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 6408

Студент Д.О. Абрамов

(*подпись*)

Преподаватель,

к.ф.-м.н. Е.С. Козлова

(*подпись*)

Самара 2022

ЗАДАНИЕ

**Произвести запуск программы с использованием библиотеки CUBLAS** для **умножения квадратных матриц размерностей N×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. Рассчитатье ускорение параллельных программ относительно последовательного варианта.**

Таблица 1 – Исходные данные на ЛР № 5

|  |  |
| --- | --- |
| Тип | float |
| N | 1500, 3000, 6000 |
| Параметры транспонирования | с T, с T |

ВВЕДЕНИЕ

Первопричиной создания компьютеров была необходимость быстрого проведения вычислительных работ, со временем запросы на данную деятельность только росли и распространялись, что привело к повсеместному использованию компьютерной техники для разного рода задач.

В настоящее время на любом домашнем компьютере стоит многоядерный процессор, который проводит вычисления параллельно. Параллельные вычисления – это использование нескольких или многих вычислительных устройств для одновременного выполнения разных частей одной программы. Основная цель параллельных вычислений – уменьшение времени решения задачи, создание ресурса параллелизма в процессах решения задач с целью достижения наибольшей эффективности использования многопроцессорной вычислительной техники [1].

Параллельные вычисления охватывают вопросы, касающиеся создания ресурсов параллелизма в процессах решения задач и обеспечения его эффективной реализации. Разделяют параллелизм на уровне задач, на уровне данных, алгоритмов. Параллелизм на уровне задач предполагает разбиение задач на подзадачи, которые реализуются одновременно. Параллелизм на уровне данных реализуется компилятором и заключается в замене множества однотипных операций одной операцией. Параллелизм на уровне алгоритмов предполагает замену последовательных алгоритмов некоторых вычислений на параллельные вычисления [2].

На практике часто имеют дело с последовательными программами, где распараллеливание не используется, но с его помощью можно ускорить работу программы. Например, параллельная программа для вычисления приближения к определённому интегралу, смысл в использовании свойства аддитивности интеграла и подсчёта большого количества меньших интегралов. Что приводит к ускорению за счёт параллелизма [3].

Для решения сложных прикладных задач, с большим объёмом вычислений, требуются всё большие вычислительные мощности, такие проблемы и решает создание параллельных вычислительных систем. Сложность подобных задач со временем только растёт, что требует постоянного развития данной отрасли [4].

Можно с уверенностью сказать, что проблемы обработки информации в параллельных вычислениях играют важную роль в современных тенденциях развития информационных технологий, что подтверждается большим количеством современных статей на данную тематику.

ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ

2.1 Результаты работы программ без параметра Q

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.

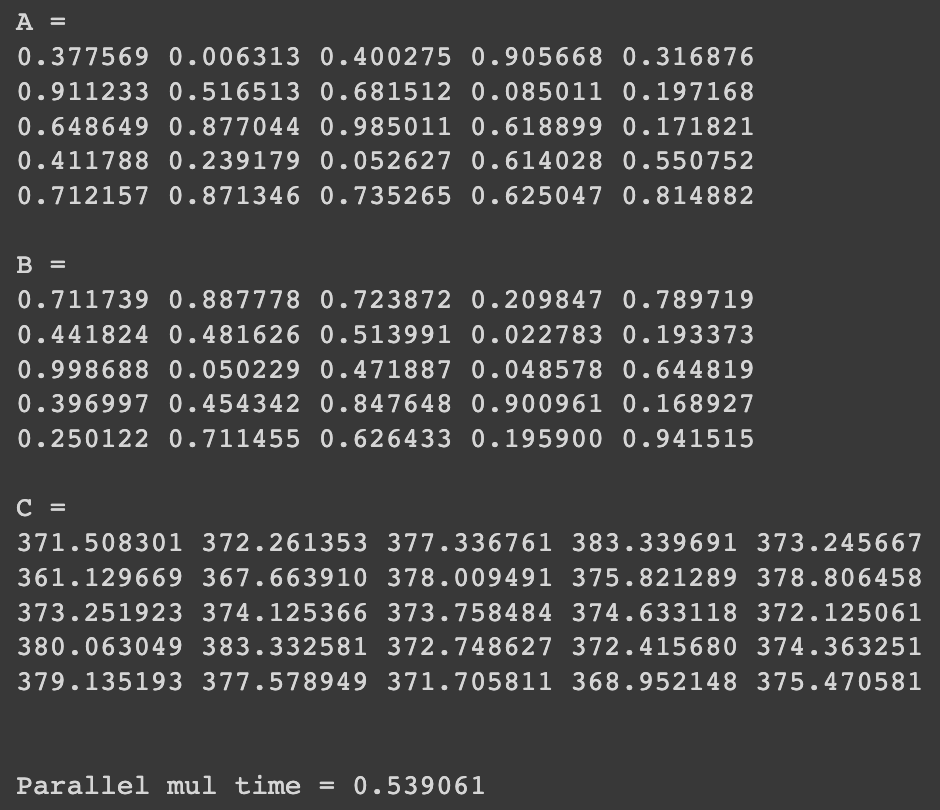


Рисунок 1 – Пример работы программы для N = 1500.

Последовательная программа представляла собой программу для подсчёта произведения двух матриц одним процессом/потоком.

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, мкс | Время параллельной программы, мкс | |
| Генерация на GPU | Генерация CPU |
| 1500 | 65546440 | 546051 | 639835 |
| 3000 | 68469630 | 564447 | 1045070 |
| 6000 | 4178046000 | 695054 | 2671820 |

Таблица 4 – Ускорение параллельных программ

|  |  |  |
| --- | --- | --- |
| N | Ускорение параллельной программы | |
| Генерация на GPU | Генерация CPU |
| 1500 | 102.44272351 | 120.037212641 |
| 3000 | 655.16788349 | 1213.03913388 |
| 6000 | 1563.7453122 | 6011.10992815 |

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.

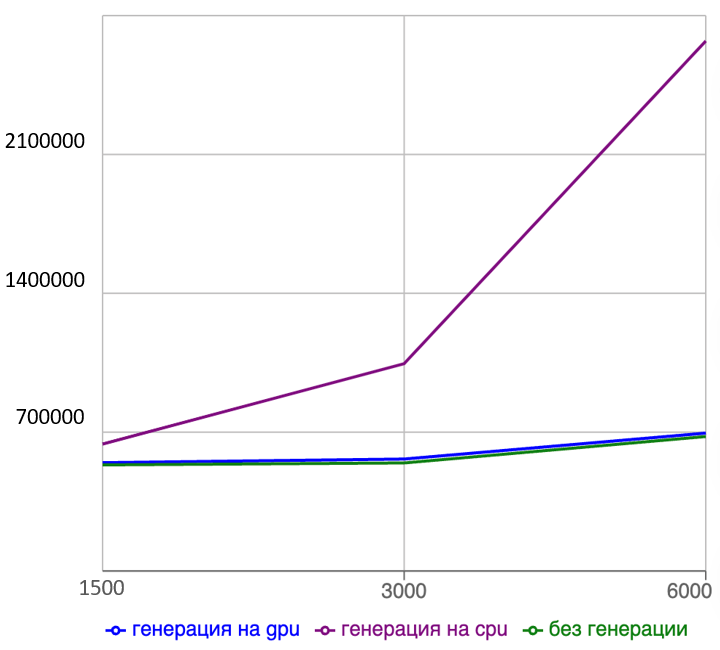


Рисунок 2 – Время работы программ

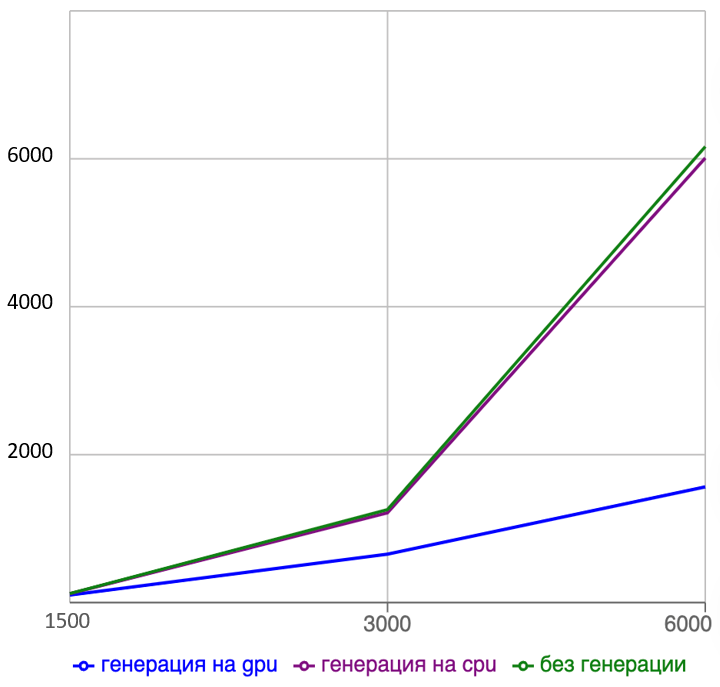


Рисунок 3 – Ускорение программ

**ВЫВОДЫ:**

Из полученных результатов видно, что:

1. При генерации на GPU время выполнения программы растёт с меньшими темпами, чем в случае генерации на CPU, графический процессор быстрее заполняет матрицы случайными данными, которые хранятся также на GPU.
2. Максимальное ускорение для GPU происходит при генерации данных на GPU, размерность N = 6000
3. В силу малых размерностей матриц, последовательная программа показала лучшие результаты, но с ростом размерности темпы ускорения растут экспоненциально. Это происходит из-за затрат на реализацию параллелизма библиотекой CUBLAS.

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, в виду того, что с ростом размерностей матриц, расчёт произведения матриц будет очень трудоёмким для последовательной программы, что решается использованием библиотеки CUBLAS.

В ходе выполнения лабораторной работы я изучил основы использования библиотеки CUBLAS, приобрел навыки по написанию параллельных программ с ее использованием. Наиболее сложной частью выполнения лабораторной работы было реализация генерации данных на GPU и произведения матриц с использованием функций библиотеки CUBLAS. Интерес вызвали возможности библиотеки CUBLAS.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Параллельные вычисления: Учебное пособие [Текст] / Ф.М. Гафаров, А.Ф. Галимянов. – Казань.: Казан. Ун-та, 2018. – 149 с.
2. Параллельные методы и алгоритмы: Учебное пособие [Электронный ресурс] / А.В. Волосова – М.: МАДИ, 2020. – 176 с. – URL.: https://lib.madi.ru/fel/fel1/fel20E533.pdf (дата обращения 18.09.2022).
3. Богачёв К.Ю. Основы параллельного программирования: Учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв – М.: БИНОМ. Лаборатория знаний, 2013. – 344 с.
4. Козлова, Е.С. Лабораторные работы по курсу «Параллельное программирование»: Методические указания [Текст] / Сост. Е.С. Козлова, А.С. Широканев − Самара, 2019. – 61 с.
5. оеводин, В. В. Параллельные вычисления [Текст] / В. В. Воеводин, Вл. В. Воеводин. — СПб.: БХВ-Петербург, 2002. — 608 с.
6. Богачёв К.Ю. Основы параллельного программирования: учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв ‑ М. : БИНОМ. Лаборатория знаний, 2013. ‑ 344 с.
7. Гергель, В. П. Теория и практика параллельных вычислений, 2-е изд. [Текст] / B. II. Гергель. — М.: Интуит. 2016. - 500 с.
8. Боресков А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDА Учеб. пособие [Текст] / А.В. Боресков ‑ М.: Издательство Московского университета, 2012. - 336 с.
9. 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/produkzija/metodichka/metodich.html> (дата обращения: 10.09.2019).

ПРИЛОЖЕНИЕ А

Код программы с технологией CUDA

%%cuda --name my\_curand.cu  
  
#include **<cstdlib>**#include **<curand.h>**#include **<cublas\_v2.h>**#include **<iostream>**#include **<ctime>  
  
using namespace** std;  
  
**void** multiply(**float**\* A, **float**\* B, **float**\* C, **int** N)  
{  
 **int** i, j, k;  
 **for** (i = 0; i < N; i++) {  
 **for** (j = 0; j < N; j++) {  
 C[i + j \* N] = 0;  
 **for** (k = 0; k < N; k++)  
 C[i + j \* N] += A[i + k \* N] \* B[k + j \* N];  
 }  
 }  
}  
  
**void** GPU\_fill\_rand(**float** \*A, **int** nr\_rows\_A, **int** nr\_cols\_A) {  
 curandGenerator\_t prng;  
 curandCreateGenerator(&prng, CURAND\_RNG\_PSEUDO\_DEFAULT);  
 curandSetPseudoRandomGeneratorSeed(prng, (**unsigned long long**) clock());  
 curandGenerateUniform(prng, A, nr\_rows\_A \* nr\_cols\_A);  
}  
  
*// C(m,n) = A(m,k) \* B(k,n)***void** gpu\_blas\_mmul(**const float** \*A, **const float** \*B, **float** \*C, **const int** m, **const int** k, **const int** n) {  
 **int** lda=m,ldb=k,ldc=m;  
 **const float** alf = 1;  
 **const float** bet = 0;  
 **const float** \*alpha = &alf;  
 **const float** \*beta = &bet;  
 cublasHandle\_t handle;  
 cublasCreate(&handle);  
 cublasSgemm(handle, CUBLAS\_OP\_N, CUBLAS\_OP\_N, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc);  
 cublasDestroy(handle);  
}  
  
**void** print\_matrix(**float**\* matrix, **int** rows, **int** cols) {  
 **for**(**int** i = 0; i < rows; ++i){  
 **for**(**int** j = 0; j < cols; ++j){  
 printf(**"%f "**, matrix[i + j \* rows]);  
 }  
 printf(**"\n"**);  
 }  
 printf(**"\n"**);  
}  
  
**void** fill\_matrix(**float**\* matrix, **int** rows, **int** cols) {  
 **for**(**int** i = 0; i < rows; ++i){  
 **for**(**int** j = 0; j < cols; ++j) {  
 matrix[i + j \* rows] = (rand() + 0.0) / **RAND\_MAX**;  
 }  
 }  
}  
  
  
**int** main() {  
 **int** nr\_rows\_A, nr\_cols\_A, nr\_rows\_B, nr\_cols\_B, nr\_rows\_C, nr\_cols\_C;  
 nr\_rows\_A = nr\_cols\_A = nr\_rows\_B = nr\_cols\_B = nr\_rows\_C = nr\_cols\_C = 6000;  
 **float** \*h\_A = (**float** \*)malloc(nr\_rows\_A \* nr\_cols\_A \* **sizeof**(**float**));  
 **float** \*h\_B = (**float** \*)malloc(nr\_rows\_B \* nr\_cols\_B \* **sizeof**(**float**));  
 **float** \*h\_C = (**float** \*)malloc(nr\_rows\_C \* nr\_cols\_C \* **sizeof**(**float**));  
 **float** \*h\_C\_ = (**float** \*)malloc(nr\_rows\_C \* nr\_cols\_C \* **sizeof**(**float**));  
  
 **float** \*d\_A, \*d\_B, \*d\_C;  
 cudaMalloc(&d\_A,nr\_rows\_A \* nr\_cols\_A \* **sizeof**(**float**));  
 cudaMalloc(&d\_B,nr\_rows\_B \* nr\_cols\_B \* **sizeof**(**float**));  
 cudaMalloc(&d\_C,nr\_rows\_C \* nr\_cols\_C \* **sizeof**(**float**));  
  
  
 GPU\_fill\_rand(d\_A, nr\_rows\_A, nr\_cols\_A);  
 GPU\_fill\_rand(d\_B, nr\_rows\_B, nr\_cols\_B);  
  
  
 //fill\_matrix(h\_A, nr\_rows\_A, nr\_cols\_A);  
 //fill\_matrix(h\_B, nr\_rows\_B, nr\_cols\_B);  
  
 **double** start = clock();  
 cudaMemcpy(d\_A, h\_A, nr\_rows\_A \* nr\_cols\_A \* **sizeof**(**float**), cudaMemcpyHostToDevice);  
 cudaMemcpy(d\_B, h\_B, nr\_rows\_B \* nr\_cols\_B \* **sizeof**(**float**), cudaMemcpyHostToDevice);  
 **double** end = clock();  
  
 cudaMemcpy(h\_A, d\_A, nr\_rows\_A \* nr\_cols\_A \* **sizeof**(**float**),cudaMemcpyDeviceToHost);  
 cudaMemcpy(h\_B, d\_B, nr\_rows\_B \* nr\_cols\_B \* **sizeof**(**float**),cudaMemcpyDeviceToHost);  
  
 printf(**"A = \n"**);  
 print\_matrix(h\_A, 5, 5);  
  
 printf(**"B = \n"**);  
 print\_matrix(h\_B, 5, 5);  
  
**double** start\_ = clock();  
 gpu\_blas\_mmul(d\_A, d\_B, d\_C, nr\_rows\_A, nr\_cols\_A, nr\_cols\_B);  
 *//multiply(h\_A, h\_B, h\_C\_, nr\_rows\_A);* **double** end\_ = clock();  
  
 cudaMemcpy(h\_C,d\_C,nr\_rows\_C \* nr\_cols\_C \* **sizeof**(**float**),cudaMemcpyDeviceToHost);  
 printf(**"C = \n"**);  
 print\_matrix(h\_C, 5, 5);  
  
 *cout << "\nParallel mul time = " << (end\_ - start\_) / CLOCKS\_PER\_SEC << endl;*  
 cudaFree(d\_A);  
 cudaFree(d\_B);  
 cudaFree(d\_C);  
  
 free(h\_A);  
 free(h\_B);  
 free(h\_C);  
 **return** 0;  
}

ПРИЛОЖЕНИЕ Б

Код последовательной программы

#include **<iostream>  
  
using namespace** std;  
  
**void** printMatrix(**float** \*\*matrix, **int** rows, **int** cols) {  
 **for** (**int** i = 0; i < rows; ++i) {  
 **for** (**int** j = 0; j < cols; ++j) {  
 printf(**"%f "**, matrix[i][j]);  
 }  
 printf(**"\n"**);  
 }  
 printf(**"\n"**);  
}  
  
**void** transpose(**float** \*matrix[], **int** nrCols) {  
 **float** t;  
 **for** (**int** i = 0; i < nrCols; ++i) {  
 **for** (**int** j = i; j < nrCols; ++j) {  
 t = matrix[i][j];  
 matrix[i][j] = matrix[j][i];  
 matrix[j][i] = t;  
 }  
 }  
}  
  
**int** main() {  
 **double** fullTime = 0, fullMullTime = 0, genTime = 0;  
 **int** numRowsA, numColsA, numRowsB, numColsB, numRowsC, numColsC;  
*// for simplicity we are going to use square arrays* numRowsA = numColsA = numRowsB = numColsB = numRowsC = numColsC = 1500;  
 **float** \*\*matrixA = **new float** \*[numColsA];  
 **float** \*\*matrixB = **new float** \*[numColsB];  
 **float** \*\*matrixC = **new float** \*[numColsC];  
  
 **for** (**int** i = 0; i < numColsA; i++) {  
 matrixA[i] = **new float**[numColsA];  
 matrixB[i] = **new float**[numColsA];  
 matrixC[i] = **new float**[numColsA];  
 }  
  
 **for** (**int** i = 0; i < 12; i++) {  
 genTime = clock();  
  
 *// Fill the arrays A and B with random numbers* **for** (**int** i = 0; i < numColsA; ++i) {  
 **for** (**int** j = 0; j < numColsA; ++j) {  
 matrixA[i][j] = (**float**) rand() / **RAND\_MAX**;  
 }  
 }  
 **for** (**int** i = 0; i < numColsB; ++i) {  
 **for** (**int** j = 0; j < numColsB; ++j) {  
 matrixB[i][j] = (**float**) rand() / **RAND\_MAX**;  
 }  
 }  
 genTime = clock() - genTime;  
 fullTime += genTime;  
 }  
  
 printf(**"Matrix A = \n"**);  
 printMatrix(matrixA, 5, 5);  
 printf(**"Matrix B = \n"**);  
 printMatrix(matrixB, 5, 5);  
 cout << **"\nAverage CPU Gen Time = "** << (fullTime) / (12 \* **CLOCKS\_PER\_SEC**) << endl;  
  
 **double** start = 0;  
 **double** stop = 0;  
 **double** multTime = 0;  
 start = clock();  
 transpose(matrixA, numColsA);  
 transpose(matrixB, numColsB);  
 **for** (**int** i = 0; i < numColsA; i++) {  
 **for** (**int** j = 0; j < numColsA; j++) {  
 matrixC[i][j] = 0;  
 **for** (**int** k = 0; k < numColsA; k++)  
 matrixC[i][j] += matrixA[i][k] \* matrixB[k][j];  
 }  
 }  
 stop = clock() - start;  
 printf(**"Matrix C = \n"**);  
 printMatrix(matrixC, 5, 5);  
 cout << **"\nAverage Mult Time = "** << (stop) / (**CLOCKS\_PER\_SEC**) << endl;  
 cout << **"\nAverage Full Time = "** << (stop + fullTime / 12) / (**CLOCKS\_PER\_SEC**) << endl;  
}