

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

Институт информатики и кибернетики
Кафедра технической кибернетики

ЛАБОРАТОРНАЯ РАБОТА № 5

по курсу
Параллельное программирование

Группа 6409
Студент

(подпись)

Д.К. Чернышова

Преподаватель,
к.ф.-м.н.

(подпись)

В.Д. Зайцев

Самара 2023

ЗАДАНИЕ

Произвести запуск программы с использованием библиотеки CUBLAS для умножения квадратных матриц размерностей $N \times N$, сгенерированных случайно. В ходе анализа работы программы оценить время ее выполнения на различном количестве исполняющих нитей.

Реализовать последовательный вариант программы. Оценить время ее выполнения. Рассчитать ускорение параллельных программ относительно последовательного варианта.

Таблица 1 – Исходные данные на ЛР № 5

Тип	float
N	(950, 1900, 3800)
Параметры транспонирования	(без T, с T)

ВВЕДЕНИЕ

Все чаще современные программные вычисления требуют значительного сокращения времени выполнения операций при работе. Организация программных вычислений на основе подходов параллельного программирования позволяет значительно выиграть в быстродействии и времени исполнения кода [1].

Матричное умножение – это фундаментальные операции, используемые в различных областях, таких как наука о данных, вычислительная наука и машинное обучение, поэтому разработка и применение методов усовершенствования данной операции является актуальной на данный момент.

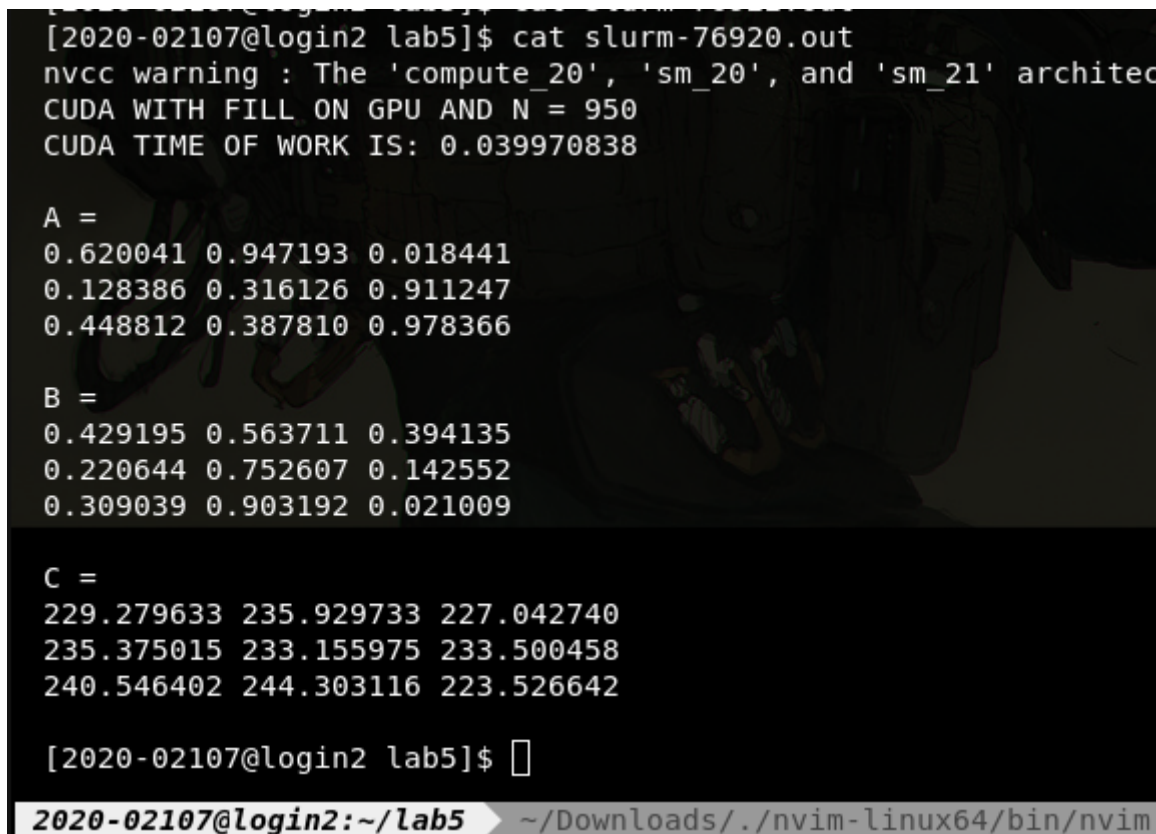
Поскольку матричное умножение является ресурсозатратной операцией, прибегают к использованию параллельных алгоритмов для обеспечения наибольшей эффективности и скорости.

ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ

2.1 Результаты работы программ

В ходе исследования времени работы программ здесь и далее проводилось усреднение не менее чем по 12 запускам.

На рисунке 1 представлен скрин запуска и работы программы.



```
[2020-02107@login2 lab5]$ cat slurm-76920.out
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are disabled because the GPU architecture does not support compilation at that level; use '--generate-code arch=compute_20,code=sm_20,code=sm_21' to enable them.
CUDA WITH FILL ON GPU AND N = 950
CUDA TIME OF WORK IS: 0.039970838

A =
0.620041 0.947193 0.018441
0.128386 0.316126 0.911247
0.448812 0.387810 0.978366

B =
0.429195 0.563711 0.394135
0.220644 0.752607 0.142552
0.309039 0.903192 0.021009

C =
229.279633 235.929733 227.042740
235.375015 233.155975 233.500458
240.546402 244.303116 223.526642

[2020-02107@login2 lab5]$
```

Рисунок 1 – Пример работы программы для $N = 950$.

Последовательная программа представляла собой перемножение двух матриц (без и с транспонирование) посредством вычисления каждого элемента результирующей матрицы.

В таблицах 2-3 представлено время выполнения параллельных программ и их ускорение по сравнению с последовательным вариантом.

Таблица 2 – Время работы программ

N	Время последовательной программы, с	Время параллельной программы, с	
		Генерация на GPU	Генерация CPU
950	8,99	0,03997	0,05383
1900	53,26750	0,06608	0,19329

3800	472,27583	0,23972	0,77432
------	-----------	---------	---------

Таблица 3 – Ускорение параллельных программ

N	Ускорение параллельной программы	
	Генерация на GPU	Генерация CPU
950	172,9295449	129,341519
1900	806,1044538	275,570177
3800	1970,150182	609,9183493

На рисунке 2 приведен график зависимости времени работы программ от размерности матрицы при различных вариантах генерации данных. На рисунке 3 приведен график зависимости ускорения программ от размерности матрицы при различных вариантах генерации данных.

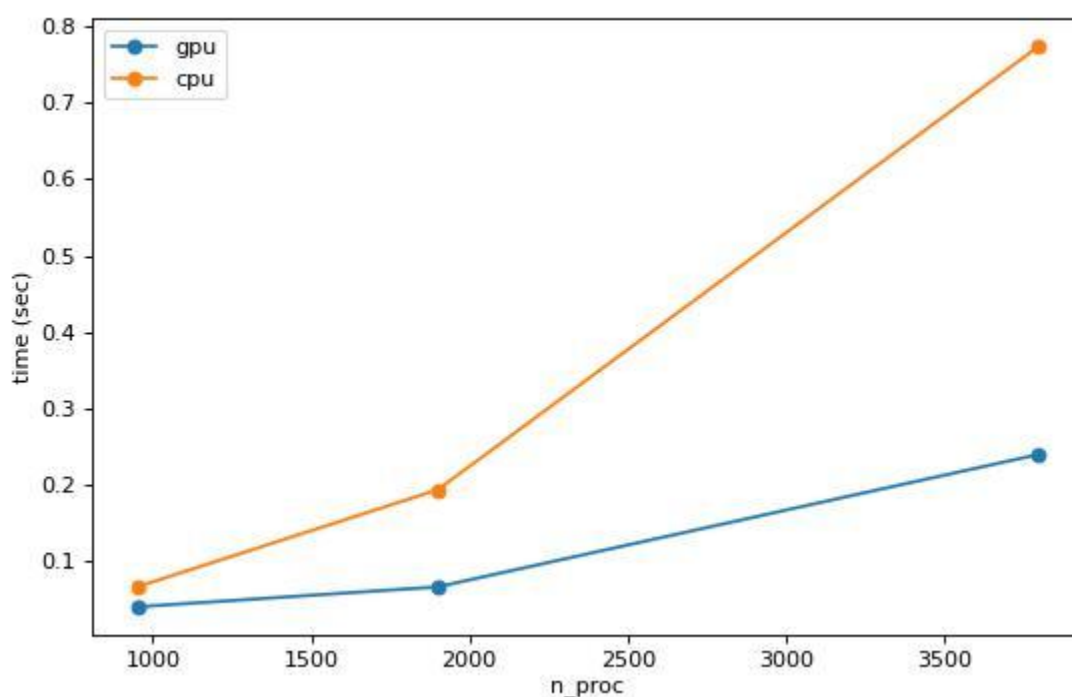


Рисунок 2 – Время работы программ

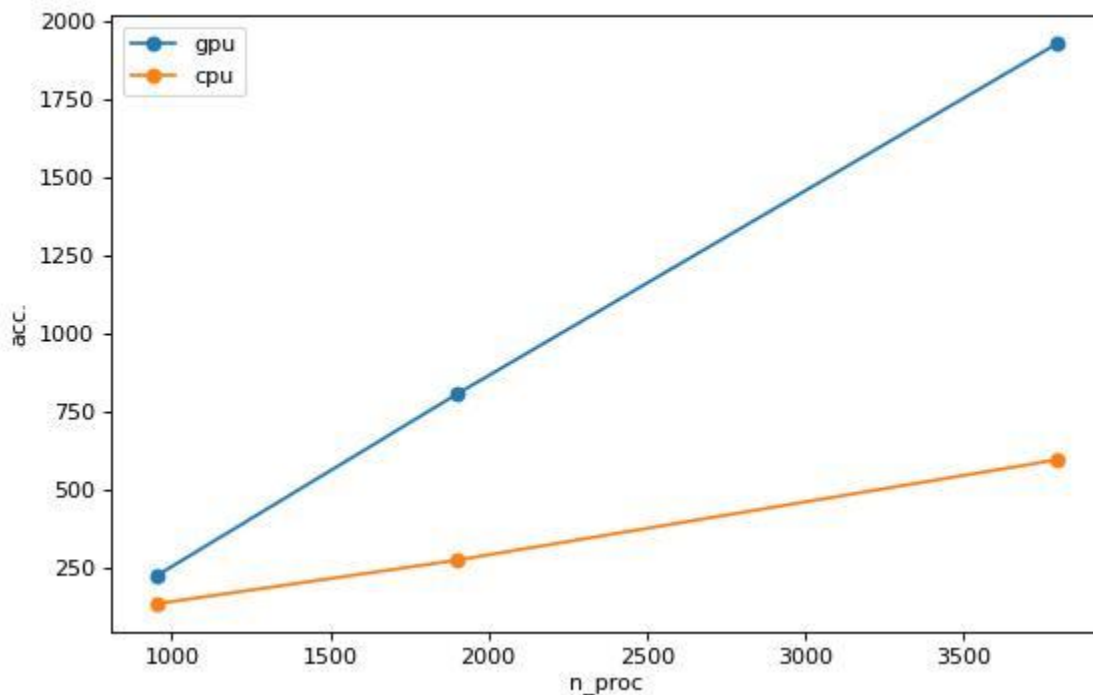


Рисунок 3 – Ускорение программ

ВЫВОДЫ:

Из полученных результатов видно, что:

1. Как видно из таблицы при всех значениях алгоритм с генерацией матрицы на CPU выполняется медленнее, чем с генерацией на GPU. Генерация матрицы на GPU тратит ресурсы на создание генератора случайных чисел и на копирование данных с GPU на CPU, однако затраты на пересылку и создание генератора являются незначительными для генерации матриц небольших размеров, например, 950x950.
2. Максимальное ускорение было получено для генерации на GPU при размерности матриц 3800x3800 – 1970,150182. Замедление получено не было, ввиду больших размеров матриц. Большое ускорение обуславливается тем, что для последовательных программ учитывалось время на генерацию и умножение матриц, а умножение матриц на GPU является на порядки более эффективной операцией, чем в последовательной программе.

3. С увеличением размерности соответственно увеличивается время на выполнение программы и ускорение по сравнению с последовательной программой. С изменением размерности время выполнения программы изменяется некратно ввиду того, что происходят затраты времени на копирование данных, а также, в случае заполнения на GPU, на создание генератора псевдослучайных чисел и из раза в раз время, затрачиваемое на такие операции, изменяется.

ЗАКЛЮЧЕНИЕ

Цель лабораторной работы – написать параллельную программу с использованием библиотеки CUBLAS для умножения квадратных матриц и сравнить время выполнения с длительностью последовательной программы достигнута. Показано, что использование параллельных технологий для данного типа программ обосновано, ввиду того, что при перемножение больших матриц использование GPU дает значимое ускорение.

В ходе выполнения лабораторной работы я изучил(а) основы использования библиотеки CUBLAS, приобрел(а) навыки по написанию параллельных программ с ее использованием. Наиболее сложной частью выполнения лабораторной работы было написание алгоритма для последовательного перемножения матриц с транспонированием в линейной форме. Интерес вызвало изучение операций на GPU с использованием CUBLAS.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 Kim, D. Analysis of Sub-Routines in NVIDIA cuBLAS Library for a series of Matrix-Matrix Multiplications in Transformer [Electronic resource] / D. Kim, I. Kim, J. Kim. // 13th International Conference on Information and Communication Technology Convergence (ICTC) – 2022. – P. 618-620 – URL: <https://ieeexplore.ieee.org/document/9952498> (дата обращения: 14.11.2023).
- 2 Козлова, Е.С. Лабораторные работы по курсу «Параллельное программирование»: Методические указания [Текст] / Сост. Е.С. Козлова, А.С. Широканев – Самара, 2019. – 61 с.
- 3 Воеводин, В. В. Параллельные вычисления [Текст] / В. В. Воеводин, Вл. В. Воеводин. – СПб.: БХВ-Петербург, 2002. – 608 с.
- 4 Богачёв К.Ю. Основы параллельного программирования: учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв – М.: БИНОМ. Лаборатория знаний, 2013. – 344 с.
- 5 Гергель, В. П. Теория и практика параллельных вычислений, 2-е изд. [Текст] / В. П. Гергель. – М.: Интуит. 2016. - 500 с.
- 6 Боресков А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA Учеб. пособие [Текст] / А.В. Боресков – М.: Издательство Московского университета, 2012. – 336 с.
- 7 Библиографическое описание документа. Общие требования и правила составления [Электронный ресурс] / сост.: В.С. Крылова, С.М. Григорьевская, Е.Ю. Кичигина // Официальный интернет-сайт научной библиотеки Томского государственного университета. – Электрон. дан. – Томск, [2010]. – <http://www.lib.tsu.ru/win/produkcija/metodichka/metodich.html> (дата обращения: 10.09.2019).

ПРИЛОЖЕНИЕ А

Код программы с технологией CUDA

```
#include <cstdlib>
#include <curand.h>
#include <cublas_v2.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// GPU_fill_rand() - Функция случайной генерации матрицы
void GPU_fill_rand(float* A, int matrixSize) {
    // Create a pseudo-random number generator
    curandGenerator_t prng;
    curandCreateGenerator(&prng, CURAND_RNG_PSEUDO_DEFAULT);
    // Set the seed for the random number generator using the system clock
    curandSetPseudoRandomGeneratorSeed(prng, (unsigned long long) clock());
    curandGenerateUniform(prng, A, matrixSize * matrixSize);
}

//gpu_blas_mmul() - Функция умножения матриц
void gpu_blas_mmul(const float* A, const float* B, float* C, const int
matrixSize) {
    int lda = matrixSize, ldb = matrixSize, ldc = matrixSize;
    const float alf = 1;
    const float bet = 0;
    const float* alpha = &alf;
    const float* beta = &bet;
    // Create a handle for CUBLAS
    cublasHandle_t handle;
    cublasCreate(&handle);
    // Do the actual multiplication
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, matrixSize, matrixSize,
matrixSize, alpha, A, lda, B, ldb, beta, C, ldc);
    // Destroy the handle
    cublasDestroy(handle);
}

void print_matrix(float* matrix, int matrixSize) {
    for (int i = 0; i < matrixSize; ++i) {
        for (int j = 0; j < matrixSize; ++j) {
            printf("%f ", matrix[j * matrixSize + i]);
        }
        printf("\n");
    }
    printf("\n");
}

int main() {
    // Allocate 3 arrays on CPU
    // for simplicity we are going to use square arrays
```

```

int matrixSize = NMAX, n2b = matrixSize * matrixSize * sizeof(float);
float* h_A = (float*)malloc(n2b);
float* h_B = (float*)malloc(n2b);
float* h_C = (float*)malloc(n2b);
// Allocate 3 arrays on GPU
float* d_A, * d_B, * d_C;
cudaMalloc(&d_A, n2b);
cudaMalloc(&d_B, n2b);
cudaMalloc(&d_C, n2b);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

float gpuTime = 0.0f;

srand(time(0));

cudaEventRecord(start, 0);

for (int i = 0; i < ITERATIONS; ++i) {
    if (isGPU) {
        // Fill the arrays A and B on GPU with random numbers
        GPU_fill_rand(d_A, matrixSize);
        GPU_fill_rand(d_B, matrixSize);
        // Optionally we can copy the data back on CPU and print the
arrays
        cudaMemcpy(h_A, d_A, n2b, cudaMemcpyDeviceToHost);
        cudaMemcpy(h_B, d_B, n2b, cudaMemcpyDeviceToHost);
        // Multiply A and B on GPU
        gpu_blas_mmul(d_A, d_B, d_C, matrixSize);
        // Copy (and print) the result on host memory
        cudaMemcpy(h_C, d_C, n2b, cudaMemcpyDeviceToHost);
    }
    else {
        for (int i = 0; i < matrixSize * matrixSize; i++) {
            h_A[i] = (float)rand() / RAND_MAX;
            h_B[i] = (float)rand() / RAND_MAX;
        }
        cudaMemcpy(d_A, h_A, n2b, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, n2b, cudaMemcpyHostToDevice);
        // Multiply A and B on GPU
        gpu_blas_mmul(d_A, d_B, d_C, matrixSize);
        cudaMemcpy(h_C, d_C, n2b, cudaMemcpyDeviceToHost);
    }
}

cudaEventRecord(stop, 0);
cudaEventElapsedTime(&gpuTime, start, stop);

if (isGPU)
    printf("CUDA WITH FILL ON GPU AND N = %d\n", NMAX);
else
    printf("CUDA WITH FILL ON CPU AND N = %d\n", NMAX);
printf("CUDA TIME OF WORK IS: %.9f\n\n", gpuTime / 1000 / ITERATIONS);

```

```
    printf("A =\n");
    print_matrix(h_A, 3);
    printf("B =\n");
    print_matrix(h_B, 3);

    printf("C =\n");
    print_matrix(h_C, 3);

    //Free GPU memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free CPU memory
    free(h_A);
    free(h_B);
    free(h_C);
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Код последовательной программы

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define IDX2C(i,j,ld) (((j)*(ld))+(i))
// NMAX = [950, 1900, 3800]
#define NMAX 3800
#define ITERATIONS 12

float* transpose(const float* A, int matrixSize, int n2b) {
    float* new_array = (float*)malloc(n2b);
    for (int i = 0; i < matrixSize; ++i)
    {
        for (int j = 0; j < matrixSize; ++j)
        {
            // Index in the original matrix.
            int index1 = i * matrixSize + j;

            // Index in the transpose matrix.
            int index2 = j * matrixSize + i;

            new_array[index2] = A[index1];
        }
    }
    return new_array;
}

void cpu_mmul(const float* A, const float* B, float* C, int matrixSize, int
n2b) {
    float* B_T = transpose(B, matrixSize, n2b);
    for (int i = 0; i < matrixSize; ++i) {
        for (int j = 0; j < matrixSize; ++j) {
            C[IDX2C(i, j, matrixSize)] = 0.0;
            for (int r = 0; r < matrixSize; ++r) {
                C[IDX2C(i, j, matrixSize)] += A[IDX2C(i, r, matrixSize)] *
B_T[IDX2C(r, j, matrixSize)];
            }
        }
    }
}

// print_matrix() - Функция вывода матрицы
void print_matrix(float* matrix, int matrixSize) {
    for (int i = 0; i < matrixSize; ++i) {
        for (int j = 0; j < matrixSize; ++j) {
            printf("%f ", matrix[j * matrixSize + i]);
        }
    }
}
```

```

        printf("\n");
    }
    printf("\n");
}

int main() {
    // Allocate 3 arrays on CPU
    // for simplicity we are going to use square arrays
    srand(time(0));
    int matrixSize = NMAX, n2b = matrixSize * matrixSize * sizeof(float);
    float* h_A = (float*)malloc(n2b);
    float* h_B = (float*)malloc(n2b);
    float* h_C = (float*)malloc(n2b);

    double start_time, end_time;

    start_time = clock();
    for (int i = 0; i < ITERATIONS; ++i) {
        for (int i = 0; i < matrixSize * matrixSize; i++) {
            h_A[i] = (float)rand() / RAND_MAX;
            h_B[i] = (float)rand() / RAND_MAX;
        }

        // Multiply A and B
        cpu_mmul(h_A, h_B, h_C, matrixSize, n2b);
    }
    end_time = clock();

    printf("LINEAR TIME OF WORK WITH N = %d IS: %.9f\n\n", NMAX, (end_time -
start_time) / CLOCKS_PER_SEC / ITERATIONS);

    printf("A =\n");
    print_matrix(h_A, 3);
    printf("B =\n");
    print_matrix(h_B, 3);

    printf("C =\n");
    print_matrix(h_C, 3);

    // Free CPU memory
    free(h_A);
    free(h_B);
    free(h_C);
    return 0;
}

```

ПРИЛОЖЕНИЕ В

Скрипт запуска CUDA программы

```
#!/bin/bash

#SBATCH --job-name=test
#SBATCH --nodes=1
#SBATCH --gres=gpu
#SBATCH --time=00:10:00

module load cuda/8.0
nvcc -g -G -O0 -DNMAX=900 -DisGPU=true -DITERATIONS=12 -lcublas -lcurand -o
mainGPU.bin cuda.cu
./mainGPU.bin
```

ПРИЛОЖЕНИЕ В

Скрипт запуска CUDA программы

```
#!/bin/bash
#SBATCH --job-name=test
#SBATCH --time=04:00:00
#SBATCH --nodes=1
#SBATCH --mem=2gb

./out
```