Homework #2 (HW2)

Autumn 2022 - GPU Computing

DUE: 24 October 2022, 5:59 PM

Learning Objectives

- implement 2D gaussian blur CUDA kernel variants
- use shared memory to store and access the blur filter coefficients
- use Nsight CUDA debugger to inspect breakpoints in the kernel execution threads
- use Nsight Compute to profile kernel execution
- Use Host timing functions to measure host and device runtimes
- compare theoretical performance metric estimates to empirical measured values

Overview

The HW2 includes two main parts: (i) answering some questions designed to reinforce your technical understanding of the material learned in the lectures and reading set, and (ii) a programming section where you will practice working with 2D CUDA grids, shared memory on the device, and the Nsight VSE debugger and Nsight Compute profiler tools.

Questions

Turn in written (or typed) solutions to these problems, including showing your work and/or describing your reasoning/explanation for the selected answer.

1. Assume that a kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?
    a. 1
    b. 1,000
    c. 512
    d. 512,000

2. For our tiled matrix-matrix multiplication kernel, if we use a 32X32 tile, what is the reduction of memory bandwidth usage for input matrices A and B?
    a. 1/8 of the original usage
    b. 1/16 of the original usage
    c. 1/32 of the original usage

d. 1/64 of the original usage

3. For the tiled single-precision matrix multiplication kernel as shown in lecture, assume that the tile size is 32X32 and the system has a DRAM burst size of 128 bytes. How many DRAM bursts will be delivered to the processor as a result of loading one A-matrix tile by a thread block?

a. 16
b. 32
c. 64
d. 128

Programming Exercises

[Preliminaries]

1. Begin from a new default CUDA visual studio solution & project (like you created in EX1 and HW1).
2. In VS22, In the Solution explorer, right click the kernel.cu file. Select Remove option
    a. Choose the Delete option to permanently delete.
3. From the class canvas website, go to Files/Homeworks/HW2
    a. Download the `hw2_starter.cu` file and put it in the project folder on your AppStream instance drive
    b. Also download the test images and utility code files
        i. hw2_testimage1.png (240x240)
        ii. hw2_testimage2.png (512x512)
        iii. hw2_testimage3.png (850x850)
        iv. hw2_testimage4.png (4023x3024)
        v. stb_image.h
        vi. stb_image_write.h
4. In VS22, right click your project name and choose Add->Existing Item
    a. navigate to your project folder and select the hw2_starter.cu file. Click Add.
        i. note you may have to manually type C:\ to reach the C: drive root in the Windows File Explorer

[CUDA Blur Kernels]

5. Follow the discussions in lectures #1 and #2 to complete the tasks outlined in the commented sections provided in the `hw2_starter.cu` file.
    a. when loading input image files

   i. use the provided image read/write header files (STB) and the provided example source code snippets in the starter *.cu file
   ii. **NOTE**: the input source images will be 8-bit monochrome, and thus the image datatypes both on host and device will use unsigned char (uchar).
      1. main internal pixel accumulation routine inside the kernel should use full INT (32 bit integer) type to avoid overflow, then convert back to a uchar when writing final normalized output pixel value.
   iii. be sure to check the pixel bit depth and use it appropriately
      1. you can double check by right-clicking the image file in File Explore and choosing Properties->Details tab, and review the Image Width, Height, Bit depth values. These should agree with what stb_load( ) returns when the image is loaded.

b. In your blur kernels
   i. use the 2D row and col thread indices as shown in Lecture2
   ii. check boundary conditions so that threads don't attempt to process or access out-of-bounds memory regions
   iii. each thread will write to a single pixel in the output image
   iv. HINT: refer to lecture 2 slide discussion of blur kernel
   v. HINT: suggest getting the basic non-Gaussian blur kernel as provided in lecture 2 slides working first
      1. test with a blur width = 10 to see clear blur effects
   vi. Use the provided 9x9 Gaussian filter coefficients array.
   vii. The gaussian blur filter coefficients will still be float values as provided, and when multiplied by the corresponding input pixel value the result should be rounded to nearest INT value.
   viii. In the gaussian blur kernels, no longer normalize the final output pixel value by the total # of pixels summed over, but instead by the sum of the applied filter coefficient weights - this will give a more appropriately weighted normalization factor.

c. Implement 3 versions of the blur kernel function:
   i. one which uses only global device memory
   ii. one using static shared memory allocation method to put the blur filter coefficients in shared memory
   iii. one using dynamic shared memory allocation method to put the blur filter coefficients in shared memory
   iv. NOTE: you will need an additional kernel argument to pass the blur filter coefficient array to the device global memory
   v. For the shared memory kernels, have the first N threads of each block collaboratively copy the blur filter coefficients into device shared memory array.
   vi. a synchronization point using `__syncthreads( )` will be required

d. refer to code examples provided in the lecture slides and also consult the PMPP textbook.

[Host-side Code Execution Timing]

6. #include and use the Windows profiling api (profileapi.h) to add high-performance timers to the host code. Define the necessary variables and add code to the commented locations to start and stop timer instances.
    a. Refer to the Lecture 3 slides for details on declaring, defining, and using the Windows high-performance timers API.
    b. The windows high-performance counter api header file is located in
        i. C:\Program Files (x86)\Windows Kits\10\Include\10.0.19041.0\um
            1. profileapi.h
7. Execute timed runs (not running in debugger as that will artificially adversely impact the performance!) for each of the provided test images 1-4.
    a. Save the time result values for each of the timers, for each test image case.

[NSight VSE Debugger]

8. When using the NSight VSE Debugger in VS22 to step through kernel code running on GPU, you will often have many warps and threads running simultaneously on the device. Thus your debugging will be constantly context-switched between threads as you step from one line of code to the next. This will make it extremely difficult to evaluate and debug your code behavior. There are (at least) two ways to address this situation.
    a. Launch a single-thread kernel execution configuration from the host.
    ```
    // in host code
    your_kernel_function<<<1,1>>>(d_arg1, d_arg2, … )
    ```
        i. this will prevent multiple threads from interrupting each other, however it may not allow debugging of synchronization/race/deadlock and other multi-thread-specific parallel execution issues.
    b. Add `if( threadIdx.x == brk_value) { … }` logic into your kernel and put a breakpoint inside the brackets
        i. this breakpoint will only be hit if the if condition evaluates to true
    c. Attempt to use the Conditional Breakpoint option to set conditions to break only on specific blockIdx and threadIdx values.
        i. Warning - I have not been able to get this to work correctly
9. Inside one of your blur kernel implementations, set two breakpoints on adjacent lines of code (for example lines 128 & 129 --- note the specific lines of code on your system will be different, the point is there are two breakpoints on adjacent lines)
    a. ensure your host kernel launch execution configuration is defining sufficient blocks and threads to cover the whole input image domain.
10. Launch the Nsight CUDA debugger (Next-Gen)
11. Once debugger stops inside your kernel function,
    a. check the values of the col & row variables (or equivalently the current blockIdx.x, blockIdx.y, threadIdx.x, and threadIdx.y built-in vars.)
    b. click Step Over (F10) option to move to the next line of code

        i. more likely than not, your code will not advance but will remain on the same line. This is not an error.
        ii. re-check the values of the col & row variables (or equivalently the current blockIdx.x, blockIdx.y, threadIdx.x, and threadIdx.y built-in vars.)
            1. they will *probably* be different (we say "probably" here because the execution of threads on the device is non-deterministic so we can't actually guarantee what is going to happen for a given instance!)
    c. click Step Over (F10) again
        i. again, your code may not advance to the next line, but you'll see the row & col values change.
            1. this is because multiple threads are hitting the same line of code (in different kernel instances) before any single kernel thread can actually move to the next line.

12. Use the Extensions->Nsight->Windows menu to open several of the provided debug windows
    a. Lanes
        i. You will see full warps (32 threads each) for nearly all cases (except for test images 3 & 4).
    b. Warp Info
        i. this view shows all threads for each warp which is current executing on an SM.
        ii. QUESTION: How may warps are running? Based on the Tesla T4 hardware, can you explain this number in terms of hardware resources?
    c. NOTE: you can click on a thread in a lane (Lanes view) and a warp (Warp Info view) to navigate to that specific thread of execution.
        i. Do this, and check the col & row values as you change between threads.
    d. Take a screen capture showing the Lanes and Warp Info windows open inside an active debugging session for the hw2_testimage1.png (240x240) case.
13. Stop debugging, go back to your host code and modify the kernel launch configuration to use the single-thread execution configuration <<<1,1>>> shown above
    a. relaunch the CUDA Debugger and hit the breakpoints,
    b. Open the Lanes and Warp Info windows and take another screen capture showing the current device execution configuration.
    c.

[NSight Compute Profile]

14. The NSight Compute Profiler tool, which comes with the NVIDIA CUDA Toolkit tool set, has both a graphical and a command-line interface (CLI). It is a powerful and complex profiler for kernels running on NVIDIA GPUs which can collect many hundreds of hardware metrics providing deep insight into how the kernel is performing during execution on the device.
    a. NOTE: NSight compute only supports devices of architecture Volta GV100 and later

i. https://docs.nvidia.com/nsight-compute/ReleaseNotes/index.html#gpu-support

b. For earlier devices, there is another profiler, `nvprof`, which supports most of the same profiling metrics, although the names and syntax is different.

15. In this HW2 we'll use the CLI to collect a basic set of metrics
16. We will continue to study and use Nsight Compute extensively during the remainder of the quarter, and in future GPU classes.
17. Do a Release, x64 build of your HW2 host application, configured to launch only one blur kernel function instance in the host main( ) function
    a. use the global-memory blur kernel function.
18. Open a command-prompt with Admin permissions.
19. Change directory to your Release build output directory.
    a. There should be a copy of the built executable .exe host application.
20. Execute the following command line. Replace the kernel function name and the (release build) host application EXE name with the correct values for `<your_kernel_function_name>` and `<your_host_application_name>`:

```
ncu -k <your_kernel_function_name>_globalmem --metrics
smsp__cycles_elapsed.sum,smsp__cycles_elapsed.sum.per_second,sm
sp__inst_executed.sum,dram__sectors_read.sum,dram__sectors_writ
e.sum,dram__bytes.sum ./<your_host_application_name>.exe
```

21. You should see output in the command console like the following

```
C:\Temp\AUT22_GPUCompute\x64\Release>ncu -k gaussBlurKernel_globalmem --metrics
smsp__cycles_elapsed.sum,smsp__cycles_elapsed.sum.per_second,smsp__inst_executed.sum,d
ram__sectors_read.sum,dram__sectors_write.sum,dram__bytes.sum ./HW2_starter_img3.exe
==PROF== Connected to process 8768
(C:\Temp\AUT22_GPUCompute\x64\Release\HW2_starter_img3.exe)
==PROF== Profiling "gaussBlurKernel_globalmem": 0%....50%....100% - 1 pass
==PROF== Disconnected from process 8768
[8768] HW2_starter_img3.exe@127.0.0.1
  gaussBlurKernel_globalmem(unsigned char *, unsigned char *, float *, int, int, int),
2022-Oct-17 19:41:10, Context 1, Stream 7
    Section: Command line profiler metrics
    ------------------------------- --------------- ------------------------------
    dram__bytes.sum                 Kbyte                           221.92
    dram__sectors_read.sum          sector                           6,935
    dram__sectors_write.sum         sector                               0
    smsp__cycles_elapsed.sum        cycle                        4,881,792
    smsp__cycles_elapsed.sum.per_second  cycle/nsecond             93.59
    smsp__inst_executed.sum          inst                        2,385,000
    ----------------------------------------------------------------------
```

22. Edit your host application code to use each of the 4 test input images, and do a separate Release build for each. Suggest you use a naming scheme such as hw2_rel_global_img1.exe, hw2_rel_global_img2.exe, etc.
    a. So that they don't get overwritten when you do a new rebuild.

23. Re-run the NSight Compute CLI command line profiler (NCU) as above, for each of the application builds which use a different size input image.
    a. Using the same global-memory blur kernel instance for each.
24. Collect all the NCU output command line metrics for each test image case
25. Repeat all above steps for the shared-memory blur kernel.
26. For each profiling run (there will be 8 total: 4 images each for global & shared memory kernel instances), compute the
    a. **kernel execution time** = smsp__cycles_elapsed.sum / smsp__cycles_elapsed.sum.per_second
        i. *convert from nanoseconds to seconds*
    b. **kernel instruction intensity** = smsp__inst_executed.sum / dram_bytes.sum
        i. *convert to bytes from Kbytes*

SUBMIT

1. Submit answers to the questions, including shown work and explained reasoning.
2. Submit your CUDA host application source code
    a. .cu, .cpp and .h files only - do not submit all project & solution files and other build by-products!
3. Submit your CUDA kernel functions
    a. may be in the same file as your host code if using a single *.cu file
4. Submit examples of the result blurred test images #1,2,3
5. Submit timing results for GPU CUDA blur kernel variants
6. Submit screen shots of CUDA kernel debugging session windows
7. Submit your NCU profiling results for all test image cases and global + shared kernels
    a. also include your calculated kernel execution time and kernel instruction intensity values.
8. EXTRA CREDIT
    a. submit timing results of your host CPU sequential blur function compared to your GPU CUDA kernel timing results.