Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет

телекоммуникаций и информатики»

Кафедра ПМиК

Расчетно-графическое задание по дисциплине

Программирование графических процессоров

Сравнительный анализ алгоритмов линейной алгебры

Выполнил: студент 3 курса

Ф. ИВТ, группа: ИП-711

Мартасов И. О.

Проверил: профессор кафедры ПМиК

Малков Евгений Александрович

Новосибирск, 2020

**Содержание**

1.Постановка задачи …...........................................................................................3

2.Используемый инструментарий и рассматриваемые алгоритмы....................4

3.Сравнительные диаграммы.................................................................................8

3.1. SAXPY………………………………………………………………….8

3.2. SGEMV…………………………………………………………………9

3.3. SGEMM……………………………………………………………….10

4.Выводы…………………………………………………………………………11

5.Приложение. Код программы..………..………………………………………12

**Постановка задачи**

Провести сравнительный анализ производительности программ, реализующих алгоритмы линейной алгебры с использованием библиотек Thrust, СUBLAS и “сырого” CUDA C кода а также при использовании программ на разных видеокартах – GEFORCE GTX 1050(серверная видеокарта) и GEFORE 840M (личная видеокарта).

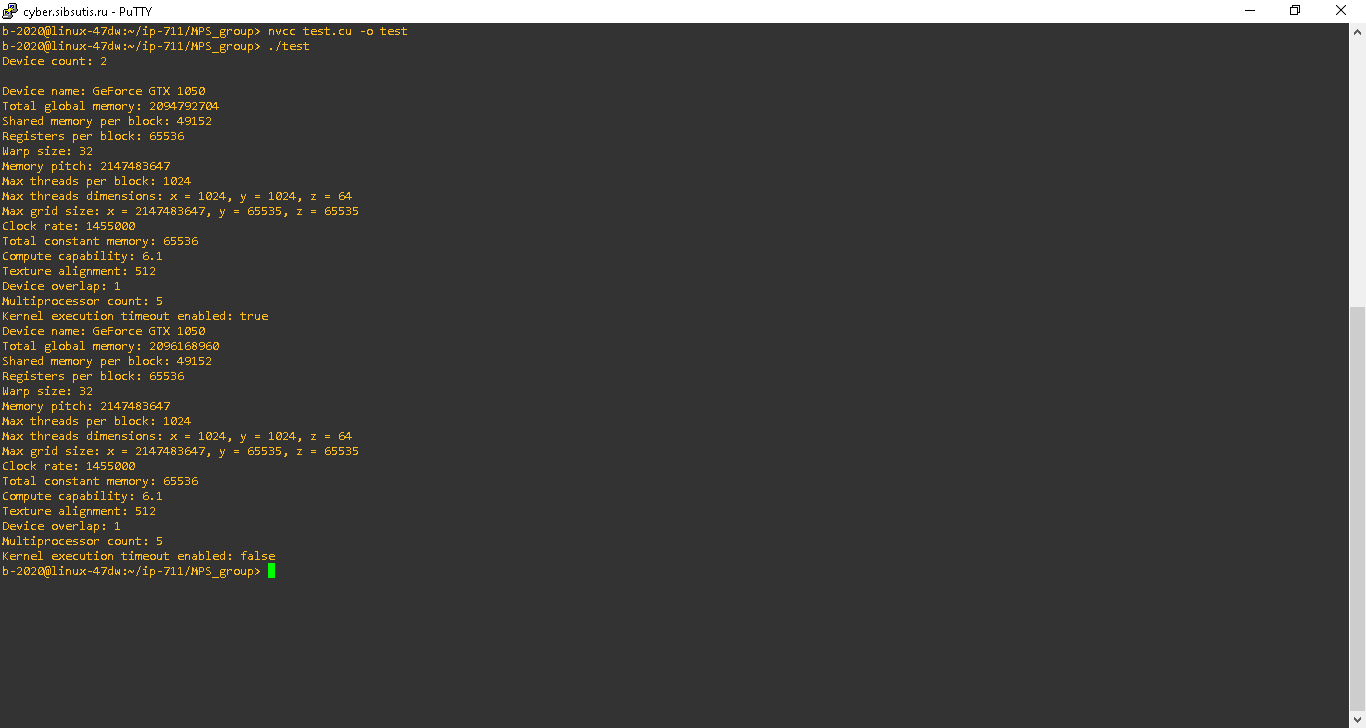
**Используемый инструментарий и алгоритмы**

В качестве используемого инструментария использовались следующие видеокарты:

* Серверная видеокарта

|  |  |  |
| --- | --- | --- |
|  | **GeForce GTX 1050** |  |
| Кодовое имя чипа | GP107 |  |  |
| Архитектура | Pascal |  |  |
| Техпроцесс | 14 нм |  |  |
| Количество транзисторов | 3,3 млрд. |  |  |
| Тактовая частота, МГц: базовая/[Boost](https://gtx-force.ru/nvidia-gpu-boost-2-0/) | 1354/1455 |  |  |
| Текстурных блоков (TMU) | 40 |  |  |
| Универсальных процессоров | 640 |  |  |
| Блоков растеризации (ROP) | 32 |  |  |
| Тип видеопамяти | GDDR5 |  |  |
| Объем видеопамяти | 2 ГБ |  |  |
| Разрядность шины, бит | 128 |  |  |
| Частота видеопамяти | 1750 МГц |  |  |
| Интерфейс | PCI-Express 3.0 |  |  |
| Потребляемая мощность | 75 Вт |  |  |
| Максимальная температура, °С | 97 |  |  |

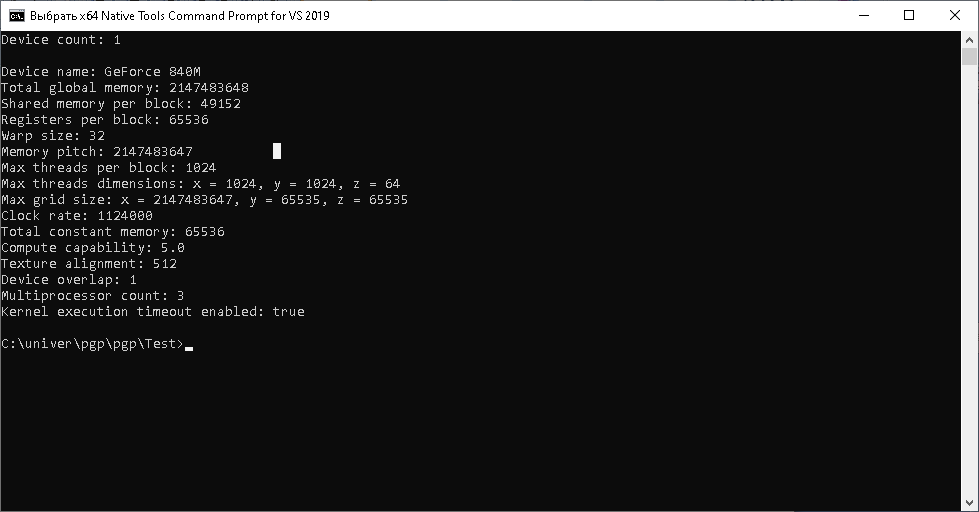
Характеристики, полученные с помощью cudaGetDeviceProperties:



* Личная видеокарта

|  |  |
| --- | --- |
| Серия | GeForce 800M |
| Архитектура | Maxwell |
| Количество шейдеров | 384 |
| Тактовая частота | 1029 МГц + boost |
| Частота памяти | 2000 MГц |
| Тип памяти | DDR3 |
| Разрядность шины памяти | 64 бит |
| Максимум видеопамяти | 4096 Мб |
| Дата выхода | 12 марта 2014 года |
|  |  |
|  |  |

Характеристики, полученные с помощью cudaGetDeviceProperties:



Рассматривались следующие алгоритмы линейной алгебры:

* SAXPY – перемножение двух векторов (размерность до 1 << 27)
* SGEMV – умножение матрицы на вектор (размерность до 1 << 14)
* SGEMM - матричное умножение (размерность до 1 << 10)

Измерения производились путем увеличения размерности массива (матрицы) на степень двойки.

В качестве итогового результата выбиралось среднее арифметическое после 10 измерений для каждой размерности векторов (матриц).

При реализации на языке CUDA использовались чистые измерения – без использования разделяемой памяти.

Измеряемые данные имели тип float.

Время работы программы измерялось в наносекундах.

**Сравнительные диаграммы**

1. **SAXPY**

**2.SGEMV**

**3. SGEMM**

**Выводы**

Из результатов видно, что время выполнения SAXPY на личном компьютере не зависит от реализации алгоритма. В данном случае оказалось, что производительность всех реализаций практически сопоставима, а разрыв между ними незначителен.

Однако на серверной видеокарте реализация алгоритма на языке CUDA в сравнении с другими реализациями оказалась чуть менее производительной – это стало заметно при размере данных начиная с 2^20 и выше.

В реализации SGEMV и SGEMM невооруженным глазом заметно сильное проседание в производительности у библиотеки Thrust что у личной, что у серверной видеокарты. При этом для SGEMV быстрее оказалась реализация CUBLAS, а для SGEMM – CUDA, причем для обеих видеокарт.

Если сравнивать общую производительность между реализациями алгоритмов на разных видеокартах, то наиболее производительной оказалась реализация CUBLAS, после чего идет CUDA, а затем Thrust.

Если же сравнивать общую производительность между видеокартами, то выигрывает серверная видеокарта, несмотря на практически идентичные характеристики с видеокартой на личном компьютере, полученные cudaGetDeviceProperties.

**Приложение. Код программы**

**1.SAXPY**

#pragma comment (lib, "cublas.lib")

#include <stdio.h>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <iostream>

#include <ctime>

#include <thrust/device\_vector.h>

#include <thrust/fill.h>

#include <thrust/host\_vector.h>

#include <thrust/sequence.h>

#include <thrust/transform.h>

#include <cublas.h>

#include <cublas\_v2.h>

using namespace std;

const int N = 1 << 26;

#define V 0.2

#define T 2

#define CUDA\_CHECK\_RETURN(value) ((cudaError\_t)value != cudaSuccess) ? printf("Error %s at line %d in the file %s\n", cudaGetErrorString((cudaError\_t)value), \_\_LINE\_\_, \_\_FILE\_\_) : printf("")

\_\_global\_\_ void saxpyblas(float \*x, float \*y, float \*z)

{

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

z[i] = x[i] \* (V \* T) + y[i];

}

struct functor

{

const float koef;

functor(float \_koef) : koef(\_koef) {}

\_\_host\_\_ \_\_device\_\_ float operator()(float x, float y)

{

return koef \* x + y;

}

};

void saxpy(float \_koef, thrust::device\_vector<float> &x, thrust::device\_vector<float> &y)

{

functor func(\_koef);

thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), func);

}

int main()

{

srand(time(NULL));

cudaEvent\_t start, stop;

float time;

float alpha = V \* T;

float max[27];

int k = 0;

for(int i = 0; i < 27; i++)

{

max[i] = 0;

}

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

cublasHandle\_t cublas\_handle;

CUDA\_CHECK\_RETURN(cublasCreate(&cublas\_handle));

cudaDeviceProp deviceProp;

CUDA\_CHECK\_RETURN(cudaGetDeviceProperties(&deviceProp, 0));

int blocks = 0;

int threads = 0;

int count = 1;

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

float \*x;

float \*y;

float \*z;

float \*x1;

float \*y1;

float \*z1;

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&x1, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&y1, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&z1, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&x, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&y, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&z, count \* sizeof(float)));

for(int i = 0; i < count; i++)

{

x[i] = rand() % 1000;

y[i] = rand() % 1000;

}

CUDA\_CHECK\_RETURN(cudaMemcpy(x1, x, count \* sizeof(float), cudaMemcpyHostToDevice));

CUDA\_CHECK\_RETURN(cudaMemcpy(y1, y, count \* sizeof(float), cudaMemcpyHostToDevice));

if(count <= deviceProp.maxThreadsPerBlock)

{

threads = count;

blocks = 1;

}

else

{

threads = deviceProp.maxThreadsPerBlock;

blocks = count / 1024;

}

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

saxpyblas <<< blocks, threads >>> (x1, y1, z1);

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

CUDA\_CHECK\_RETURN(cudaMemcpy(z, z1, count \* sizeof(float), cudaMemcpyDeviceToHost));

printf("%d - CUDA: %f ms\n", count, time);

max[k] += time;

CUDA\_CHECK\_RETURN(cudaFreeHost(x));

CUDA\_CHECK\_RETURN(cudaFreeHost(y));

CUDA\_CHECK\_RETURN(cudaFreeHost(z));

CUDA\_CHECK\_RETURN(cudaFree(x1));

CUDA\_CHECK\_RETURN(cudaFree(y1));

CUDA\_CHECK\_RETURN(cudaFree(z1));

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 27; i++)

{

cout << max[i] / 10 \* 1000 << endl;

max[i] = 0;

}

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

thrust::host\_vector<float> cpumem1(count);

thrust::host\_vector<float> cpumem2(count);

for (int i = 0; i < count; i++)

{

cpumem1[i] = rand() % 1000;

cpumem2[i] = rand() % 1000;

}

thrust::device\_vector<float> gpumem1 = cpumem1;

thrust::device\_vector<float> gpumem2 = cpumem2;

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

saxpy(V \* T, gpumem2, gpumem1);

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

printf("%d - Thrust: %f ms\n", count, time);

max[k] += time;

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 27; i++)

{

cout << max[i] / 10 \* 1000 << endl;

max[i] = 0;

}

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

float \*host\_x;

float \*host\_y;

float \*dev\_x;

float \*dev\_y;

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&dev\_x, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&dev\_y, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&host\_x, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&host\_y, count \* sizeof(float)));

for (int i = 0; i < count; i++)

{

host\_x[i] = rand() % 1000;

host\_y[i] = rand() % 1000;

}

CUDA\_CHECK\_RETURN(cublasSetVector(count, sizeof(float), host\_x, 1, dev\_x, 1));

CUDA\_CHECK\_RETURN(cublasSetVector(count, sizeof(float), host\_y, 1, dev\_y, 1));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

CUDA\_CHECK\_RETURN(cublasSaxpy(cublas\_handle, count, &alpha, dev\_x, 1, dev\_y, 1));

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

CUDA\_CHECK\_RETURN(cublasGetVector(count, sizeof(float), dev\_x, 1, host\_x, 1));

CUDA\_CHECK\_RETURN(cublasGetVector(count, sizeof(float), dev\_y, 1, host\_y, 1));

printf("%d - cuBLAS: %f ms\n", count, time);

max[k] += time;

CUDA\_CHECK\_RETURN(cudaFreeHost(host\_x));

CUDA\_CHECK\_RETURN(cudaFreeHost(host\_y));

CUDA\_CHECK\_RETURN(cudaFree(dev\_x));

CUDA\_CHECK\_RETURN(cudaFree(dev\_y));

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 27; i++)

{

cout << max[i] / 10 \* 1000 << endl;

max[i] = 0;

}

CUDA\_CHECK\_RETURN(cublasDestroy(cublas\_handle));

return 0;

}

**2.SGEMV**

#pragma comment (lib, "cublas.lib")

#include <stdio.h>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <iostream>

#include <ctime>

#include <thrust/device\_vector.h>

#include <thrust/fill.h>

#include <thrust/host\_vector.h>

#include <thrust/sequence.h>

#include <thrust/transform.h>

#include <cublas.h>

#include <cublas\_v2.h>

using namespace std;

const int N = 1 << 14;

#define V 0.2

#define T 2

#define CUDA\_CHECK\_RETURN(value) ((cudaError\_t)value != cudaSuccess) ? printf("Error %s at line %d in the file %s\n", cudaGetErrorString((cudaError\_t)value), \_\_LINE\_\_, \_\_FILE\_\_) : printf("")

void output(float\* a, int n)

{

for(int i = 0; i < n; i++)

{

for(int j = 0; j < n; j++)

{

printf("%g ", a[i + j \* n]);

}

printf("\n");

}

printf("\n");

}

\_\_global\_\_ void sgemvblas(float \*a, float \*x, float \*y, int count)

{

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

if(i < count)

{

float res = 0;

for(int j = 0; j < count; j++)

{

res += a[i \* count + j] \* x[j];

}

y[i] = res;

}

}

struct dp

{

float \*A, \*B;

int count;

dp(float \*\_A, float \*\_B, int \_count): A(\_A), B(\_B), count(\_count){};

\_\_host\_\_ \_\_device\_\_

float operator()(size\_t idx)

{

float sum = 0.0f;

int row = idx / count;

int col = idx - (row \* count);

for (int i = 0; i < count; i++)

{

sum += A[col \* i + row] \* B[i];

}

return sum;

}

};

void sgemv(int count, thrust::device\_vector<float> &data, thrust::device\_vector<float> &other, thrust::device\_vector<float> &result)

{

thrust::transform(thrust::counting\_iterator<int>(0), thrust::counting\_iterator<int>(count), result.begin(), dp(thrust::raw\_pointer\_cast(data.data()), thrust::raw\_pointer\_cast(other.data()), count));

}

int main()

{

srand(time(NULL));

cudaEvent\_t start, stop;

float time = 0;

float alpha = 1;

float max[15];

for(int i = 0; i < 15; i++)

{

max[i] = 0;

}

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

cublasHandle\_t cublas\_handle;

CUDA\_CHECK\_RETURN(cublasCreate(&cublas\_handle));

int count = 1;

int k = 0;

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

float \*a;

float \*x;

float \*y;

float \*a\_dev;

float \*x\_dev;

float \*y\_dev;

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&a\_dev, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&x\_dev, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&y\_dev, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&a, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&x, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&y, count \* sizeof(float)));

for(int i = 0; i < count; i++)

{

x[i] = rand() % 1000;

y[i] = 0;

for(int j = 0; j < count; j++)

{

a[i \* count + j] = rand() % 1000;

}

}

CUDA\_CHECK\_RETURN(cudaMemcpy(a\_dev, a, count \* count \* sizeof(float), cudaMemcpyHostToDevice));

CUDA\_CHECK\_RETURN(cudaMemcpy(x\_dev, x, count \* sizeof(float), cudaMemcpyHostToDevice));

CUDA\_CHECK\_RETURN(cudaMemcpy(y\_dev, y, count \* sizeof(float), cudaMemcpyHostToDevice));

dim3 threadBlock(1024, 1);

dim3 blockGrid(count / 1024 + 1, 1, 1);

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

sgemvblas <<< blockGrid, threadBlock >>> (a\_dev, x\_dev, y\_dev, count);

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

CUDA\_CHECK\_RETURN(cudaMemcpy(y, y\_dev, count \* sizeof(float), cudaMemcpyDeviceToHost));

printf("%d - CUDA: %f ms\n", count, time);

max[k] += time;

CUDA\_CHECK\_RETURN(cudaFreeHost(a));

CUDA\_CHECK\_RETURN(cudaFreeHost(x));

CUDA\_CHECK\_RETURN(cudaFreeHost(y));

CUDA\_CHECK\_RETURN(cudaFree(a\_dev));

CUDA\_CHECK\_RETURN(cudaFree(x\_dev));

CUDA\_CHECK\_RETURN(cudaFree(y\_dev));

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 15; i++)

{

cout << max[i] / 10 << endl;

max[i] = 0;

}

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

thrust::device\_vector<float> data(count \* count, 2);

thrust::device\_vector<float> other(count, 5);

thrust::device\_vector<float> result(count, 0);

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

sgemv(count, data, other, result);

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

printf("%d - THRUST: %f ms\n", count, time);

max[k] += time;

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 15; i++)

{

cout << max[i] / 10 << endl;

max[i] = 0;

}

/////////////////////////////////////////////////////CUBLAS//////////////////////////////////////////////////////////////

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

float \*a;

float \*x;

float \*y;

float \*a\_dev;

float \*x\_dev;

float \*y\_dev;

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&a\_dev, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&x\_dev, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&y\_dev, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&a, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&x, count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&y, count \* sizeof(float)));

for(int i = 0; i < count; i++)

{

x[i] = rand() % 1000;

y[i] = 0;

for(int j = 0; j < count; j++)

{

a[i \* count + j] = rand() % 1000;

}

}

CUDA\_CHECK\_RETURN(cublasSetMatrix(count, count, sizeof(float), (void \*)a, count, (void \*)a\_dev, count));

CUDA\_CHECK\_RETURN(cublasSetVector(count, sizeof(float), (void \*)x, 1, (void \*)x\_dev, 1));

CUDA\_CHECK\_RETURN(cublasSetVector(count, sizeof(float), (void \*)y, 1, (void \*)y\_dev, 1));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

CUDA\_CHECK\_RETURN(cublasSgemv(cublas\_handle, CUBLAS\_OP\_N, count, count, &alpha, a\_dev, count, x\_dev, 1, &alpha, y\_dev, 1));

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

CUDA\_CHECK\_RETURN(cublasGetMatrix(count, count, sizeof(float), (void \*)a\_dev, count, (void \*)a, count));

CUDA\_CHECK\_RETURN(cublasGetVector(count, sizeof(float), (void \*)x\_dev, 1, (void \*)x, 1));

CUDA\_CHECK\_RETURN(cublasGetVector(count, sizeof(float), (void \*)y\_dev, 1, (void \*)y, 1));

printf("%d - cuBLAS: %f ms\n", count, time);

max[k] += time;

CUDA\_CHECK\_RETURN(cudaFreeHost(a));

CUDA\_CHECK\_RETURN(cudaFreeHost(x));

CUDA\_CHECK\_RETURN(cudaFreeHost(y));

CUDA\_CHECK\_RETURN(cudaFree(a\_dev));

CUDA\_CHECK\_RETURN(cudaFree(x\_dev));

CUDA\_CHECK\_RETURN(cudaFree(y\_dev));

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 15; i++)

{

cout << max[i] / 10 << endl;

max[i] = 0;

}

CUDA\_CHECK\_RETURN(cublasDestroy(cublas\_handle));

return 0;

}

**3.SGEMM**

#pragma comment (lib, "cublas.lib")

#include <stdio.h>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <iostream>

#include <ctime>

#include <thrust/device\_vector.h>

#include <thrust/fill.h>

#include <thrust/host\_vector.h>

#include <thrust/sequence.h>

#include <thrust/transform.h>

#include <cublas.h>

#include <cublas\_v2.h>

using namespace std;

const int N = 1 << 10;

#define V 0.2

#define T 2

#define CUDA\_CHECK\_RETURN(value) ((cudaError\_t)value != cudaSuccess) ? printf("Error %s at line %d in the file %s\n", cudaGetErrorString((cudaError\_t)value), \_\_LINE\_\_, \_\_FILE\_\_) : printf("")

\_\_global\_\_ void sgemmblas(float \*a, float \*b, float \*c, int count)

{

int bx = blockIdx.x;

int by = blockIdx.y;

int tx = threadIdx.x;

int ty = threadIdx.y;

float sum = 0;

int ia = count \* bx + tx;

int ib = bx + tx;

for(int k = 0; k < count; k++)

{

sum += a[ia + k] \* b[ib + k \* count];

}

int ic = count \* by \* bx;

c[ic + count \* ty + tx] = sum;

}

struct dp

{

float \*A, \*B;

int count;

dp(float \*\_A, float \*\_B, int \_count): A(\_A), B(\_B), count(\_count){};

\_\_host\_\_ \_\_device\_\_

float operator()(size\_t idx)

{

float sum = 0.0f;

int row = idx / count;

int col = idx - (row \* count);

for (int i = 0; i < count; i++)

{

sum += A[col + row \* i] \* B[col + row \* i];

}

return sum;

}

};

void sgemm(int count, thrust::device\_vector<float> &data, thrust::device\_vector<float> &other, thrust::device\_vector<float> &result)

{

thrust::transform(thrust::counting\_iterator<int>(0), thrust::counting\_iterator<int>(count \* count), result.begin(), dp(thrust::raw\_pointer\_cast(data.data()), thrust::raw\_pointer\_cast(other.data()), count));

}

int main()

{

srand(time(NULL));

cudaEvent\_t start, stop;

float time;

float alpha = 1;

float max[11];

int k = 0;

for(int i = 0; i < 11; i++)

{

max[i] = 0;

}

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

cublasHandle\_t cublas\_handle;

CUDA\_CHECK\_RETURN(cublasCreate(&cublas\_handle));

cudaDeviceProp deviceProp;

CUDA\_CHECK\_RETURN(cudaGetDeviceProperties(&deviceProp, 0));

int count = 1;

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

float \*a;

float \*b;

float \*c;

float \*a\_dev;

float \*b\_dev;

float \*c\_dev;

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&a\_dev, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&b\_dev, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&c\_dev, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&a, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&b, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&c, count \* count \* sizeof(float)));

for(int i = 0; i < count; i++)

{

for(int j = 0; j < count; j++)

{

a[i \* count + j] = rand() % 1000;

b[i \* count + j] = rand() % 1000;

c[i \* count + j] = 0;

}

}

CUDA\_CHECK\_RETURN(cudaMemcpy(a\_dev, a, count \* count \* sizeof(float), cudaMemcpyHostToDevice));

CUDA\_CHECK\_RETURN(cudaMemcpy(b\_dev, b, count \* count \* sizeof(float), cudaMemcpyHostToDevice));

CUDA\_CHECK\_RETURN(cudaMemcpy(c\_dev, c, count \* count \* sizeof(float), cudaMemcpyHostToDevice));

dim3 threadBlock(1024, 1);

dim3 blockGrid(count / 1024 + 1, 1, 1);

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

sgemmblas <<< blockGrid, threadBlock >>> (a\_dev, b\_dev, c\_dev, count);

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

CUDA\_CHECK\_RETURN(cudaMemcpy(c, c\_dev, count \* count \* sizeof(float), cudaMemcpyDeviceToHost));

printf("%d - CUDA: %f ms\n", count, time);

max[k] += time;

CUDA\_CHECK\_RETURN(cudaFreeHost(a));

CUDA\_CHECK\_RETURN(cudaFreeHost(b));

CUDA\_CHECK\_RETURN(cudaFreeHost(c));

CUDA\_CHECK\_RETURN(cudaFree(a\_dev));

CUDA\_CHECK\_RETURN(cudaFree(b\_dev));

CUDA\_CHECK\_RETURN(cudaFree(c\_dev));

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 11; i++)

{

cout << max[i] / 10 << endl;

max[i] = 0;

}

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

thrust::device\_vector<float> data(count \* count, 2);

thrust::device\_vector<float> other(count \* count, 5);

thrust::device\_vector<float> result(count \* count, 0);

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

sgemm(count, data, other, result);

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

printf("%d - THRUST: %f ms\n", count, time);

max[k] += time;

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 11; i++)

{

cout << max[i] / 10 << endl;

max[i] = 0;

}

for(int i = 0; i < 10; i++)

{

count = 1;

while(count <= N)

{

float \*a;

float \*b;

float \*c;

float \*a\_dev;

float \*b\_dev;

float \*c\_dev;

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&a\_dev, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&b\_dev, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void \*\*)&c\_dev, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&a, count \* count \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&b, count \* count \*sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void \*\*)&c, count \* count \* sizeof(float)));

for (int i = 0; i < count; i++)

{

for(int j = 0; j < count; j++)

{

a[i \* count + j] = rand() % 1000;

b[i \* count + j] = rand() % 1000;

c[i \* count + j] = 0;

}

}

CUDA\_CHECK\_RETURN(cublasSetMatrix(count, count, sizeof(float), (void \*)a, count, (void \*)a\_dev, count));

CUDA\_CHECK\_RETURN(cublasSetMatrix(count, count, sizeof(float), (void \*)b, count, (void \*)b\_dev, count));

CUDA\_CHECK\_RETURN(cublasSetMatrix(count, count, sizeof(float), (void \*)c, count, (void \*)c\_dev, count));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(start));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

CUDA\_CHECK\_RETURN(cublasSgemm(cublas\_handle, CUBLAS\_OP\_N, CUBLAS\_OP\_N, count, count, count, &alpha, a\_dev, count, b\_dev, count, &alpha, c\_dev, count));

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&time, start, stop));

CUDA\_CHECK\_RETURN(cublasGetMatrix(count, count, sizeof(float), (void \*)a\_dev, count, (void \*)a, count));

CUDA\_CHECK\_RETURN(cublasGetMatrix(count, count, sizeof(float), (void \*)b\_dev, count, (void \*)b, count));

CUDA\_CHECK\_RETURN(cublasGetMatrix(count, count, sizeof(float), (void \*)c\_dev, count, (void \*)c, count));

printf("%d - cuBLAS: %f ms\n", count, time);

max[k] += time;

CUDA\_CHECK\_RETURN(cudaFreeHost(a));

CUDA\_CHECK\_RETURN(cudaFreeHost(b));

CUDA\_CHECK\_RETURN(cudaFreeHost(c));

CUDA\_CHECK\_RETURN(cudaFree(a\_dev));

CUDA\_CHECK\_RETURN(cudaFree(b\_dev));

CUDA\_CHECK\_RETURN(cudaFree(c\_dev));

count <<= 1;

k++;

}

k = 0;

}

for(int i = 0; i < 11; i++)

{

cout << max[i] / 10 << endl;

max[i] = 0;

}

CUDA\_CHECK\_RETURN(cublasDestroy(cublas\_handle));

return 0;

}