МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(Национальный исследовательский университет)

Лабораторная № 1

Освоение программного обеспечения среды программирования nVidia

Студент ____________ Янковский А. В.
Преподаватель ____________ Семенов С. А.,

Москва, 2015

Содержание

1 Постановка задачи 3

2 Алгоритм решения 4

3 Аппаратное обеспечение 5

4 Программное обеспечение 6

5 Результаты 7

6 График, сравнение 8

7 Вывод 9

8 Приложение 10

Список литературы 15

1. Постановка задачи

Необходимо сгенерировать массив, содержащий 10^8 случайных чисел, используя возможности платформы CUDA. Полученный результат сравнить с однопоточной программой с аналогичной функциональностью.

2. Алгоритм решения

Для генерации массива случайных чисел воспользуемся библиотекой cuRAND [1], поставляемой вместе с платформой CUDA. Для этого опишем функцию __global__ void setupKernel(unsigned long seed, curandState *state), инициализирующую генератор случайных чисел, и функцию __global__ void generateRandomNum numbers, curandState *globalState, int size), выполняющую непосредственную генерацию массива случайных чисел.

3. Аппаратное обеспечение

Для выполнения задачи использовалась ЭВМ со следующими характеристиками:

- CPU - Intel Core i7 4710MQ 2.5 Ghz
- RAM - 8Gb
- GPU - nVidia GeForce 940M

4. Программное обеспечение

На используемой ЭВМ установлена ОС Linux Mint 17.2 x64 и пакет CUDA-Toolkit версии 7.0

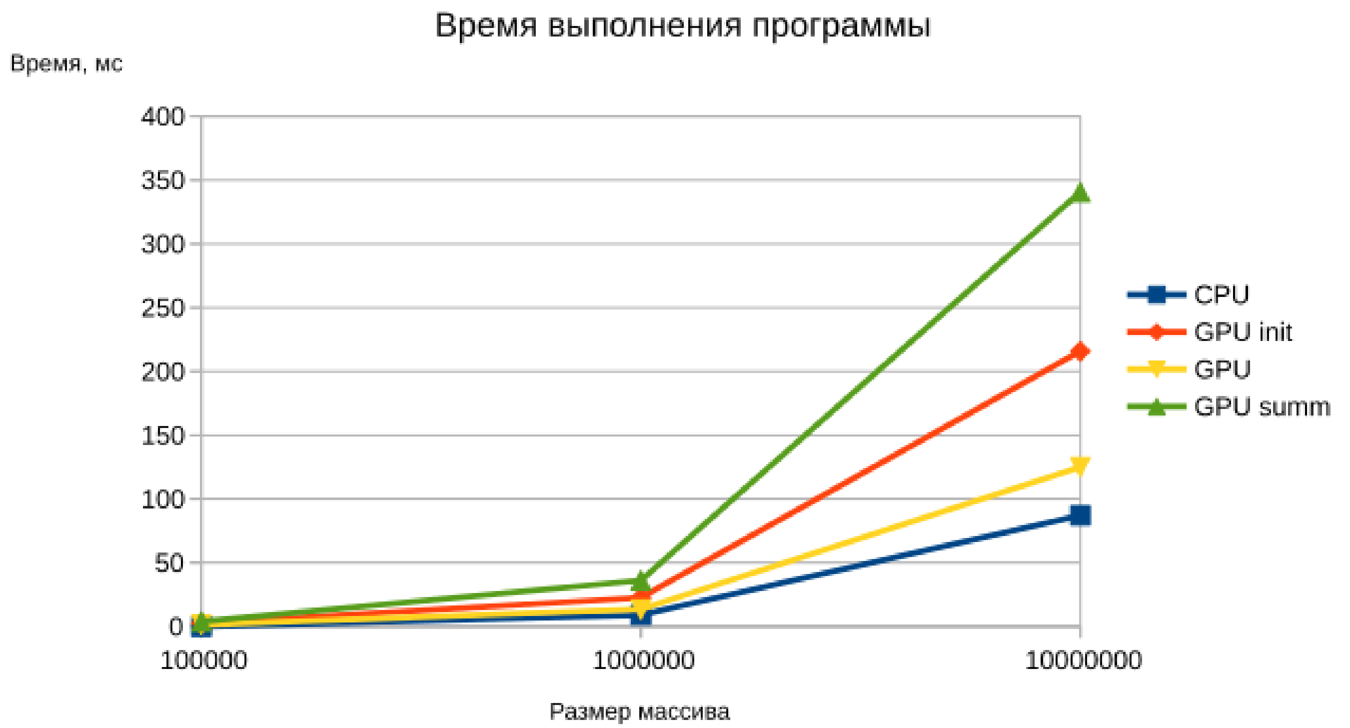
5. Результаты

В результате выполнения программы получены следующие результаты:

- Время выполнения на CPU при размере массива 10^7 - 80 мс
- Время инициализации генератора случайных чисел на GPU curand_init при размере массива 10^7 - 250 мс
- Время создания случайных чисел при размере массива 10^7 - 150 мс

Протестировать программу при требуемом размере массива в 10^8 не удалось, т.к., была получена ошибка времени выполнения out of memory.

6. График, сравнение



7. Вывод

На приведенном графике видно, что создание массива случайных чисел при помощи библиотеки cuRAND значительно медленнее, чем создание аналогичного массива в однопоточной программе, выполняемой на CPU, однако формирование данного массива в памяти GPU освобождает от необходимости копирования данных из RAM ЭВМ в память видеокарты.

8. Приложение

Листинг 1

```
#include <iostream>
#include <numeric>
#include <stdlib.h>
#include <curand.h>
#include <curand_kernel.h>
#include <time.h>
#include <stdio.h>
#include <chrono>

static void CheckCudaErrorAux (const char *, unsigned,
const char *, cudaError_t);
#define CUDA_CHECK_RETURN(value) CheckCudaErrorAux(__FILE__,
__LINE__, #value, value)


using namespace std;

__global__ void setupKernel(unsigned long seed, curandState *state) {
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    curand_init((seed << 20) + id, 0, 0, &state[id]);
}

__global__ void generateRandomNumbers(float* numbers,
curandState *globalState, int size) {

    int i = threadIdx.x + blockIdx.x * blockDim.x;

    if (i < size) {
     curandState localState = globalState[i];
```

```
     numbers[i]= curand_uniform( &localState );
     globalState[i] = localState;
    }
}

int main(void)
{
static const int WORK_SIZE = 10000 * 1000;



int nDevices;
CUDA_CHECK_RETURN(cudaGetDeviceCount(&nDevices));

if (nDevices == 0) {
return 1;
}

float *cpuData = new float[WORK_SIZE];

srand(time(0));

using clock = std::chrono::high_resolution_clock;

auto cpuThreadStart = clock::now();
for (int i = 0; i < WORK_SIZE; i++) {
*(cpuData + i) = rand();
}
auto cpuThreadStop =  clock::now();


cout << "Cpu time: " << chrono::duration_cast<chrono::milliseconds>
(cpuThreadStop - cpuThreadStart).count() << " ms" << endl;
```

```
delete[] cpuData;


float *gpuData;
CUDA_CHECK_RETURN(cudaMalloc((void **)&gpuData, sizeof(float)*WORK_SIZE));
CUDA_CHECK_RETURN(cudaMemset(gpuData, 0, sizeof(float)*WORK_SIZE));

curandState *devStates;
CUDA_CHECK_RETURN(cudaMalloc((void **)&devStates,
sizeof(curandState) * WORK_SIZE));


static const int BLOCK_SIZE = 384;
const int blockCount = (WORK_SIZE+BLOCK_SIZE-1)/BLOCK_SIZE;

cudaEvent_t setupStart = 0, setupStop = 0;
cudaEventCreate(&setupStart);
cudaEventRecord(setupStart, 0);

setupKernel<<<blockCount, BLOCK_SIZE>>>(time(NULL), devStates);

cudaEventCreate(&setupStop);
cudaEventRecord(setupStop, 0);

cudaEventSynchronize(setupStop);

float setupMilliseconds = 0;
cudaEventElapsedTime(&setupMilliseconds,
setupStart, setupStop);
cout << "Setup time: " << setupMilliseconds << " ms" << endl;
```

```
cudaEvent_t gpuThreadStart = 0, gpuThreadStop = 0;
cudaEventCreate(&gpuThreadStart);
cudaEventRecord(gpuThreadStart, 0);

generateRandomNumbers<<<blockCount, BLOCK_SIZE>>>
(gpuData, devStates, WORK_SIZE);

CUDA_CHECK_RETURN(cudaPeekAtLastError());
CUDA_CHECK_RETURN(cudaDeviceSynchronize());
cudaEventCreate(&gpuThreadStop);
cudaEventRecord(gpuThreadStop, 0);

cudaEventSynchronize(gpuThreadStop);

float gpuMilliseconds = 0;
cudaEventElapsedTime(&gpuMilliseconds,
gpuThreadStart, gpuThreadStop);

CUDA_CHECK_RETURN(cudaFree(gpuData));


cout << "Gpu time: " << gpuMilliseconds << " ms" << endl;




return 0;
}

/**
 * Check the return value of the CUDA runtime API call and exit
 * the application if the call has failed.
 */
```

```
static void CheckCudaErrorAux (const char *file, unsigned line,
const char *statement, cudaError_t err) {
if (err == cudaSuccess)
return;
std::cerr << statement<<" returned " <<
cudaGetErrorString(err) << "("<<err<< ") at
"<<file<<":"<<line << std::endl;
exit (1);
}
```

Список литературы

1. cuRAND. 2015. http://docs.nvidia.com/cuda/curand/index.html.