

1. **Цель работы​.** Ознакомление и установка программного обеспечения для работы с программноаппаратной архитектурой параллельных вычислений(CUDA). Реализация метода Гаусса с выбором главного элемента по столбцу.
2. **Вариант 6. Нахождение ранга матрицы.**
   1. **Входные данные.** ​На первой строке заданы числа n и m -­­ размеры матрицы. В следующих n строках, записано по m вещественных чисел ­­ элементы матрицы. n \* m ≤ 10^8
   2. **Выходные данные.** ​Необходимо вывести одно число -­­ ранг матрицы.

**Программное и аппаратное обеспечение**

**GPU:**

Name : GeForce GTX 650

Compute capability : 3.0

Total Global Memory : 2147483648

Shared memory per block : 49152

Registers per block : 65536

Warp size : 32

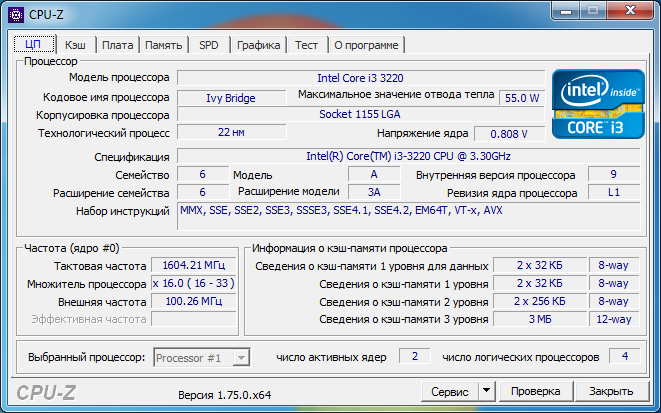
Max threads per block : (1024, 1024, 64)

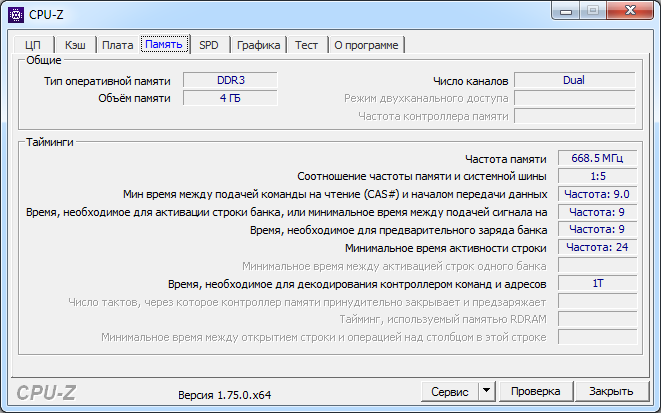
Max block : (2147483647, 65535, 65535)

Total constant memory : 65536

Multiprocessors count : 2

**CPU:**





**HDD:**

Производитель: Seagate

Модель: ST2000DM001-1CH164 ATA Device

Размер : 1,82 ТБ

**OS:**

Windows 7 Professional SP1 64-bit

**IDE:**

Microsoft Visual Studio 2013 + CUDA Plugins

**Compiler:**

nvcc

**Profiler:**

nvprof

**Метод решения**

Был реализован алгоритм Гаусса приведения матрицы к ступенчатому виду с поиском главного элемента по столбцу (максимального), обменом строк, и обнулением столбца под ведущим элементом. Обмен строк не производился в памяти, а эмулировался при помощи переадресации обращения к строкам матрицы через специальный массив индексов. После считаем ранг, пробегая по «лесенке». Распараллеливанию на CUDA подверглись только вычитание строк и заполнение массива коэффициентов, необходимых для зануления столбца и вычитания строк. Матрица линеаризована по строкам. Параметры многопоточных ядер: количество использованных потоков – 1024, блоков - размерность/кол-во потоков + 1.

**Описание программы**

Программа состоит из одного файла, но разбита на функцию main несколько ядер.  
  
Основной цикл расположенный в функции main:

unsigned int threads\_count = BLOCK\_SIZE;

unsigned int blocks\_count = MAX(width, height) / threads\_count + 1;

unsigned int offset = 0;

double \* dev\_ratios;

CSC(cudaMalloc(&dev\_ratios, sizeof(double) \* height));

for (unsigned int i = 0; i < width; i++)

{

kernel\_main << < 1, 1 >> > (dev\_matrix, i, height, width, dev\_index, device\_index\_of\_max, offset);

CSC(cudaMemcpy(&host\_index\_of\_max, device\_index\_of\_max, sizeof(int), cudaMemcpyDeviceToHost));

if (host\_index\_of\_max != -1)

{

offset++;

kernel\_count\_ratios << < height / threads\_count + 1, threads\_count >> >(dev\_matrix, i, height, width, dev\_index, offset, dev\_ratios);

kernel\_rows\_substraction << < blocks\_count, threads\_count >> > (dev\_matrix, i, height, width, dev\_index, offset, dev\_ratios);

}

}

unsigned int \* dev\_rank;

CSC(cudaMalloc(&dev\_rank, sizeof(unsigned int)));

kernel\_rank\_count << < 1, 1 >> > (dev\_matrix, height, width, dev\_index, dev\_rank);

В этом цикле реализован весь алгоритм работы программы при помощи запуска отдельных ядер. Цикл проходит по всем столбцам матрицы. Рассмотрим ядра далее.

Ядро kernel\_main:

\_\_global\_\_ void kernel\_main(double \* matrix, unsigned int i, unsigned int height, unsigned int width, unsigned int \* index, int \*index\_of\_max, unsigned int offset)

{

double max = 0.0000001;

\*index\_of\_max = -1;

for (unsigned int j = offset; j < height; j++)

{

if (fabs(matrix[index[j] \* width + i]) > max)

{

max = fabs(matrix[index[j] \* width + i]);

\*index\_of\_max = j;

}

}

if (\*index\_of\_max != -1)

{

unsigned int tmp = index[\*index\_of\_max];

index[\*index\_of\_max] = index[offset];

index[offset] = tmp;

}

}

Происходит поиск максимума в столбце и обмен строк. Отличие \*index\_of\_max от -1 сигнализирует другим ядрам об успешности операции, и что им можно работать.

Ядро kernel\_count\_ratios:

\_\_global\_\_ void kernel\_count\_ratios(double \* matrix, unsigned int i, unsigned int height, unsigned int width, unsigned int \* index, unsigned int offset, double \* ratios)

{

/\* double tmp = matrix[index[(offset - 1)] \* width + i];

for (unsigned int l = offset; l < height; l++)

{

ratios[l] = matrix[index[l] \* width + i] / tmp;

}

\*/

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

\_\_shared\_\_ double tmp;

tmp = matrix[index[(offset - 1)] \* width + i];

while (offset + tid < height)

{

ratios[offset + tid] = matrix[index[offset + tid] \* width + i] / tmp;

tid += blockDim.x \* gridDim.x;

}

}

Это ядро производит заполнение массива коэффициентов необходимых для успешного зануления столбцов и вычитания строк. В комментарии можно увидеть нераспараллеленный вариант.

Ядро kernel\_rows\_substraction:

\_\_global\_\_ void kernel\_rows\_substraction(double \* matrix, unsigned int i, unsigned int height, unsigned int width, unsigned int \* index, unsigned int offset, double \* ratios)

{

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

while (i + tid < width)

{

double factor = matrix[index[(offset - 1)] \* width + i + tid];

for (unsigned int l = offset; l < height; l++)

{

matrix[index[l] \* width + i + tid] -= ratios[l] \* factor;

}

tid += blockDim.x \* gridDim.x;

}

}

Это ядро производит зануление столбца под ведущим элементом и вычитаие строк.

Ядро kernel\_rank\_count:

\_\_global\_\_ void kernel\_rank\_count(double \* matrix, unsigned int height, unsigned int width, unsigned int \* index, unsigned int \*rank)

{

unsigned int i = 0, j = 0;

unsigned int rank\_ = 0;

while (true)

{

if (fabs(matrix[index[i] \* width + j]) > 0.0000001)

{

rank\_++;

i++;

j++;

if (i >= height || j >= width)

break;

}

else

{

j++;

if (j >= width)

break;

}

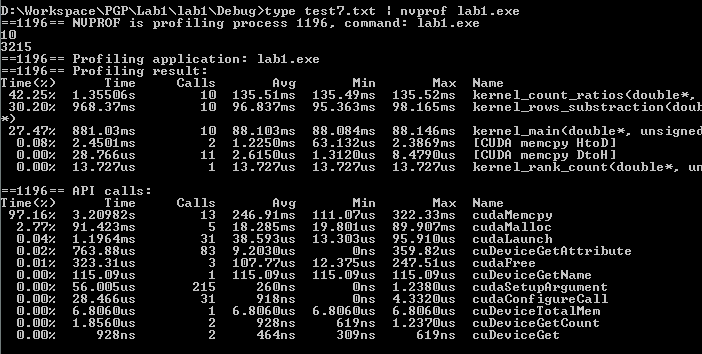
}

(\*rank) = rank\_;

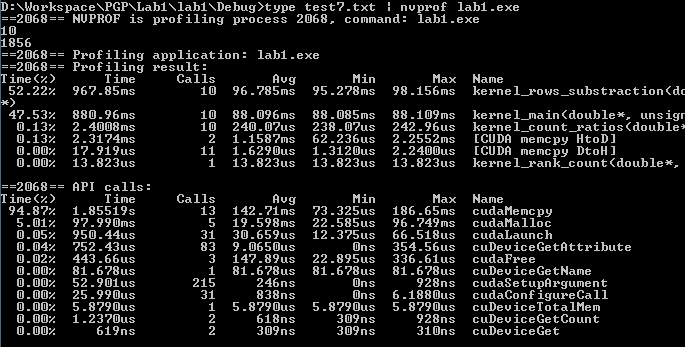
}

Производит подсчёт ранга по полученной ступенчатой матрице.

**Результаты**

В процессе написания кода появилась необходимость освоить профилятор, чтобы искать проседания в производительности. Например, можно увидеть насколько отличается производительность программы (конкретных ядер) до распараллеливания заполнения массива коэффициентов и после на тесте со 100000 строками и 10 столбцами.  
  
До:  


После:



Разница просто огромна. Профилятор является очень полезным инструментом и помог мне не один раз.

Давайте сравним производительность программы, реализованной на CPU и GPU с использованием CUDA. Вставил в код таймеры непосредственно перед и после выполнения алгоритма.

Тест с 10 строками и 100000 столбцами.

CPU:



GPU:



Распараллеленная программа работает на 200 мс быстрее. На самом деле нужно сгенерировать более объёмный тест (порядка n \* m ≤ 10^8), чтобы разница в производительности была более заметной, но мне не хватило терпения дождаться пока он запишется на диск.

**Выводы**

В результате выполнения данной работы я научился распараллеливать на видеокарте при помощи CUDA некоторые задачи линейной алгебры, а также приобрёл незабываемый опыт работы с профилятором. Наглядно увидел разницу в производительности программ, написанных с использованием CUDA и без.

**Литература**

* + - 1. Джейсон Сандерс, Эдвард Кэндрот. Технология CUDA в примерах. 2011 г.