Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет

телекоммуникаций и информатики»

**Лабораторная работа №4**

**«Разделяемая память»**

Выполнил: студент 3 курса

ИВТ, гр. ИП-713

Михеев Н.А.

Проверил: ассистент кафедры ПМиК

Нужнов А.В.

Новосибирск, 2020 г.

**Задание на лабораторную работу**

• написать программу транспонирования матриц, реализующую алгоритм

без использования разделяемой памяти, наивный алгоритм с

использованием разделяемой памяти и алгоритм с разрешением

конфликта банков разделяемой памяти;

• провести профилирование программы с использованием nvprof или nvpp -

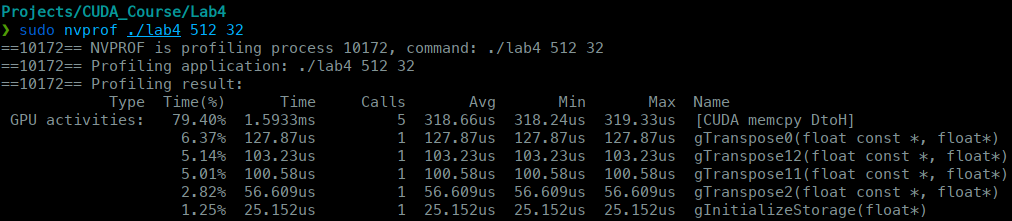
сравнить время выполнения ядер, реализующих разные алгоритмы, и

оценить эффективность использования разделяемой памяти (лекция 4).

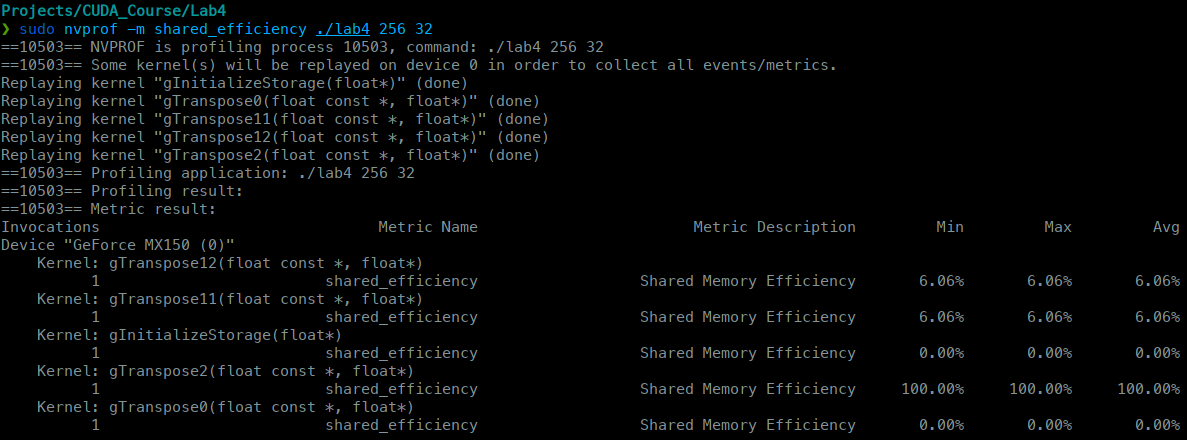
### **Ход выполнения лабораторной работы**

Для выполнения лабораторной работы была разработана программа для транспонирования матриц с различными способами использования разделяемой памяти. Готовая программа была отпрофилированна с получением результатов требуемых метрик.

В программе были реализованны функции: gTranspose0() - транспонирование без использования разделяемой памяти, gTransopose11() - транспонирование с «наивным» использованием разделяемой памяти с динамическим выделением, gTranspose12() - транспонирование с «наивным» использованием разделяемой памяти со статическим выделением памяти и gTranspose2() - транспонирование с использованием разделяемой памяти избавленная от конфликтов банков разделяемой пямяти.

 Рис. 1 — запуск программы под профилировщиком nvprof

На рисунке №1 видно, что чем правильнее используется разделяемая память, тем эффективнее происходит транспонирование матрицы. Время выполнения программы при «наивном» использовании памяти - gTranspose11 и после устранения конфликтов банков пямяти - gTranspose2 различается практически в 2 раза.

Рис. 2 — результат работы профилировщика с использованием метрики shared\_efficiency

На рисунке №2 предоставлен результат работы профилировщика с ипользованием метрики -m shared\_efficiency, которая отображает эффективность использования разделяемой памяти. Соответственно видно, что наибольшая эффективность достигается при разрешении конфликтов банков разделяемой памяти. Эффективность использования памяти остается примерно одинаковой (разница в сотых долях процента ± 0.01% при различных размерах матрицы) в статическом и динамическом выделении памяти.

**Листинг программы**

#include <iostream>

#include <cuda\_runtime.h>

void Output(float \*a, int N) {

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

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

fprintf(stdout, "%g\t", a[j + i \* N]);

fprintf(stdout, "\n");

}

fprintf(stdout, "\n\n\n");

}

\_\_global\_\_ void gInitializeStorage(float \*storage\_d) {

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

unsigned j = threadIdx.y + blockIdx.y \* blockDim.y;

unsigned N = blockDim.x \* gridDim.x;

storage\_d[i + j \* N] = (float) (i + j \* N);

}

\_\_global\_\_ void gTranspose0(const float \*storage\_d, float \*storage\_d\_t) {

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

unsigned j = threadIdx.y + blockIdx.y \* blockDim.y;

unsigned N = blockDim.x \* gridDim.x;

storage\_d\_t[j + i \* N] = storage\_d[i + j \* N];

}

\_\_global\_\_ void gTranspose11(const float \*storage\_d, float \*storage\_d\_t) {

extern \_\_shared\_\_ float buffer[];

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

unsigned j = threadIdx.y + blockIdx.y \* blockDim.y;

unsigned N = blockDim.x \* gridDim.x;

buffer[threadIdx.y + threadIdx.x \* blockDim.y] = storage\_d[i + j \* N];

\_\_syncthreads();

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

j = threadIdx.y + blockIdx.x \* blockDim.y;

storage\_d\_t[i + j \* N] = buffer[threadIdx.x + threadIdx.y \* blockDim.x];

}

#define SH\_DIM 32

\_\_global\_\_ void gTranspose12(const float \*storage\_d, float \*storage\_d\_t) {

\_\_shared\_\_ float buffer\_s[SH\_DIM][SH\_DIM];

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

unsigned j = threadIdx.y + blockIdx.y \* blockDim.y;

unsigned N = blockDim.x \* gridDim.x;

buffer\_s[threadIdx.y][threadIdx.x] = storage\_d[i + j \* N];

\_\_syncthreads();

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

j = threadIdx.y + blockIdx.x \* blockDim.y;

storage\_d\_t[i + j \* N] = buffer\_s[threadIdx.x][threadIdx.y];

}

\_\_global\_\_ void gTranspose2(const float \*storage\_d, float \*storage\_d\_t) {

\_\_shared\_\_ float buffer[SH\_DIM][SH\_DIM + 1];

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

unsigned j = threadIdx.y + blockIdx.y \* blockDim.y;

unsigned N = blockDim.x \* gridDim.x;

buffer[threadIdx.y][threadIdx.x] = storage\_d[i + j \* N];

\_\_syncthreads();

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

j = threadIdx.y + blockIdx.x \* blockDim.y;

storage\_d\_t[i + j \* N] = buffer[threadIdx.x][threadIdx.y];

}

int main(int argc, char \*argv[]) {

if (argc < 3) {

fprintf(stderr, "USAGE: matrix <dimension of matrix><dimension\_of\_threads>\n");

return -1;

}

int N = atoi(argv[1]);

int dim\_of\_threads = atoi(argv[2]);

if (N % dim\_of\_threads) {

fprintf(stderr, "change dimensions\n");

return -1;

}

int dim\_of\_blocks = N / dim\_of\_threads;

const int max\_size = 1 << 8;

if (dim\_of\_blocks > max\_size) {

fprintf(stderr, "too many blocks\n");

return -1;

}

float \*storage\_d, \*storage\_d\_t, \*storage\_h;

cudaMalloc((void \*\*) &storage\_d, N \* N \* sizeof(float));

cudaMalloc((void \*\*) &storage\_d\_t, N \* N \* sizeof(float));

storage\_h = (float \*) calloc(N \* N, sizeof(float));

gInitializeStorage<<<dim3(dim\_of\_blocks, dim\_of\_blocks),

dim3(dim\_of\_threads, dim\_of\_threads)>>>(storage\_d);

cudaDeviceSynchronize();

memset(storage\_h, 0.0, N \* N \* sizeof(float));

cudaMemcpy(storage\_h, storage\_d, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);

// Output(storage\_h, N);

gTranspose0<<<dim3(dim\_of\_blocks, dim\_of\_blocks),

dim3(dim\_of\_threads, dim\_of\_threads)>>>(storage\_d, storage\_d\_t);

cudaDeviceSynchronize();

memset(storage\_h, 0.0, N \* N \* sizeof(float));

cudaMemcpy(storage\_h, storage\_d\_t, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);

// Output(storage\_h, N);

gTranspose11<<<dim3(dim\_of\_blocks, dim\_of\_blocks),

dim3(dim\_of\_threads, dim\_of\_threads),

dim\_of\_threads\*dim\_of\_threads\*sizeof(float)>>>(storage\_d, storage\_d\_t);

cudaDeviceSynchronize();

memset(storage\_h, 0.0, N \* N \* sizeof(float));

cudaMemcpy(storage\_h, storage\_d\_t, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);

// Output(storage\_h, N);

gTranspose12<<<dim3(dim\_of\_blocks, dim\_of\_blocks),

dim3(dim\_of\_threads, dim\_of\_threads)>>>(storage\_d, storage\_d\_t);

cudaDeviceSynchronize();

memset(storage\_h, 0.0, N \* N \* sizeof(float));

cudaMemcpy(storage\_h, storage\_d\_t, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);

// Output(storage\_h, N);

gTranspose2<<<dim3(dim\_of\_blocks, dim\_of\_blocks),

dim3(dim\_of\_threads, dim\_of\_threads)>>>(storage\_d, storage\_d\_t);

cudaDeviceSynchronize();

memset(storage\_h, 0.0, N \* N \* sizeof(float));

cudaMemcpy(storage\_h, storage\_d\_t, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);

// Output(storage\_h, N);

cudaFree(storage\_d);

cudaFree(storage\_d\_t);

free(storage\_h);

return 0;

}