Computer Architecture Fall 2022
Assignment 4: Explicit Parallelism

Deadline: Friday, December 23, 2022 ~~Friday, December 16, 2022~~

Whereas in the third assignment we studied techniques to optimize single-thread performance, in this fourth, and final, assignment we will consider techniques for explicit parallelism. A number of parallel architectures are discussed in the chapters on Data-Level Parallelism (DLP) and Thread-Level Parallelism (TLP). This year, you will be adapting a number of programs in order to explicitly expose parallelism on GPUs.

Similar to the third assignment, this assignment is structured around a problem set, which involves the implementation of performance improvements, followed by experiments to analyze and quantify the effectiveness. As deliverables we again expect a report and the modified materials that were used to conduct the experiments. The tasks are built upon image processing kernels similar to those used in assignment 3. For completeness sake we included the brief explanation of some aspects of this framework again as Appendix A.

Background

Within this assignment you will write GPU implementations (exploiting Data-Level Parallelism) of image processing kernels using CUDA. Next to writing the required CUDA kernels, you need to add memory allocation and memory transfer functional calls. For problems 1 and 2 skeleton codes are provided. The skeleton includes a single image copy CUDA kernel as an example.

On BrightSpace you can find a concise introduction to CUDA programming. In the CUDA code the same image framework is used (Appendix A). Additionally, it is important to know that the `image_get_pixel` macro will work correctly for host as well as device pointers and that instead of the type `rgba_t` the type `float4` is used. Both types have the same `x, y, z, w` member fields. You may assume that the input images will always have dimensions that are a multiple of 64 (but are not necessarily square). This property will greatly simplify optimization, since many corner cases do not have to be checked and do not have to be dealt with.

To test the implementation, we have included a test mode in the template. This can be enabled with the `-t` command line option. An example usage of the 'grayscale' program (problem 1) is:

```
./problem1 -t /path/to/lab3data/color4096.png
```

This command will apply your 'grayscale' kernel to the specified image file name and compare the output of your GPU kernel to that of the CPU kernel (reference implementation). This way, you can test your implementation without repeatedly having to transfer the output image over SFTP. Once the test mode indicates the images match, we recommend you to perform a visual inspection of the result as well. By removing the `-t` option and adding `-r 5`, the program will measure the performance of the CUDA kernel and repeat this measurement 5 times.

The program for problem 2 ('tile composition') will be extended into a program that can process images in batches for problems 3 and 4. Therefore, it accepts a directory of images as input argument rather than a single filename. We have provided a directory with frames of a video stored as separate PNG files: `/vol/share/groups/liacs/scratch/ca2022/lab3data/frames/`. The second argument specifies an image file containing the tile to use.

The following command will run the tile composite kernel for all images found in the specified directory, one image at a time:

```
./problem2 /path/to/lab3data/frames/ /path/to/lab3data/tux64.png
```

Add the `-1` option to stop processing after 1 image. With the additional `-r 5` option, the computation is repeated 5 images. You can also control the batch size (useful for problems 3 and 4). The following command processes a single batch of 8 images and repeats this 3 times:

```
./problem2 -1 -b 8 -r 3 /path/to/lab3data/frames/ /path/to/lab3data/tux64.png
```

Finally, also the `problem2` program has a `-t` test option, which will only process the first image found in this directory and always runs with a batch size of 1.

If you would like to reconstruct a video from the frames in the output directory, refer to Appendix B.

To optimize the kernels, you will need to think about the potential performance impact of code changes made to the kernel. Carefully measure the execution time of the different variants and determine whether there is a significant difference. Profiling tools do exist for GPUs, however, through which low-level metrics can be collected. Up till CUDA 10, a tool called `nvprof` can be used to look at particular metrics to avoid optimization. For CUDA 11 and later, two different tools known as NSight Systems and NSight Compute are available. NSight Systems can be used to collect timing information about kernel execution and memory transfers. Low-level metrics about the compute kernels can be collected with NSight Compute (`nv-nsight-cu-cli`). Unfortunately, the low-level metrics cannot be collected on the LIACS computers due to security constraints. Therefore, we will not require such metrics to be included in your report. Still, if you have a CUDA-capable GPU available yourself, you can use these tools.

Environment

A suitable version of CUDA is not installed on the LIACS workstations by default. We have prepared installations of this software in the `ca2022` environment. This environment can be enabled as follows:

```
source /vol/share/groups/liacs/scratch/ca2022/ca2022.bashrc
```

Note that `remotelx` (`huisuil`) is *not* equipped with a GPU. However, many LIACS lab computers are equipped with an NVIDIA GPU. Exceptions are rooms 307 and 309 (which are not LIACS lab rooms) and a small number of workstations with a "No NVIDIA GPU" sticker.

Problem Set

Problem 1: Exploring Thread Block Configuration

For the first problem, you will work with the 'grayscale' kernel. The main goal is to become familiar with CUDA and to explore the influence of different thread block configurations on the performance. The file `problem1.cu` contains the skeleton code to use. As described above, it contains a test mode which compares the output to that of the CPU-version of the grayscale kernel. Make sure to use a color image as input! The first step (RQ1) is to implement a one-to-one translation of this CPU kernel to a CUDA kernel. You can verify correct operation of your CUDA kernel using the test mode. In the subsequent steps you will implement a number of variants to answer the following research questions:

- RQ1: Implement a CUDA kernel that processes 1 pixel per thread and uses a thread block size of 8×8. Measure its performance on images of increasing size and observe the scaling trend.
- RQ2: Write a variant that processes more than one pixel per thread. Does the performance improve?
- RQ3: Investigate the influence of different thread block configurations on the execution time of the kernel.

For the performance measurements it is important to repeat the experiments and to look at the processing time of individual frames/images. In this case 'compute time' only includes the time required to execute the CUDA kernels on the GPU. It does not include the time required to perform memory allocation and memory transfers, which is fine, because we will not consider it in this problem.

Problem 2: Shared Memory

We now investigate whether the use of the shared memory present in each Streaming Multiprocessing can help us to improve the performance. We will use the 'tile composition' kernel, which this time places the tile on the entire background image, instead of just the diagonal. Use `problem2.cu` as the skeleton program with the `-1` option to only process a single image (the batch size already defaults to 1). You can use the first of the provided 'frames' as the background image and a tile of 64×64 pixels. The GTX 1050 Ti present in the lab room computers has 49 KiB shared memory available per thread block.

- RQ: Investigate the effectiveness of placing the tile in GPU shared memory. Develop two variants of the CUDA kernel: one that does not use shared memory and one that does. Compare their performance.

When placing the tile in shared memory, do not already perform the pixel unpacking, so that we really investigate just the impact of using shared memory. Make sure to architect the kernel such that data loaded into shared memory is used multiple times – which should be feasible as the tile is placed onto the background multiple times.

Problem 3: Batched Processing

Up till now we have designed the CUDA kernels such that they only work on a single image. What if we process multiple frames per kernel invocation? This reduces the number of kernel launches, which might be beneficial. More in particular, we are interested in finding out whether this increases the effectiveness of the shared memory optimization.

- RQ: If we process multiple images in batches, does the effectiveness of the shared memory optimization increase? Create batched variants of both kernels from problem 2 (copy the file to `problem3.cu` first!). Measure their performance and compare the results. Do both show a similar improvement in performance, or does one improve more than the other?

The number of images in a batch is configurable using the `-b` command line option, as described earlier in this document. This option can be used to see the effect of different settings.

Problem 4: Overlapped Data Transfer / Compute

This final problem is based on your code for problem 3. So far, we have only assessed compute time and not the time required for data transfer. However, the overall execution time comprises both compute as well as data transfer time (for input and output!).

- RQ: Can the overall performance be improved by overlapping data transfers and compute when processing all frames of the video? Implement such a variant, and compare its performance to your program for problem 3. When running the programs, remove the `-1` option such that all frames are processed.

Make a copy of your program with the batched kernels from problem 3, name it `problem4.cu` and modify the code such that data transfer of the next batch of images happens at the same time as running the kernel for the current batch of images. Such functionality can be implemented using, for example, CUDA Streams. Note that the results also need to be transferred back to the

host. The skeleton code measures load time, compute time and overall execution time. The data transfer time is not included in the load or compute time, but is included in the overall execution time. So, by properly implementing the overlapped data transfers, we expect the overall execution time to decrease.

Submission and Grading

Teams may be formed that consist of at most *two* persons. The following needs to be submitted:

- Source code of the implemented optimizations. It *MUST* be based on the starting point (template) of this academic year, otherwise your submission will not be graded.

  Note again: we want to see the code for each task individually. And please ensure to not include binaries, object files or image files in your submissions to keep the size of the file archives manageable. Use `make clean`! If the size of the tarball containing your source code is larger than 250Kb, you probably did something wrong.

- A well-organized report (preferably written in LaTeX) in PDF-format. Refer to the guidelines as noted in the description of assignment 3. For this particular assignment it is also important to mention GPU model and CUDA version used for implementation and experiments.

Ensure that all files that are submitted (source code, report, etc.) contain your names and student IDs! The source code tarball and PDF-report are to be submitted as *separate* files. Please create a "gzipped tar" archive of your source code:

```
tar -czvf assignment4-sXXXXXXX-sYYYYYYY.tar.gz assignment4/
```

Substitute `XXXXXXX` and `YYYYYYY` with your student IDs.
Use the filename `report-sXXXXXXX-sYYYYYYY.pdf` for your report.
Submit the tar-archive and your report through the BrightSpace submission site. Also note your names and student IDs in the text box in the submission website. *Please, ensure only one team member submits the assignment, such that there is a single submission per team!*

Deadline: Friday, December 23, 2022 ~~Friday, December 16, 2022~~

The maximum grade that can be obtained is 10. The grade is the sum of the scores for the following components, maximum score between square brackets:

- **[2.0 out of 10]** Overall quality of the report.
- **[3.0 out of 10]** Problem 1.
- **[2.0 out of 10]** Problem 2.
- **[1.5 out of 10]** Problem 3.
- **[1.5 out of 10]** Problem 4.

All submitted reports and source code will be subjected to (automatic) plagiarism checks.

A Some details on the image operation kernels

The two-dimensional images are stored in a one-dimensional array. To find the pixel in memory for a location (x, y) the formula $A[x + W * y]$ (column-major) or $A[y + H * x]$ (row-major) is used. In the file `image.h` the macro `image_get_pixel` is defined which will return a pixel to the requested pixel. The macro `image_get_pixel_data` accomplishes the same, but allows you to specify the data pointer and rowstride directly.

Image arrays contain elements of the type `uint32_t` and each element contains a single pixel. So, each pixel is stored in 32 bits. Within this assignment we are using the `RGBA8888` pixel format. This means that each pixel consists of four channels that take eight bits each: red, green, blue and alpha. Red is stored in the most significant byte and alpha in the least significant byte. In order to extract the value of the red channel, the following code can be used:

```
uint32_t pixel = *image_get_pixel(image, x, y);
int red = ( pixel >> 24 ) & 0xff;
```

The alpha channel indicates the opacity and must remain `0xff` in most cases. Most image operations are performed in the floating-point domain. However, our image arrays consist of integer elements. To solve this, pixels are unpacked and packed when required. Packing is the encoding of a vector $\langle r, g, b, a \rangle$ as a single integer. Unpacking is the inverse operation. The reason why the operations are performed in the floating-point domain is that in the floating-point domain all channels have a 32-bits precision, while this is only 8-bits within the packed integer domain. Within `image.h` a number of useful macros is defined that simplify these operations. The following example illustrates these macros:

```
/* Retrieve a pixel from A, multiply this with a constant and write back
 * to the image array.
 */
uint32_t pixel = *image_get_pixel(image, x, y );
rgba_t pixel_rgba, factor, result;

RGBA(factor, .5f, .5f, .5f, 1.f);
RGBA_unpack(pixel_rgba, pixel);
RGBA_mult(result, pixel_rgba, factor); // result = pixel_rgba * factor
RGBA_pack(pixel, pixel_rgba);

*image_get_pixel(image, x, y) = pixel;
```

The type `rgba_t` is used to represent a vector consisting of four float components. These components can be accessed individually using the member fields `x, y, z, w`, which correspond in that order with the channels r, g, b, and a. So, `my_rgba.z` gives the blue channel value.

B Reconstructing a video from individual frames

The result of the program will be a directory containing the modified frames. If you want to see the result as a video, you can use the command (ffmpeg must be installed):

```
ffplay -framerate 30 -i frame%04d.png
```

You can also reassemble the modified frames into a new MP4 file. This can be achieved with the following command:

```
ffmpeg -r 30 -f image2 -i frame%04d.png -vcodec libx264 -crf 25 \
    -pix_fmt yuv420p -s 1920x1024 test.mp4
```

After completion of this command, open `test.mp4` with a video player.