Exercise 6

hand out: 2021-01-20, hand in: 2021-02-03 14:00

Task 1: Mandelbrot on GPUs

30 Punkte

We have started with a simple Mandelbrot set implementation and used different techniques to improve the code's runtime on the CPU. Now we want to leverage the power of the GPU and make our calculations run even faster using OpenCL.

a) (5 P.) Setup the necessary OpenCL environment for your system. See below for a guide to install OpenCL on Ubuntu 20.04. Hand in the output of `clinfo` of at least one OpenCL device for this exercise. (Including only the entries up to `Max compute units` is sufficient.)

b) (15 P.) Implement the host code (this part runs on the CPU and controls the flow of the program) in `mandelbrot.cc`. The host code should load and compile your kernel, then run the kernel on your device to fill the image. Finally transfer the image back to the host and store it as png.

Hint: You might find the macros `CATCH_CL_ERROR` and `CATCH_CL_BUILD_FAILURE` defined in `cl_utils.h` useful. `CATCH_CL_ERROR(err)` will check the error code `err` returned by an OpenCL function. If `err` is not `CL_SUCCESS` it will print the error and the location of the error, before aborting the program. `CATCH_CL_BUILD_FAILURE(err, device, program)` works similarly, but will also print the compiler log if the compilation of your device sources failed.

c) (10 P.) Write the kernel which will calculate the color value for a pixel (this is the code which will be executed on the GPU in parallel). You can use the code provided in `mandelbrot.cl` as a starting point.

Setting up OpenCL

Setting up an OpenCL environment involves two parts. First you need to install the OpenCL libraries and headers. This is straightforward:

```
sudo apt install ocl-icd-opencl-dev clinfo
```

Second you need to install an OpenCL driver for your GPU. Ways to install OpenCL drivers for various vendors are described below as well as a fallback solution for virtual machines.

To check if your system is setup correctly, use the `clinfo` command. It should output various parameters for all devices you have installed OpenCL drivers for.

For troubleshooting refer to this guide.

NVIDIA

NVIDIA ships an OpenCL 1.2 driver with their CUDA toolkit:

```
sudo apt install nvidia-cuda-toolkit
```

Intel

The OpenCL driver for Intel's integrated GPUs of Broadwell systems and upward can be installed like this:

```
sudo apt install intel-opencl-icd
```

GPUs of older architectures might be supported by the legacy beignet driver:

```
sudo apt install beignet-opencl-icd
```

AMD

Support for OpenCL on AMD devices is available via the Mesa [1] OpenCL driver:

```
sudo apt install mesa-opencl-icd
```

This driver supports "only" OpenCL 1.1 but that should suffice for this exercise.

For newer GPUs (Fiji, Polaris, Vega) you can also try to install AMD's ROCm (Radeon Open Compute) platform to use more recent OpenCL drivers: `https://rocmdocs.amd.com/en/latest/Installation_Guide/Installation-Guide.html`

Fallback

If your GPU doesn't have an OpenCL driver or if you're solving this exercise in a virtual machine without GPU passthrough, you can use the POCL runtime [2] as a fallback:

```
sudo apt install pocl-opencl-icd libpocl2
```

POCL will enable you to compile and run OpenCL 1.2 code on your CPU.

[1] `https://www.mesa3d.org/`
[2] `http://portablecl.org/`

Task 2: Sequential consistency

6 Punkte

Assume a multi-processor system performing a sequence of memory read and write operations. The notation is as follows:

- Each column shows operations from a single processor. All operations read or write to the **same** address and are executed in order from top to bottom.
- $W_n(X)$ – processor n writes the value X.
- $R_n \rightarrow X$ – processor n reads and gets the value X.

Initially, all memory locations are initialized with value **c**.

Can a given set of operations occur on a sequentially consistent system? If so explain your answer by writing at least one possible sequential ordering of the given operations. If not show the contradiction.

Example:

$\mathbf{P}_1$	$\mathbf{P}_2$
$W_1(a)$	$R_2 \rightarrow a$
$W_1(b)$	

This sequence can occur in a sequentially consistent system.
One possible ordering is: $W_1(a)$, $R_2 \rightarrow a$, $W_1(b)$

a) (2 P.)

$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_3$
$W_1(a)$	$R_2 \rightarrow a$	$R_3 \rightarrow b$
$W_1(b)$	$R_2 \rightarrow b$	$R_3 \rightarrow a$

b) (2 P.)

$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_3$	$\mathbf{P}_4$
$W_1(a)$	$W_2(b)$	$R_3 \rightarrow b$	$R_4 \rightarrow a$
$R_1 \rightarrow a$			$R_4 \rightarrow b$

c) (2 P.)

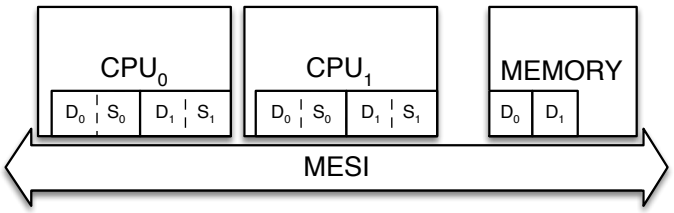
$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_3$	$\mathbf{P}_4$
$W_1(a)$	$W_2(b)$	$R_3 \rightarrow b$	$R_4 \rightarrow a$
		$R_3 \rightarrow b$	$R_4 \rightarrow a$

Task 3: Cache Coherence Protocol

15 Punkte

In this task we assume a multi-processor system with parallel caches.

- The system has two CPUs which are connected through a common bus to the memory.



- Each CPU has a direct-mapped, write-back cache with two cache blocks.
- Each cache block holds the cached data D_n and the state of the cache block S_n.
- The system uses the **M**odified - **E**xclusive - **S**hared - **I**nvalid (MESI) cache coherence protocol.
- The memory bus can execute one operation at a time.
- The bus supports BusRd, BusRdX and Flush operations as defined in the lecture.

The following table gives a sequence of serial processor memory operations (to the cache), where R_x means a read operation to address x and $W_x(C)$ means that the value C is written to address x.

	CPU_0	CPU_1
1	R_0	
2		R_1
3	$W_1(z)$	
4	$W_1(w)$	
5		R_0
6		$W_0(u)$
7	R_1	
8		R_1

Table 1: Sequence of operations

a) (10 P.) Fill the given table with the final states and values of the CPU caches and the main memory as well as the occurring bus transactions for each step in Table 1. You may split steps into substeps if it aids clarity.

b) (3 P.) Which signal is required in addition to the snooped bus transactions? Explain its purpose.

c) (2 P.) Name one advantage of the MESI protocol over the MSI protocol and point it out in the example below.

	CPU_0				CPU_1				Memory		Bus
	D_0	S_0	D_1	S_1	D_0	S_0	D_1	S_1	D_0	D_1	
0	–	I	–	I	–	I	–	I	x	y	
1											
2											
3											
4											
5											
6											
7											
8											