**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**"Уфимский государственный авиационный технический университет"**

**Кафедра** Высокопроизводительных вычислительных технологий и систем

**Дисциплина:** Технологийпараллельного программирования

**Отчет по лабораторной работе №6**

**Тема:** «Параллельное вычисление произведения двух матриц на графическом процессоре.»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа ПМ-357 | Фамилия И.О. | Подпись | Дата | Оценка |
| Студент | Лонщакова А.А. |  |  |  |
| Принял | Добровольцев А.С. |  |  |  |

**Уфа 2022**

**Цель:** на примере задачи параллельного вычисления произведения двух матриц научиться проводить декомпозицию на двумерные блоки, а также оптимизировать программы для GPU с использованием разделяемой памяти.

**Теоретическая часть**

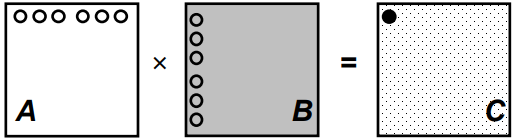
Глобальная память – память, в которую могут записывать и читать все нити GPU. Скорость доступа при это не самая большая.

Разделяемая память – быстрая память, расположенная на мультипроцессоре. Объявляется разделяемая память с помощью спецификатора \_\_shared\_\_.

Вычисление матриц

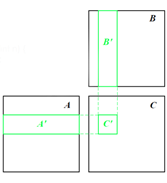
Требуется перемножить квадратные матрицы и вычислить квадрат евклидовой нормы результирующей матрицы:

Матричное умножение выражается формулой



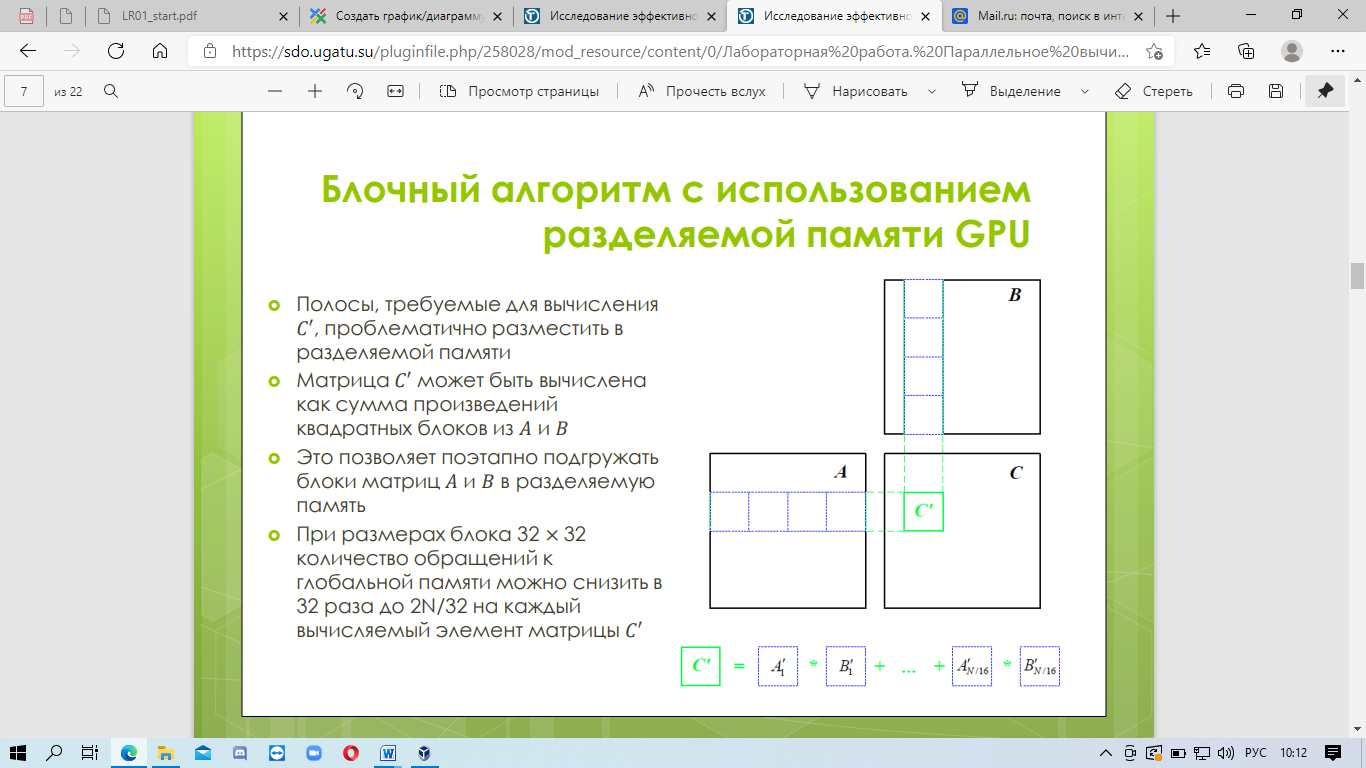
*Рисунок 1.Умножение матриц.*

На GPU это можно реализовать, убрав два внешних цикла из классического алгоритма умножения матриц. Простейшим способом будет использование глобальной памяти.



*Рисунок 2. Простейший алгоритм с использованием глобальной памяти GPU.*

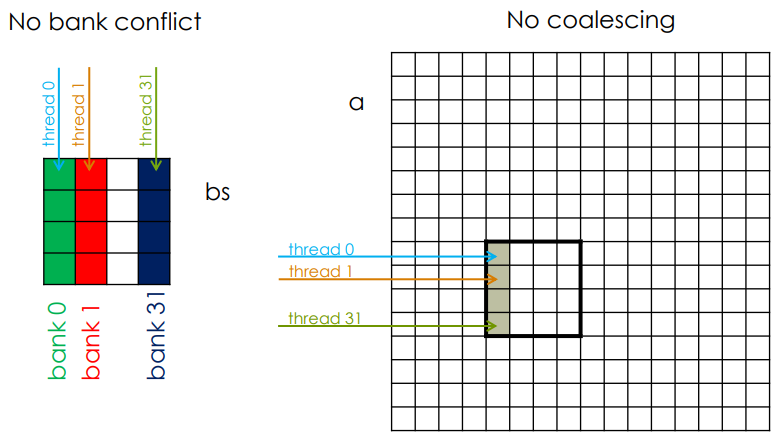
Можно оптимизировать данный алгоритм, снизив количество обращений в долгую глобальную память.



*Рисунок 3. Блочный алгоритм с использованием разделяемой памяти на GPU.*

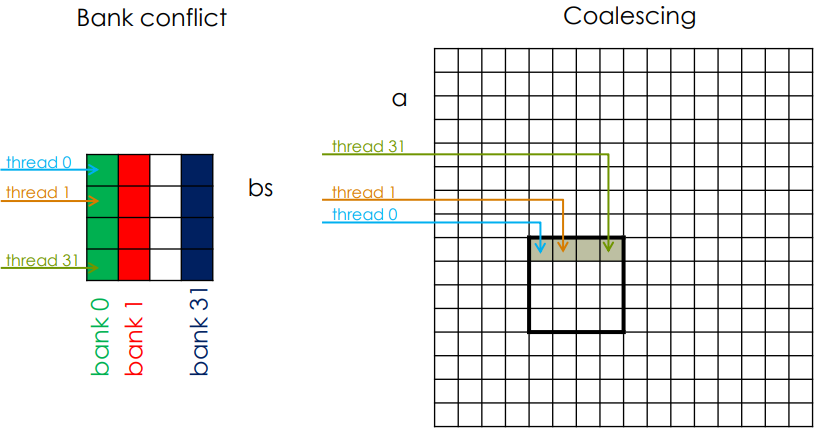
Матриц 𝐶′ может быть вычислена как сумма произведений квадратных блоков из A и B. Это дает нам возможность поэтапно подгружать блоки матриц в разделяемую память. При размерах блока 32 × 32 количество обращений к глобальной памяти можно снизить в 32 раза до 2N/32 на каждый вычисляемый элемент матрицы 𝐶′.

Существуют дальнейшие оптимизации при вычислении произведения матриц, где место B мы используем AT. Т.к. размер матрицы в разделяемой памяти 32, каждый столбец хранится в одном банке. В классической реализации не происходит банк конфликта, но нет coalescing’а.



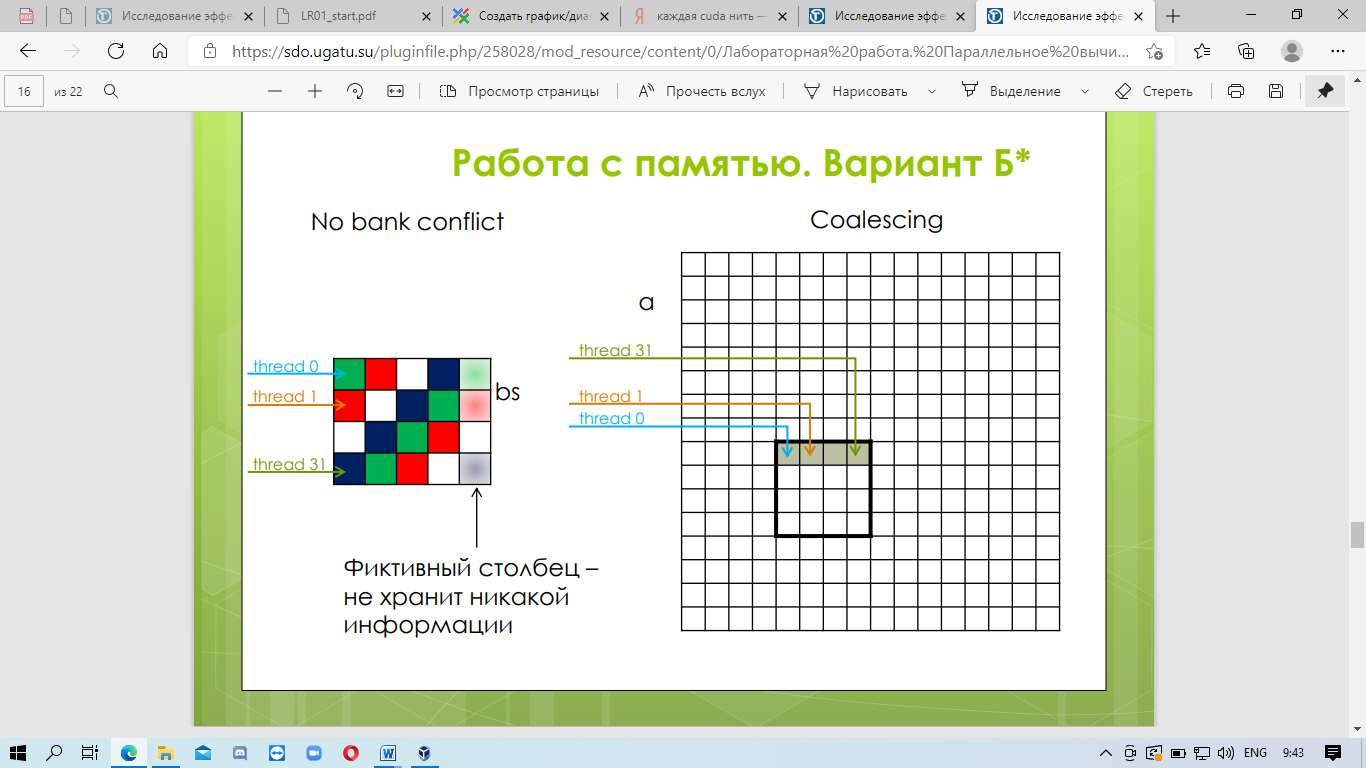
*Рисунок 4. Вариант А.*

Второй вариант – считывать строки из глобальной памяти и записывать в столбцы, тогда происходит coalescing, но при этом существует банк конфликт 32 порядка.



*Рисунок 5. Вариант Б.*

Попробуем оптимизировать данный алгоритм, избавившись от банк-конфликтов. Достаточно добавить фиктивный столбец, который не хранит данных, тогда элементы столбца будут лежать в разных банках, следовательно, не будет конфликтов.



*Рисунок 6. Вариант Б\*.*

**Практическая часть**

Параметры графического процессора, на котором проводились вычисления:

1. Архитектура AMPERE
2. CUDA Capability Major/Minor version number 8.6
3. Total amount of global memory 8GB
4. Пропускная способность памяти 224 GB/s
5. Число СUDA-ядер 2560
6. Пиковая производительность 9.098 TFLOPS

**Задачи**

* Разработка программы, реализующей перемножения двух квадратных матриц на CPU (последовательно и параллельно);
* Реализация простейшего алгоритма вычисления матричного произведения на GPU с использованием глобальной памяти;
* Реализация блочного алгоритма вычисления матричного произведения с использованием разделяемой памяти;
* Реализация различных вариантов вычисления ;
* Тестирование производительности и профилирование разработанных программ.

**Индивидуальное задание №1.**

Задание:

1. Размерности матриц предполагаются кратными 32 (если нет, программа завершается с сообщением об ошибке).
2. Перемножаются матрицы 𝐴 и 𝐵 с элементами вещественного типа одинарной точности, сгенерированные ранее на CPU.
3. Необходимо предусмотреть расчет нормы матрицы 𝐶.
4. Реализовать последовательную и многопоточную (OpenMP) версии для CPU, а также с использованием глобальной и разделяемой памяти для GPU.

*При компиляции для CPU выставить ключ -O2;*

*При компиляции для GPU выставить ключ -arch=sm\_50 или выше.*

*Таблица 1. Время выполнения программ.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Версия, платформа | Время(ms) | N=256 | N = 1024 | N = 4096 |
| Последовательная, CPU | Время расчета на CPU | 25 | 7114.83 | 556762 |
| Параллельная, CPU | Время расчета на CPU | 8.4235 | 4393.98 | 316807 |
| С использованием глобальной памяти, GPU | Время обмена данными с GPU | 0.32 | 2.6305 | 29.668 |
| Время расчета на GPU | 0.18 | 6.579 | 280.751 |
| Полное время работы на GPU | 0.5 | 9,209 | 310.419 |
| С использованием разделяемой памяти, GPU | Время обмена данными с GPU | 0.285 | 2.204 | 33.892 |
| Время расчета на GPU | 0.1467 | 3.516 | 242.453 |
| Полное время работы на GPU | 0,4317 | 5,72 | 276,345 |

*Таблица 2. Ускорение на GPU относительно параллельной версии на CPU.*

|  |  |  |  |
| --- | --- | --- | --- |
| Версия | N = 256 | N = 1024 | N = 4096 |
| С использованием глобальной памяти | 16,847 | 477,03 | 1 020,57 |
| С использованием разделяемой памяти | 19,512 | 768 | 1 146,41 |

*Таблица 3. Производительность разработанных версий.*

|  |  |  |  |
| --- | --- | --- | --- |
| Версия, платформа | N = 256 | N = 1024 | N = 4096 |
| Параллельная, CPU | 3,21 | 3,49 | 0,41 |
| С использованием глобальной памяти, GPU | 90,69 | 112,97 | 145,85 |
| С использованием разделяемой памяти, GPU | 145,89 | 219,58 | 437,9 |

*Уточнение. Используемое в расчёте время было переведено в секунды.*

По полученным данным, видно, что любая реализация на GPU выигрывает по времени версии на CPU. Более того, расчеты с использованием разделяемой памяти GPU являются наиболее производительными.

**Индивидуальное задание №2.**

Задание:

1. Реализовать три версии ядра перемножения матрицы A на AT с использованием разделяемой памяти.

1. Без конфликтов банков и без коалесинга (Вариант А).

2. С конфликтами банков и коалесингом (Вариант Б).

3. Без конфликтов банков и с коалесингом (Вариант Б\*).

2. Провести исследование полученных реализаций.

При компиляции выставить ключ -arch=sm\_50 (не меньше, чем 50).

*Таблица 4. Время выполнения функции ядра.*

|  |  |  |  |
| --- | --- | --- | --- |
| Версия | N=256 | N=1024 | N=4096 |
| Вариант А | 0,243 | 4.46 | 316.29 |
| Вариант Б | 0.42 | 5.664 | 298.65 |
| Вариант Б\* | 0.17 | 3.34 | 218.15 |

Реальной производительностью называется количество команд или операций, которое выполняет вычислительная система в единицу времени для конкретного алгоритма.

*Таблица 5. Производительность функции-ядра (GFlops)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Версия | Размерность | N=256 | N=1024 | N=4096 |
| Вариант А | Производительность | 138.084 | 481.49 | 434.53 |
| % от пиковой | 1.53 | 5.349 | 4.82 |
| Вариант Б | Производительность | 79.891 | 379.14 | 460.20 |
| % от пиковой | 0.88 | 4.212 | 5.11 |
| Вариант Б\* | Производительность | 197.379 | 642.959 | 630.02 |
| % от пиковой | 2.19 | 7.1439 | 7 |

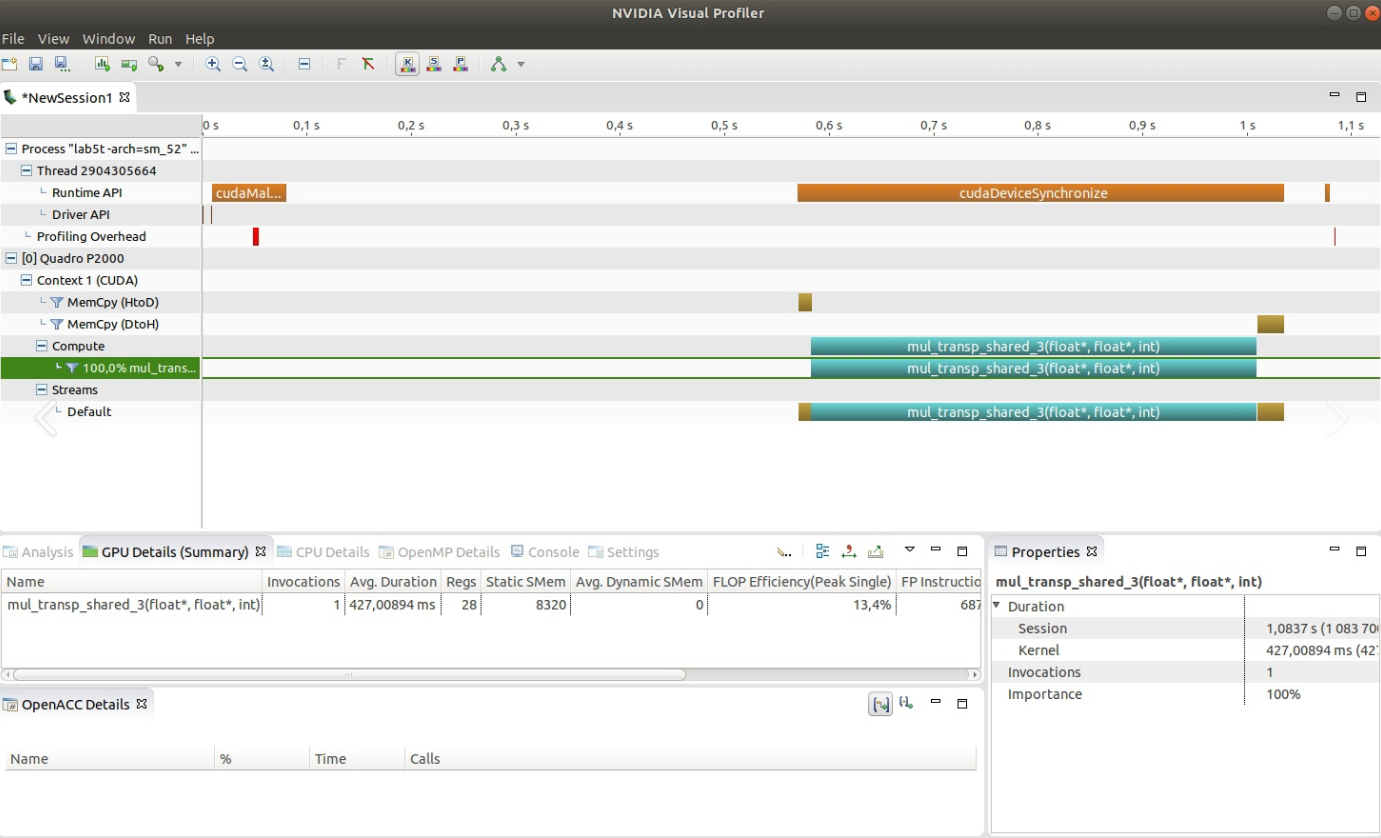
*Рисунок 7. Процент от пиковой производительности ядра.*

https://sun9-39.userapi.com/impg/JZaL-PFA6wFT_IfHkY5JhEvOZ83n60Dp9fFkFA/eqPAgRnxsmg.jpg?size=485x58&quality=96&sign=28a49de3da5f2f8b6f96025d0652cefe&type=album

https://sun9-8.userapi.com/impg/WSPXkr0bGDH4ZUAAIBExEF6Hb5CKoOf2Cthg9g/R2VaJvayXqU.jpg?size=526x53&quality=96&sign=57bb1a0939cebb23a3882526e6b0540c&type=album



*Рисунок 8. Скриншоты банк конфликта для варианта А, Б, Б\* соответственно.*



*Рисунок 9. Скриншот NVIDIA Profiler для Б\* и N=4096.*

**Вывод**

В данной лабораторной работе, на примере произведения двух матриц, была продемонстрирована оптимизация программы для GPU с использованием разделяемой памяти. Были рассмотрены различные способы повышения производительности программы.

**Приложение**

**1. Параллельная, CPU.**

#include <iostream>

#include <time.h>

#include <math.h>

#include <cstdlib>

#include <random>

#include <omp.h>

#define n 256

using namespace std;

int main()

{

double\* matrix1 = new double[n \* n];

double\* matrix2 = new double[n \* n];

double\* matrix3 = new double[n \* n];

cout << "N = " << n << endl;

int q = 1;

srand(time(NULL));

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

{

matrix1[i] = (double)rand() / RAND\_MAX - 0.5;

matrix2[i] = (double)rand() / RAND\_MAX - 0.5;

}

double sum = 0;

double start\_time = omp\_get\_wtime();

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

{

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

{

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

{

sum = 0;

#pragma omp parallel for reduction(+:sum) num\_threads(4)

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

{

sum += matrix1[i \* n + k] \* matrix2[j + n \* k];

}

matrix3[i \* n + j] = sum;

}

}

}

double result = 0;

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

{

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

{

result = 0;

result += matrix3[i \* n + j] \* matrix3[i \* n + j];

}

}

double end\_time = omp\_get\_wtime(); // конечное время

cout << "Program execution time = " << (end\_time - start\_time) << endl;

cout << "Norm = " << sqrt(result) << endl;

free(matrix1);

free(matrix2);

free(matrix3);

return 0;

}

**2. На CUDA C=AB.**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

constexpr int N = 1024;

constexpr int BS = 32;

constexpr int BLOCK\_SIZE = 32;

#include <cstdlib>

#include <iostream>

#include <math.h>

\_\_global\_\_ void mul\_shared(float\* a, float\* b, float\* c, int N)

{

int i, j,

bx = blockIdx.x, by = blockIdx.y,

tx = threadIdx.x, ty = threadIdx.y,

ix = bx \* blockDim.x + tx, iy = by \* blockDim.y + ty;

float s = 0.;

\_\_shared\_\_ float as[BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float bs[BLOCK\_SIZE][BLOCK\_SIZE];

for (i = 0; i < N / BLOCK\_SIZE; i++)

{

as[ty][tx] = a[(by \* BLOCK\_SIZE + ty) \* N + i \* BLOCK\_SIZE + tx];

bs[ty][tx] = b[(i \* BLOCK\_SIZE + ty) \* N + bx \* BLOCK\_SIZE + tx];

\_\_syncthreads();

for (j = 0; j < BLOCK\_SIZE; j++)

s += as[ty][j] \* bs[j][tx];

\_\_syncthreads();

}

c[iy \* N + ix] = s;

}

\_\_global\_\_ void kernel(float\* a, float\* b, float\* c, int n)

{

int row = blockIdx.y \* blockDim.y + threadIdx.y;

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

float s = 0.;

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

s += a[row \* n + i] \* b[i \* n + col];

c[row \* n + col] = s;

}

void random\_floats(float\* a, int n)

{

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

{

a[i] = (float)rand() / RAND\_MAX;

}

}

int main()

{

if (N % 32 != 0)

{

std::cout << "N is not correct\n";

exit(1);

}

printf("N = %d\n", N);

srand(0);

float\* d\_a, \* d\_b, \* d\_c;

float\* a, \* b, \* c;

float norma = 0.;

dim3 threads(BS, BS);

dim3 blocks(N / BS, N / BS);

int size = N \* N \* sizeof(float);

cudaEvent\_t start, stop;

float time = 0;

a = new float[N \* N];

b = new float[N \* N];

c = new float[N \* N];

cudaMalloc((void\*\*)&d\_a, size);

cudaMalloc((void\*\*)&d\_b, size);

cudaMalloc((void\*\*)&d\_c, size);

random\_floats(a, N \* N);

random\_floats(b, N \* N);

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

cudaMemcpy(d\_a, a, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, b, size, cudaMemcpyHostToDevice);

cudaDeviceSynchronize();

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);

printf("Memory copy time: %.10f ms\n", time);

cudaEventDestroy(start);

cudaEventDestroy(stop);

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

mul\_shared <<<blocks, threads>>>(d\_a, d\_b, d\_c, N);

//kernel <<<blocks, threads>>> (d\_a, d\_b, d\_c, N);

cudaDeviceSynchronize();

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);

printf("Computing time : %.10f ms\n", time);

cudaEventDestroy(start);

cudaEventDestroy(stop);

cudaMemcpy(c, d\_c, size, cudaMemcpyDeviceToHost);

cudaError\_t error = cudaGetLastError();

if (error != cudaSuccess)

{

std::cout << "Cuda error: " << cudaGetErrorString(error) « std::endl;

}

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

{

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

{

norma += c[N \* i + j] \* c[N \* i + j];

}

}

std::cout << "norma = " << sqrt(norma) << std::endl;

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

delete[] a;

delete[] b;

delete[] c;

return 0;

}

**3. На CUDA .**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

constexpr int N = 1024;

constexpr int BS = 32;

constexpr int BLOCK\_SIZE = 32;

#include <cstdlib>

#include <iostream>

\_\_global\_\_ void mul\_transp\_shared\_1(float\* a, float\* c, int N) //no bank conflict, no coalescing

{

int i, j,

bx = blockIdx.x, by = blockIdx.y,

tx = threadIdx.x, ty = threadIdx.y,

ix = bx \* blockDim.x + tx, iy = by \* blockDim.y + ty;

float s = 0.;

\_\_shared\_\_ float as[BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float bs[BLOCK\_SIZE][BLOCK\_SIZE];

for (i = 0; i < N / BLOCK\_SIZE; i++)

{

as[ty][tx] = a[(by \* BLOCK\_SIZE + ty) \* N + i \* BLOCK\_SIZE + tx];

bs[ty][tx] = a[(bx \* BLOCK\_SIZE + tx) \* N + i \* BLOCK\_SIZE + ty];

\_\_syncthreads();

for (j = 0; j < BLOCK\_SIZE; j++)

s += as[ty][j] \* bs[j][tx];

\_\_syncthreads();

}

c[iy \* N + ix] = s;

}

\_\_global\_\_ void mul\_transp\_shared\_2(float\* a, float\* c, int N) //bank conflict, coalescing

{

int i, j,

bx = blockIdx.x, by = blockIdx.y,

tx = threadIdx.x, ty = threadIdx.y,

ix = bx \* blockDim.x + tx, iy = by \* blockDim.y + ty;

float s = 0.;

\_\_shared\_\_ float as[BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float bs[BLOCK\_SIZE][BLOCK\_SIZE];

for (i = 0; i < N / BLOCK\_SIZE; i++)

{

as[ty][tx] = a[(i \* BLOCK\_SIZE + ty) \* N + bx \* BLOCK\_SIZE + tx];

bs[tx][ty] = a[(i \* BLOCK\_SIZE + ty) \* N + bx \* BLOCK\_SIZE + tx];

\_\_syncthreads();

for (j = 0; j < BLOCK\_SIZE; j++)

s += as[ty][j] \* bs[j][tx];

\_\_syncthreads();

}

c[iy \* N + ix] = s;

}

\_\_global\_\_ void mul\_transp\_shared\_3(float\* a, float\* c, int N) //no bank conflict, coalescing

{

int i, j,

bx = blockIdx.x, by = blockIdx.y,

tx = threadIdx.x, ty = threadIdx.y,

ix = bx \* blockDim.x + tx, iy = by \* blockDim.y + ty;

float s = 0.;

\_\_shared\_\_ float as[BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float bs[BLOCK\_SIZE][BLOCK\_SIZE + 1];

for (i = 0; i < N / BLOCK\_SIZE; i++)

{

as[ty][tx] = a[(i \* BLOCK\_SIZE + ty) \* N + bx \* BLOCK\_SIZE + tx];

bs[tx][ty] = a[(i \* BLOCK\_SIZE + ty) \* N + bx \* BLOCK\_SIZE + tx];

\_\_syncthreads();

for (j = 0; j < BLOCK\_SIZE; j++)

s += as[ty][j] \* bs[j][tx];

\_\_syncthreads();

}

c[iy \* N + ix] = s;

}

void random\_floats(float\* a, int n)

{

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

{

a[i] = (float)rand() / RAND\_MAX;

}

}

int main()

{

if (N % 32 != 0)

{

std::cout << "N is not correct\n";

exit(1);

}

printf("N = %d\n", N);

srand(0);

float\* d\_a, \* d\_b, \* d\_c;

float\* a, \* b, \* c;

float norma = 0.;

dim3 threads(BS, BS);

dim3 blocks(N / BS, N / BS);

int size = N \* N \* sizeof(float);

cudaEvent\_t start, stop;

float time = 0;

a = new float[N \* N];

b = new float[N \* N];

c = new float[N \* N];

cudaMalloc((void\*\*)&d\_a, size);

cudaMalloc((void\*\*)&d\_b, size);

cudaMalloc((void\*\*)&d\_c, size);

random\_floats(a, N \* N);

random\_floats(b, N \* N);

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

cudaMemcpy(d\_a, a, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, b, size, cudaMemcpyHostToDevice);

cudaDeviceSynchronize();

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);

printf("Memory copy time: %.10f ms\n", time);

cudaEventDestroy(start);

cudaEventDestroy(stop);

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

mul\_transp\_shared\_1 <<<blocks, threads>>> (d\_a, d\_c, N);

//mul\_transp\_shared\_2 <<<blocks, threads>>>(d\_a, d\_c, N);

//mul\_transp\_shared\_3 <<<blocks, threads>>> (d\_a, d\_c, N);

cudaDeviceSynchronize();

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);

printf("Computing time : %.10f ms\n", time);

cudaEventDestroy(start);

cudaEventDestroy(stop);

cudaMemcpy(c, d\_c, size, cudaMemcpyDeviceToHost);

cudaError\_t error = cudaGetLastError();

if (error != cudaSuccess)

{

std::cout << "Cuda error: " << cudaGetErrorString(error) << std::endl;

}

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

{

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

{

norma += c[N \* i + j] \* c[N \* i + j];

}

}

std::cout << "norma = " << sqrt(norma) << std::endl;

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

delete[] a;

delete[] b;

delete[] c;

return 0;

}