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.

```
Projects/CUDA_Course/Lab4
❯ sudo nvprof ./lab4 512 32
==10172== NVPROF is profiling process 10172, command: ./lab4 512 32
==10172== Profiling application: ./lab4 512 32
==10172== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   79.40%  1.5933ms         5  318.66us  318.24us  319.33us  [CUDA memcpy DtoH]
                    6.37%  127.87us         1  127.87us  127.87us  127.87us  gTranspose0(float const *, float*)
                    5.14%  103.23us         1  103.23us  103.23us  103.23us  gTranspose12(float const *, float*)
                    5.01%  100.58us         1  100.58us  100.58us  100.58us  gTranspose11(float const *, float*)
                    2.82%  56.609us         1  56.609us  56.609us  56.609us  gTranspose2(float const *, float*)
                    1.25%  25.152us         1  25.152us  25.152us  25.152us  gInitializeStorage(float*)
```

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.

```
Projects/CUDA_Course/Lab4
❯ sudo nvprof -m shared_efficiency ./lab4 256 32
==10503== NVPROF is profiling process 10503, command: ./lab4 256 32
==10503== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "gInitializeStorage(float*)" (done)
Replaying kernel "gTranspose0(float const *, float*)" (done)
Replaying kernel "gTranspose11(float const *, float*)" (done)
Replaying kernel "gTranspose12(float const *, float*)" (done)
Replaying kernel "gTranspose2(float const *, float*)" (done)
==10503== Profiling application: ./lab4 256 32
==10503== Profiling result:
==10503== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "GeForce MX150 (0)"
    Kernel: gTranspose12(float const *, float*)
          1                     shared_efficiency                  Shared Memory Efficiency       6.06%       6.06%       6.06%
    Kernel: gTranspose11(float const *, float*)
          1                     shared_efficiency                  Shared Memory Efficiency       6.06%       6.06%       6.06%
    Kernel: gInitializeStorage(float*)
          1                     shared_efficiency                  Shared Memory Efficiency       0.00%       0.00%       0.00%
    Kernel: gTranspose2(float const *, float*)
          1                     shared_efficiency                  Shared Memory Efficiency     100.00%     100.00%     100.00%
    Kernel: gTranspose0(float const *, float*)
          1                     shared_efficiency                  Shared Memory Efficiency       0.00%       0.00%       0.00%
```

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;
}
```