Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Сибирский государственный университет
телекоммуникаций и информатики» (СибГУТИ)

09.03.01 "Информатика и вычислительная техника"
профиль "Программное обеспечение средств
вычислительной техники и автоматизированных систем"

ОТЧЕТ
по дисциплине «Программирование графических процессоров»
лабораторная работа 4

Выполнил:
Разумов Д. Б.
студент группы ИП-811

Проверил:
преподаватель Малков Е.А.

Новосибирск 2020

Оглавление

Задание

Задание:

• 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).

Цель: научиться использовать разделяемую память.

Листинг

```
#include <stdio.h>
#include <time.h>
#include <malloc.h>

#define CUDA_CHECK_RETURN(value) {\
	cudaError_t _m_cudaStat = value;\
	if (_m_cudaStat != cudaSuccess) {\
		fprintf(stderr, "Error \"%s\" at line %d in file %s\n",\
			cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);\
		exit(1);\
	}\
} //макрос для обработки ошибок

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 gInitializeMatrixByRows(float* matrix_d){
	int i = threadIdx.x + blockIdx.x * blockDim.x;
	int j = threadIdx.y + blockIdx.y * blockDim.y;
	int N = blockDim.x * gridDim.x;
```

```
        matrix_d[i+j*N] = (float)(i+j*N);
}

__global__ void gInitializeMatrixByColumns(float* matrix_d){
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        int j = threadIdx.y + blockIdx.y * blockDim.y;
        int N = blockDim.x * gridDim.x;
        matrix_d[j+i*N] = (float)(j+i*N);
}

__global__ void gTranspose0(float* storage_d, float* storage_d_t){
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        int j = threadIdx.y + blockIdx.y * blockDim.y;
        int N = blockDim.x * gridDim.x;
        storage_d_t[j + i * N] = storage_d[i + j * N];
}

__global__ void gTranspose11(float* storage_d, float* storage_d_t) {
        extern __shared__ float buffer[];
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        int j = threadIdx.y + blockIdx.y * blockDim.y;
        int 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(float* storage_d, float* storage_d_t) {
    __shared__ float buffer_s[SH_DIM][SH_DIM];
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int 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(float* storage_d, float* storage_d_t) {
    __shared__ float buffer[SH_DIM][SH_DIM+1];
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int 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));
gInitializeMatrixByRows<<<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);
if (argc == 4) {
        printf("изначальная матрица:\n");
        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);
if (argc == 4) {
        printf("gTranspose0:\n");
        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);
if (argc == 4) {
        printf("gTranspose11:\n");
        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);
if (argc == 4) {
        printf("gTranspose12:\n");
        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);
	if (argc == 4) {
		printf("gTranspose2:\n");
		Output(storage_h, N);
	}

	cudaFree(storage_d);
	cudaFree(storage_d_t);
	free(storage_h);
	return 0;
}
```

Скриншоты

```
dmitry@pc:~/CUDA/lab4$ sudo nvprof ./lab4.exe 4096 32
==11829== NVPROF is profiling process 11829, command: ./lab4.exe 4096 32
==11829== Profiling application: ./lab4.exe 4096 32
==11829== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   84.45%  102.79ms         5  20.557ms  20.482ms  20.607ms  [CUDA memcpy DtoH]
                    5.39%  6.5542ms         1  6.5542ms  6.5542ms  6.5542ms  gTranspose0(float*, float*)
                    3.44%  4.1871ms         1  4.1871ms  4.1871ms  4.1871ms  gTranspose12(float*, float*)
                    3.39%  4.1300ms         1  4.1300ms  4.1300ms  4.1300ms  gTranspose11(float*, float*)
                    2.25%  2.7419ms         1  2.7419ms  2.7419ms  2.7419ms  gTranspose2(float*, float*)
```

Рисунок 1 — время выполнения разных ядер по транспанированию матриц

```
dmitry@pc:~/CUDA/lab4$ sudo nvprof -m shared_efficiency ./lab4.exe 2048 32
==11623== NVPROF is profiling process 11623, command: ./lab4.exe 2048 32
==11623== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "gInitializeMatrixByRows(float*)" (done)
Replaying kernel "gTranspose0(float*, float*)" (done)
Replaying kernel "gTranspose11(float*, float*)" (done)
Replaying kernel "gTranspose12(float*, float*)" (done)
Replaying kernel "gTranspose2(float*, float*)" (done)
==11623== Profiling application: ./lab4.exe 2048 32
==11623== Profiling result:
==11623== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce MX350 (0)"
    Kernel: gTranspose0(float*, float*)
          1                         shared_efficiency                  Shared Memory Efficiency       0.00%       0.00%       0.00%
    Kernel: gTranspose2(float*, float*)
          1                         shared_efficiency                  Shared Memory Efficiency     100.00%     100.00%     100.00%
    Kernel: gInitializeMatrixByRows(float*)
          1                         shared_efficiency                  Shared Memory Efficiency       0.00%       0.00%       0.00%
    Kernel: gTranspose11(float*, float*)
          1                         shared_efficiency                  Shared Memory Efficiency       6.06%       6.06%       6.06%
    Kernel: gTranspose12(float*, float*)
          1                         shared_efficiency                  Shared Memory Efficiency       6.06%       6.06%       6.06%
dmitry@pc:~/CUDA/lab4$
```

Рисунок 2 — эффективность использования разделяемой памяти

Заключение

Исходя из результатов тестирования, можно сделать вывод, что разделяемая память позволяет быстрее производить вычисления. Также очень ускоряет вычисление устранение конфликта банков разделяемой памяти (в этом более низкая латентность память).