TDT4200
PS 6 (Optional) - Advanced topics in CUDA

Jacob Odgård Tørring

Due Friday, Nov 20, 22:00, 2020, or over New Year´s (*)

In this assignment, you will build upon PS 5, and optimize your solution using advanced CUDA techniques.

(*) This is an optional Pass/Fail assignment. For those that have missed a previous assignment, you may with instructor approval replace this assignment with a missed one, by the Nov 20 due date. Note: You only need to do one of the three tasks to pass the assignment.

WARNING

All of the tasks in this assignment are a considerable amount of work, so you will be given points for partly working solutions as well. We believe that the Floating Point or Cooperative groups tasks are the easier of these three tasks.

You should also solve one of the three tasks proposed in this if you want to get a recommendation from Elster for international studies or for other reasons. For recommendation purposes, you may turn this in by January 4, 2021. Your solution to this assignment(and previous assignments) will be used to give the recommendation.

Access to hardware: Snotra cluster

If you do not have a laptop or PC with an NVIDIA card, you can log onto the Snotra cluster and get an NVIDIA GPU allocated:

ssh username@snotra.idi.ntnu.no.

See the previous posts for Cybele and Snotra under "Course Zoom URLs & information" on Blackboard for more tips on how to use this interface.

If you end up on a host called "minip" you need to run **connect**. This shell is upon for 2 hours before it automatically closes. You can reconnect when the shell closes if you need more time. You can also request a specific node by specifying the name of the node:

ssh -t username@snotra.idi.ntnu.no connect nodename

Current list of nodes (subject to change):

selbu: 20 T4s:

1 GTX1080 each: clab02, clab04, clab06, clab09, clab12, clab15, clab18, clab21, clab23, clab25

The Code

We recommend that you use the code you developed during PS5. Please contact us if you did not complete PS5.
NB!: Carefully read the section "Deliverables" on how to submit your results!

Task A Cooperative groups

Even with shared memory, every pixel is read at least once every iteration. CUDA 9 introduced cooperative groups that can be used to synchronize threads across thread blocks. More information on their use here. For an example of use, which includes probing the GPU for information about the number of blocks that can be run on each SM, see this sample from Nvidia.

Note that the whole cooperative group has to be able to be run simultaneous across the SMs. The main limitations for our application is register usage and shared memory since **in order not to synchronize with global memory, the picture manipulated needs to fit in the SM's shared memory.** Note that there is a large difference in no. of SMs in, say NVIDIA TX2 vs GTX 1080 (20SMs) vs Tesla V100 (80SMs).

Task B Data type optimizations: (Half-)Floats instead of Unsigned Chars.

Many operations in GPUs are faster for floating point operations compared with integer operations. Converting the data types in the pixel structs to floats (or packed half-floats) will therefore give a considerable speedup, depending on the hardware being used. The objective in this task is thus to change the data types to floats and ensure that you get the same final image, but with greatly improved performance. NOTE: You might not get a speedup for all hardware, so test your code on selbu and clab. When you have successfully generated a correct image with floats you should implement packed half-floats(half2) to improve the performance further. Documentation for performing operations on packed half-floats can be found here.

Task C Tensor cores and convolution as GEMM

The problem for this task is re-organized to better fit the traditional uses of tensor cores for image convolution.

In PS5 you were given a set of filters F and would perform a convolution given a filter $f \in F$ over an image i times, thus producing a convoluted image. In this task you are to produce $|F|$ convoluted images by using $|F|$ different filters, as quickly as possible. I.e. emulating the behavior of running your program from PS5 for each of the filters $f \in F$ with a corresponding output **img_f.bmp**. You may choose the selection of F filters, including defining your own if you see fit, or use the 5 3x3 filters already available.

By transforming the problem to a GEneral Matrix Multiplication(GEMM) we can efficiently convolute all of the filters over the same source image simultaneously. This is done by transforming the images to column matrices(im2col) and multiplying them with a matrix of flattened filters before converting them back to images (col2im). Full explanations of im2col, col2im and how to structure this problem as a GEMM can be found many places online, including the original paper from Nvidia here. A simple implementation of im2col and col2im can be found here. You will need to adapt this code or the image array to access all 3 channels (r,g,b) efficiently for the GEMM.

cuDNN and all major deep learning frameworks (Tensorflow, PyTorch, etc.) use this approach for Convolutional Neural Networks. By transforming the convolution into a GEMM, we can tap into heavily optimized GEMM implementations to solve the task. This also enables us to use Tensor Cores to perform the GEMM. For a high-level introduction to Tensor cores, see this link. For detailed documentation about the wmma api to use tensor cores, see this link.

The necessary changes to the program are thus:

1. Specify the filters you want to use for the task(e.g. the 5 3x3 filters already available)
2. Copy your shared-memory implementation and change it to convolute over all of the selected input filters and output the corresponding images. You can then compare against this solution later.
3. Create a new kernel for the GEMM implementation, which needs the same set of images and filters as the previous kernel, but you might want to restructure the data arrays.
4. Transform the images into column matrices by implementing and using im2col. (Advanced: Or do this implicitly during runtime, to save memory).
5. Compute the GEMM using wmma (tensor cores).
6. Transform the column matrices back to images by implementing and using col2im.

7. Save each of the output images in separate files and compare the correctness against your implementation from Task 1.

8. Compare the performance between the multi-filter shared-memory kernel and your GEMM-based multi-filter kernel. Is it faster or slower? Why do you think this is the case?

Make sure that you compile for sm75 and that you're running on a GPU that has Tensor Cores (e.g. selbu on the Snotra cluster).

Some hints for Task A, useful information for CUDA programming in general – and final preparations:

Memory usage

In order to have all blocks active at once, we need two things to be true:

1. The total shared memory usage per block times the number of blocks per SM must not exceed the total capacity of the SM.

2. The total register usage per block times the number of blocks per SM must not exceed the total size of the register file of the SM.

Luckily, there is a straight-forward way to check if these conditions are met by calling the function: `cudaOccupancyMaxActiveBlocksPerMultiprocessor()`

Example of its usage can be found in the cooperative groups sample code linked above. In short, this function takes the kernel you want to run, the size of your blocks in no. of threads and the shared memory usage per block, and gives you the number of blocks you can run per SM. If this number is more than or equal to the number you need, everything is fine, otherwise, back to the drawing board. Note that you need to include all of your shared memory usage, including the filter and perhaps an extra buffer for the out image if you want both in and out images in shared memory.

Note also that TASK A will be significantly more memory hungry than our baseline solution, so there are a couple of new strategies you might need to employ:

1. In order to make sure the shared memory usage is small enough, you could simply state that you program does not work for images exceeding a certain size – **do remember to state this in the delivery!**

2. In order to get the overall register usage down, however, you need to reduce the per thread register usage. This can be done with thread coarsening, having each thread work on several pixels.

Once the system accepts your memory resource use, start by running a single iteration. There are many more variables in play, so there is no need to overcomplicate it, so start small and clean.

Grid synchronization

When you get to the point where you are able to run multiple iterations within the kernel, it is time for the actual grid synchronization. Write the borders (those are the only values shared between blocks) into global memory, swap the in and out buffers (both presumably in shared memory), sync the grid and go again. After all the iterations are done, write all values back to global memory.

Lastly: Synchronization across blocks is a relatively new feature, and might not be available on the architecture the compiler selects by default.

Use the `-arch=sm_60 -rdc=true` options to make sure that everything compiles correctly.

Deliverables

The following tasks require documentation (if solved):

- Explanation for how to solve Task A or B or C
- Partial or full solution to Task A or B or C

The documentation must be delivered as a PDF file together with the source code packed in a ZIP file and submitted in time on blackboard. The code must be documented. Do not use any libraries apart from standard libraries or those suggested in the assignment text.

Note:

Deliver your code in the folder structure that is provided to you. Include at least every file that you have changed. Your application must stay compatible with the provided Makefiles or CMake files.

Do not include any object, input or output files.