Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Архитектура высокопроизводительных процессоров

ОТЧЁТ

по лабораторной работе №3

на тему

ВЫЧИСЛЕНИЯ ПРИ ПОМОЩИ CUDA

Выполнили студенты группы 050503 Кириллов В.И.

Русель А.А.

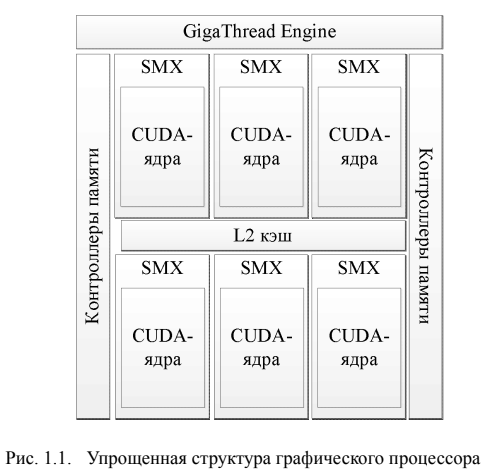
Преподаватель Русакович А.В.

МИНСК 2023

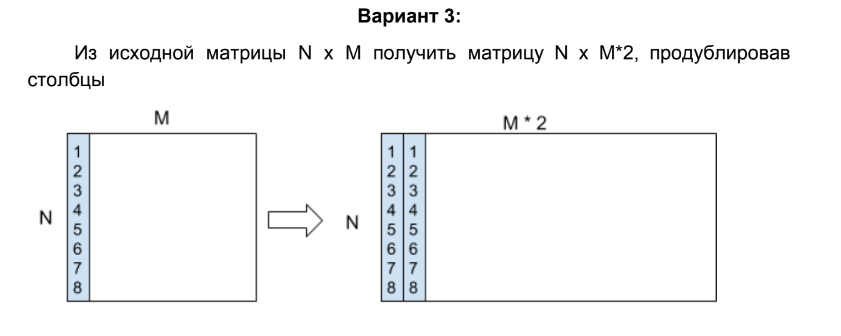
**ВВЕДЕНИЕ**

CUDA (Compute Unified Device Architecture) [1] – программно-аппаратная архитектура массового параллелизма для графических процессоров NVIDIA©. Данная архитектура обладает одной из наиболее успешных и простых моделей массового параллелизма, необходимого для эффективного программирования GPU.

В упрощенном виде вычислительный блок графического процессора представляет собой совокупность потоковых мультипроцессоров (streaming multiprocessor, SMX), управляемые с помощью GigaThread Engine. Каждый SMX состоит из множества вычислительных CUDA-ядер (рисунок 1).



**Задание:**



1. **СРЕДА РАЗРАБОТКИ И ПАРАМЕТРЫ СБОРКИ**

Конструирование лабораторной работы производилось на ОС Linux Manjaro 5.6.1

Язык: С++

Компилятор: nvcc

1. **РАЗРАБОТАННЫЕ ФУНКЦИИ (С, С++)**

#include <iostream>

#include <chrono>

#include <iomanip>

#include <curand.h>

using namespace std;

using namespace std::chrono;

using DataType = float;

constexpr size\_t M = 7485;

constexpr size\_t N = 9889;

DataType h\_inputMatrix[N][M];

DataType h\_outputMatrix[N][2 \* M];

DataType cpu\_outputMatrix[N][2 \* M];

void AssertCudaSuccess(cudaError status, std::string\_view functionName) {

if (status != cudaError\_t::cudaSuccess) {

throw std::runtime\_error(std::string{functionName} + " error");

}

}

void AssertCurandSuccess(curandStatus status, std::string\_view functionName) {

if (status != curandStatus::CURAND\_STATUS\_SUCCESS) {

throw std::runtime\_error(std::string{functionName} + " error");

}

}

void CudaFillRandom(DataType \*matrix) {

curandGenerator\_t generator;

AssertCurandSuccess(curandCreateGenerator(

&generator, CURAND\_RNG\_PSEUDO\_DEFAULT),

"curandCreateGenerator");

AssertCurandSuccess(curandSetPseudoRandomGeneratorSeed(generator, 1234ULL),

"curandSetPseudoRandomGeneratorSeed");

AssertCurandSuccess(curandGenerateUniform(generator, matrix, N \* M),

"curandGenerateUniform");

}

void PrintDeviceProperties() {

int device;

AssertCudaSuccess(cudaGetDevice(&device),

"cudaGetDevice");

cudaDeviceProp props{};

AssertCudaSuccess(cudaGetDeviceProperties(&props, device),

"cudaGetDeviceProperties");

std::cout << "totalGlobalMem : " << props.totalGlobalMem / 1024 / 1024 << " mb\n";

}

\_\_global\_\_ void KernelDoubleRows(const DataType \*inputMatrix,

DataType \*outputMatrix,

int rows,

int columns) {

auto row = blockIdx.x \* blockDim.x + threadIdx.x;

if (row < rows) {

for (int col = 0; col < columns; col++) {

auto baseRow = row \* columns;

outputMatrix[2 \* (baseRow + col)] = inputMatrix[baseRow + col];

outputMatrix[2 \* (baseRow + col) + 1] = inputMatrix[baseRow + col];

}

}

}

void CpuDoubleRows(const DataType input\_matrix[N][M],

DataType output\_matrix[N][2 \* M]) {

auto start = high\_resolution\_clock::now();

for (int row = 0; row < N; row++) {

for (int col = 0; col < M; col++) {

output\_matrix[row][2 \* col] = input\_matrix[row][col];

output\_matrix[row][2 \* col + 1] = input\_matrix[row][col];

}

}

std::cout << "CPU time: " << std::setprecision(6) <<

duration\_cast<milliseconds>(high\_resolution\_clock::now() - start).count()

<< " ms\n";

}

void GpuDoubleRows(DataType \*inputMatrix, DataType \*outputMatrix) {

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

dim3 threadsPerBlock(512);

dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x);

KernelDoubleRows<<<numBlocks, threadsPerBlock>>>(

inputMatrix, outputMatrix, N, M);

cudaEventSynchronize(stop);

float milliseconds;

cudaEventElapsedTime(&milliseconds, start, stop);

std::cout << "GPU time: " << std::setprecision(6) << milliseconds << " ms\n";

}

bool CompareResults(const DataType cpuMatrix[N][2 \* M],

const DataType gpuMatrix[N][2 \* M]) {

for (int i = 0; i < N; i++) {

if (!std::equal(&cpuMatrix[i][0],

&cpuMatrix[i][2 \* M - 1],

&gpuMatrix[i][0])) {

return false;

}

}

return true;

}

int main() {

PrintDeviceProperties();

DataType \*d\_inputMatrix, \*d\_outputMatrix;

AssertCudaSuccess(cudaMalloc((void \*\*) &d\_inputMatrix,

N \* M \* sizeof(DataType)),

"cudaMalloc [dev input]");

CudaFillRandom(d\_inputMatrix);

AssertCudaSuccess(cudaMemcpy(h\_inputMatrix, d\_inputMatrix,

N \* M \* sizeof(int), cudaMemcpyDeviceToHost),

"cudaMemcpy [dev input]");

CpuDoubleRows(h\_inputMatrix, cpu\_outputMatrix);

AssertCudaSuccess(cudaMalloc((void \*\*) &d\_outputMatrix,

N \* 2 \* M \* sizeof(DataType)),

"cudaMalloc [dev output]");

AssertCudaSuccess(cudaMemcpy(d\_inputMatrix, h\_inputMatrix,

N \* M \* sizeof(int), cudaMemcpyHostToDevice),

"cudaMemcpy [dev input]");

GpuDoubleRows(d\_inputMatrix, d\_outputMatrix);

AssertCudaSuccess(cudaMemcpy(h\_outputMatrix,

d\_outputMatrix,

N \* 2 \* M \* sizeof(int),

cudaMemcpyDeviceToHost),

"cudaMemcpy [host output]");

std::cout << "\nResult matrices equal: " << std::boolalpha <<

CompareResults(cpu\_outputMatrix,

h\_outputMatrix) << "\n";

AssertCudaSuccess(cudaFree(d\_inputMatrix),

"cudaFree [dev input]");

AssertCudaSuccess(cudaFree(d\_outputMatrix),

"cudaFree [dev output]");

return 0;

}

**ВЫВОД**

Тут вывод самой проги нужен