TDT4200-PS6
CUDA Graded - Image Morphing

Jacob O. Tørring and Zawadi Svela

Deadline: Wed November 10, 2021 at 22:00 on BlackBoard
Evaluation: Graded (15% of final course grade)

- **This assignment must be done individually and w.o. help from any one but The TDT4200 staff.** We encourage that you post clarification questions on the Forum in BB so all can benefit. All sources used found on the internet or otherwise must be referenced
- **Grading: This assignment is graded and counts towards your final grade in this course.** The problem set counts for 15% of your final grade and must be passed to take the final in the course.
- This assignment has 3 parts. You will need to get the correct output using your CUDA code in order to get a pass for a given part. Note that:
  - 10% of the total score hinges on the codes submitted being clear and well documented
  - **Completing Part 1 is mandatory to pass** the assignment and counts up to 60%
  - Part 2 Shared-memory counts 20%.
  - Part 3 Occupancy will give additional percentage points up to 100%.
  - Partial solutions for Part 2 and 3 will be given some percentage points if the intent behind the code is clear and well-documented.
- For Part 2 your shared-memory version should not give a significant slowdown (more than 20%). A speedup is not necessary for Part 2 or Part 3, only correct usage of the optimization techniques.
- Code should not use other external dependencies aside from the ones we specified. Do not deliver any other files than those specified in the Deliverables section.

Image Morphing with Beier-Neely using CUDA

In this exercise you will parallelize a pre-written serial program for image morphing that utilizes the Beier-Neely algorithm, for the GPU.

The Beier-Neely algorithm is a process of morphing an image. Given an image and a set of feature lines, each meant to outline some feature in the image, like the angle of a mouth, the position of a jawline, ear, etc., the algorithm creates a new image for a new set of positions for these lines. The process of creating the morphed output image is to for each output-pixel determine a single position in the input to extract color values from. In short, for each pixel, we look at *all* the lines in the out-image to determine their relative closeness, and then use that information to find a corresponding position in the in-image to extract our color from. So the morphed image will look similar to the source-image close to each line, while looking more warped elsewhere.

A common use-case for this algorithm, and the one we will implement here, is to create a series of morphed images to transition from one image to another. Two input images, each with their corresponding feature lines, are morphed towards a set of equal line positions. If we calculate those common position as a weighted average of the two input sets, and fade the images in and out in a similar fashion, we can create a continuous transition, where one image warps into another.

Writing the baseline CUDA version of Beier-Neely accounts for 60% of the grade in the assignment. You must complete this part correctly to pass the assignment. Our solution gives more than 300x speedup(10 steps of man9-man10) compared with the serial implementation. You will then use shared memory to optimize the memory performance of the program (20%). Lastly, we optimize the selection of block sizes using the CUDA Occupancy API (10%). The remaining 10% of the points are given based on code clarity and documentation.

Important: When you deliver your code, make sure that you only deliver the files explicitly asked for. This will only include code you have edited yourself, and not binary files, example input, etc. A consequence of this is that *you are not allowed to utilize external libraries not already included in the hand-out code.*

Setup

For this problem set we recommend using the PCs at the Cybele Lab or the Snotra compute cluster as described under Course Information.

Before you begin the tasks, you should start by testing the program. Compile the serial implementation using Make and run the program.

```
./morph images/man9.jpg images/man10.jpg lines/lines-man9-man10.txt output-png/ 10
```

This should create a series of images in the output-folder, each morphed some way between the source and destination images. Note down roughly how much time each call of morphKernel takes and the total time.
You can then start to implement your CUDA program and compile it using "make". A correct implementation of the CUDA program should be at least 300x faster on the Nvidia T4 GPUs on Selbu compared with the serial implementation (for man9-man10 with steps=10, i.e. the example above.).

You can also use cuda-gdb to debug your program similarly to gdb, and cuda-memcheck to check for memory errors.

Tasks

The following tasks are all marked with corresponding TODO comments in the handout code. After each task, you should ensure that the program compiles, though it might not run after some of the changes.

Part 1: Baseline CUDA implementation (60%)

A (10%) Use cudaMalloc to allocate space for data in device-side memory. Use cudaMemcpy to copy the data from the variables containing the host-side images to the device-side variables.

B (10%) Define the block size and grid size. Until Part 3 the block size should be defined as 8x8 (64 threads in total per block). The grid should be defined as a function of the block size and output image dimensions. There should then be a total number of threads in the grid matching or exceeding the total amount of pixels. Each pixel can then be mapped to an individual thread. Change the morphKernel function to be a Kernel with launch parameters, using gridSize and blockSize. You need to change the code both where the function is defined and where the function is called in main (marked by TODO 1 b).

C (15%) Parallelize the double for loop in the morphKernel function. Use the threadIdx.x, blockIdx.x and blockDim.x syntax to get the global index of the current thread in the x dimension
(i.e. int x = threadIdx.x + blockIdx.x $*$ blockDim.x). Use the index in both x and y dimensions. Make sure to include a boundary check to check if the index is within the bounds of the image.

D (10%) Copy the device-side data back to the host-side variables using cudaMemcpy. Free the malloced variables using free() and cudaFree().

E (15%) Optimize the CUDA API Calls by moving any eligible cudaMalloc and cudaMemcpy calls from TODO 1.a to TODO 1.e.
Optional: Run the nsys-profile.sh script. Use "nsys stats report1.qdrep" to see how much time is spent on different CUDA API calls. Is there a way to reduce the number of calls? Move the relevant API calls. Compare the difference. There is no need to report the results in your submission.

Part 2: Shared-memory (20%)

Use Dynamic Shared-memory [1] to optimize memory performance.
Note: For some GPUs, shared-memory might not give a measurable speedup.

A (3%) Define the shared memory size.

B (2%) Change the launch configuration of the kernel with the appropriate shared memory size.

C (15%) Change the kernel to use shared memory where applicable.
Optional: Compare before and after with the ncu-MemoryWorkloadAnalysis.sh script in the handout. How did the memory performance change between before and after?

Part 3: Occupancy (10%)

Note: For some GPUs and kernels, the largest possible block size might not always give optimal occupancy/performance [1].

A (5%) Use the Occupancy API [2] to get a maximum potential block size.

B (5%) Subdivide the block size into two dimensions. Use these values for the block size of your kernel.
Optional: Compare before and after with the ncu-Occupancy.sh script in the handout. How did the achieved occupancy change compared with the theoretical maximum?

C **Optional** (0%): The Occupancy API is based on the assumption that the largest possible block size gives the best performance for a kernel. Sometimes this is not the case (see work by former MSc student at HPC-Lab [1]). Try to programmatically test out different values for the block size and find a block size that maximizes performance.

Deliverables

Deliver the finished morph.cu file on Blackboard. It should run error-free and produce the correct output. **Do not zip the file.**

References

[1] Lars Bjertnes, Jacob O. Tørring, and Anne C. Elster. LS-CAT: A Large-Scale CUDA AutoTuning Dataset. *arXiv:2103.14409 [cs]*, March 2021. arXiv: 2103.14409.

[1] https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/
[2] https://developer.nvidia.com/blog/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/