Parallel Architectures and Programming Models

Assignment 3

Krukovich Palina, a12134146

Introduction

Project structure

The assignment contains 4 files:

1) assignment3.cl - file contains kernel implementation.
2) conf.hpp - file contains user settings that might be changed to influence program behavior. After any changes are committed, the project must be recompiled.
3) Makefile - contains the only command to build project
4) assignment3.cpp - contains main and solution.

Project settings

User settings can be changed in the file “conf.hpp”. Below are the main settings that can be changed.

```
// user settings
#define HEIGHT                  8192
#define WIDTH                   8192
#define NUM_OF_ITERS            23

#define WORK_GROUP_SIZE_X       128
#define WORK_GROUP_SIZE_Y       1

#define WORK_GROUP_SIZE         (WORK_GROUP_SIZE_X)

//#define PRINT_MATRIX
```

“HEIGHT” - height of the destination matrix. Must be multiple of 32.
“WIDTH” - width of the destination matrix. Must be multiple of 32.
“NUM_OF_ITERS” - number of iterations to be performed.
“WORK_GROUP_SIZE_X” - workgroup size by x
“WORK_GROUP_SIZE_Y” - workgroup size by y

“WORK_GROUP_SIZE” - workgroup size in total. In case WORK_GROUP_SIZE_Y not 1 should be replaced with (WORK_GROUP_SIZE_Y*WORK_GROUP_SIZE_X).
“PRINT_MATRIX” - uncomment macro in order to print final matrix.

Also within the main file defined “special” macros. They shouldn’t be changed but they critically influence the program. The defines are provided below.

```
#define WARP_SIZE            32
#define MAX_PLATFORMS        8
#define MAX_DEVICES          8
#define TARGET_PLATFORM      "NVIDIA CUDA\0"
#define TARGET_DEVICE        "Tesla K20m\0"
#define NUM_DEVICES_USED     1
```

“WARP_SIZE” - for the Nvidia GPU warp size is 32. We don’t read it by the driver but just define
“MAX_PLATFORMS” - represents how many platforms we read to check for the “TARGET_PLATFORM”
“TARGET_PLATFORM” - platform we tent to use
“MAX_DEVICES” - represents how many devices we read to check for “TARGET_DEVICE”
“TARGET_DEVICE” - device we tent to use
“NUM_DEVICES_USED” - number of devices we tend to use

To prevent incorrect values usage static asserts are used:

```
static_assert(WORK_GROUP_SIZE_X > 0, "WORK_GROUP_SIZE_X must be bigger than 0");
static_assert(WORK_GROUP_SIZE_Y > 0, "WORK_GROUP_SIZE_Y must be bigger than 0");
static_assert(HEIGHT > 0, "HEIGHT must be bigger than 0");
static_assert(WIDTH > 0, "WIDTH must be bigger than 0");
static_assert(NUM_OF_ITERS > 0, "NUM_OF_ITERS must be bigger than 0");
```

Project time measurement

In order to measure GPU time we start counting the “clFinish” function as shown below.

```
start = clock();

clFinish(command_queue);

end = clock();
time_sec = ((double) (end - start)) / CLOCKS_PER_SEC;
std::cout << "OpenCL solution takes: " << time_sec << " sec" << std::endl;
// read results
```

The reason is the documentation: “*All previously queued OpenCL commands in command_queue are issued to the associated device, and the function blocks until all previously queued commands have completed.* ***clFinish*** *does not return until all previously queued commands in command_queue have been processed and completed.* ***clFinish*** *is also a synchronization point.*”. So we can truly wait for the function completion to estimate time properly.

Solution

One of the first observed optimizations is using “__local” memory. We can allocate size to work group size and before each computation loads it. Also, it’s important to store data as constants. The local memory is shared between work items. On the figure 1 provided usage of constant memory usage and local memory.

```
const int i = get_global_id(0) + 1;
const int j = get_global_id(1);
const int local_id = get_local_id(0);
float tmp;
__local float r_local[WORK_GROUP_SIZE];
__local float g_local[WORK_GROUP_SIZE];
__local float b_local[WORK_GROUP_SIZE];


const int offset = i * w + j;
int pos = offset + 0;

int rpos = pos - w;
int bpos = pos + w;
```

Figure 1 - allocation of local memory and constant variables

One of the most effective optimizations - reducing “if” statements for checking borders. We can skip it since our NDrange by height decreased for 2(Figure 3). So before each work item starts working we increase i to 1. This guarantees we do not exceed the borders. By this optimization, we avoid such an important thing as divergence. The barrier is used to synchronize threads before the next step(Figure 2).

We keep collapsed memory access since our threads access memory in one row. The memory access is sequential and that is one of the reasons work size by x is 128 and by y is 1. The occupancy calculator for the current GPU approves 128 work size groups as one of the best. Also, proper attention was paid to the number of registers used per thread. It should not exceed 32, so the kernel satisfies that completely.

```
r_local[local_id] = r[rpos];
g_local[local_id] = g[pos];
b_local[local_id] = b[bpos];
tmp = native_rsqrt(b_local[local_id] + r_local[local_id]);
g_local[local_id] += tmp;
g[pos] = g_local[local_id];

barrier(CLK_LOCAL_MEM_FENCE);
```

Figure 2 - one iteration

In figure 3 we can observe that “HEIGHT” decreased by two(as discussed above) and WIDTH divided into 2. The reason is that the number of tasks is fixed - 2. It was tested (Figure 5, 6) that there is no difference between task sizes of 2 and 1. So 2 was fixed. Also it was observed that a task size of more than 8 is redundant since native_sqrt is a rather complicated function that takes time. Work group as was discussed above can be set up via conf.hpp file. Since NDrange by width divided into two, in kernel we should proceed cell i and i + WIDTH/2.

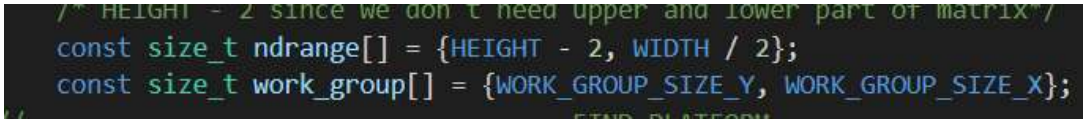

Figure 3 - NDrange and work group sizes

Tests with group size showed that indeed if the size is not 128 or 256(occupancy calculator), general performance falls down. That fact is represented in Figures 4 and 5. Also is work group size for Y less than warp_size * task_size (in our case 32 * 2 = 64), the performance also fails since access to memory will not be sequential and takes more transactions that might be achieved with proper parameters.

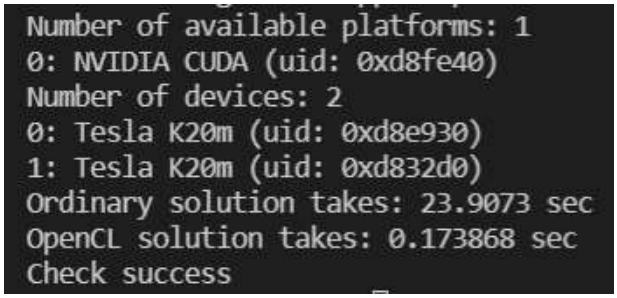

Figure 4 - speed for work group size by X=128, Y=1, tasks=2, iters=23

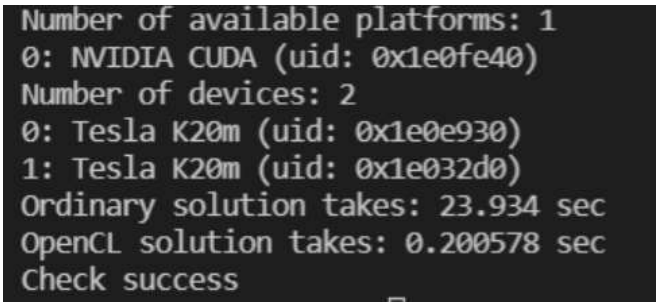

Figure 5 - speed for work group size by X=64, Y=1, tasks=2, iters=23

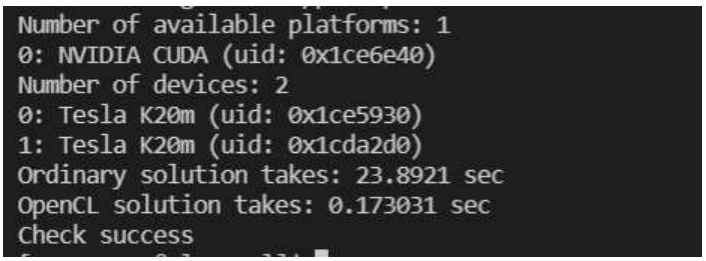

Figure 6 - speed for work group size by X=128, Y=1, tasks=1, iters=23

One of the possible optimizations would be memory address alignment. If the address is misaligned it may take several transactions to load the data. Before each memory allocation, we should strictly align memory to the required number of bytes. For example in our case, it would be 32. Also, we should care about task size. if task size is not multiple of task size, there will be some warps that will have no work items to proceed. Also the same is related to warp size and group size. For example, if warp 32 and workgroup size is 100, 3 warp will be full of work items and the 4th warp will run only 4 work items, so 28 threads will waste time. It would be difficult to avoid since there is no guarantee in workgroup execution order within NDrange.

It's easy to observe that constant data may be computed directly in the kernel. Moreover, it can be once counted and used via local memory for several iterations since "r" and "b" are constant over all iterations. We can see that constant data is reused so if it's stored within local data, one no more needs these huge arrays to be passed into the kernel.

One of the sufficient ways to optimize kernel performance on GPU is to utilize wider data types. Actually, it requires an increased number of "tasks" and uses vector operations. Unfortunately, NVIDIA GPUs don't have vector operations, for example adding and multiplication but in HW(hardware) they have special wide data types that support the load and store instructions both for the external and the local memory. Example of such kernel possible(not correct, just example of the idea) implementation provided above:

```
#define TASK_SIZE 4

__kernel void opencl_solution(__global const float *r, __global float *g, __global const float *b, const int h, const int w)
{
        const int i = get_global_id(0) + 1;
        const int j = get_global_id(1) * TASK_SIZE;

        const int off = i * w + j;
        float4 r4 = {r[off - w], r[off + 1 - w], r[off + 2 - w], r[off + 3 - w]};
        float4 g4 = {g[off], g[off + 1], g[off + 2], g[off + 3]};
        float4 b4 = {b[off + w], b[off + 1 + w], b[off + 2 + w], b[off + 3 + w]};

        g4 = native_rsqrt(r4 + b4);

        g[off] = g4.x; g[off + 1] = g4.y; g[off + 2] = g4.z; g[off + 3] = g4.w;

}
```