

Assignment III: CUDA Basics

New Attempt

Due 16 Dec 2022 by 23:59 **Points** 4 **Submitting** a file upload
File types pdf

To submit your assignment, prepare and upload a PDF file fulfilling the following requirements:

- Named as *DD2360HT22_HW3_Surname_Name.pdf*
- Include your answers to each exercise
- Add your codes to your previous Git repository *DD2360HT22*. Include the link in your report.
- Use the following folder structure in this Git repo : *hw_3/ex_ExerciseNumber*
/your_source_code_files

Important note: This assignment will need knowledge of CUDA programming. You will find lectures, including the introductory and video lectures, tutorials, and the reading materials useful for this assignment.

Exercise 1 - Your first CUDA program and GPU performance metrics

This exercise will get you familiar with the basic structure of a simple CUDA code. You will learn how to compile and run CUDA-based programs. You will practice memory allocation in host memory and device memory, the data movement between CPU and GPU, and the distribution of the computational threads in CUDA. You will also use Nvidia Nsight (see the tutorial on Nsight profiling https://canvas.kth.se/courses/36161/pages/tutorial-nvidia-nsight-systems-for-profiling?module_item_id=582434 (https://canvas.kth.se/courses/36161/pages/tutorial-nvidia-nsight-systems-for-profiling?module_item_id=582434)) to profile your program.


When comparing the output between CPU and GPU implementation, the precision of the floating-point operations might differ between different versions, which can translate into rounding error differences. Hence, use a margin error range when comparing both versions.

Please implement a simple *vectorAdd* program that sums two vectors and stores the results into a third vector. You will understand how to index 1D arrays inside a GPU kernel. Please complete the following main steps in your code. You can create your own code, or, use the following code template (**Download Code Template Here** [lab3_ex1_template.cu](https://canvas.kth.se/courses/36161/files/6009924?wrap=1) (<https://canvas.kth.se/courses/36161/files/6009924?wrap=1>) [↓](https://canvas.kth.se/courses/36161/files/6009924/download?download_frd=1) (https://canvas.kth.se/courses/36161/files/6009924/download?download_frd=1)) and edit code parts demarcated by the `//@@@` comment lines.


- *Allocate host and device memory*

- Initialize host memory and create a reference result in CPU memory
- Copy from host memory to device memory
- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernel
- Copy results from GPU to CPU
- Compare the results with the reference result
- Free host and device memory
- Write the CUDA kernel

Questions to answer in the report

1. Explain how the program is compiled and run.
2. For a vector length of N:
 1. How many floating operations are being performed in your vector add kernel?
 2. How many global memory reads are being performed by your kernel?
3. For a vector length of 1024:
 1. Explain how many CUDA threads and thread blocks you used.
 2. Profile your program with Nvidia Nsight. What **Achieved Occupancy** did you get? You might find <https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#nvprof-metric-comparison>  (<https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#nvprof-metric-comparison>) useful.
4. Now increase the vector length to 131070:
 1. Did your program still work? If not, what changes did you make?
 2. Explain how many CUDA threads and thread blocks you used.
 3. Profile your program with Nvidia Nsight. What **Achieved Occupancy** do you get now?
5. Further increase the vector length (try 6-10 different vector length), plot a stacked bar chart showing the breakdown of time including (1) data copy from host to device (2) the CUDA kernel (3) data copy from device to host. For this, you will need to add simple CPU timers to your code regions.

Exercise 2 - 2D Dense Matrix Multiplication

This exercise will extend to using 2D computational grids for a GPU kernel. You will implement a basic matrix multiplication program that multiplies a 2D matrix A of (numARows, numAColumns) by matrix B of (numBRows, numBColumns) and stores the results into a matrix C (numCRows, numCColumns). You will understand how to index 2D arrays inside a GPU kernel. Please complete the following main steps in your code. You can create your own code, or, use the following code template (**Download Code Template Here** [lab3_ex2_template.cu](https://canvas.kth.se/courses/36161/files/6009944?wrap=1) (<https://canvas.kth.se/courses/36161/files/6009944?wrap=1>)  (https://canvas.kth.se/courses/36161/files/6009944/download?download_frd=1)) and edit code parts demarcated by the `//@@` comment lines.

- Allocate host and device memory

- *Initialize host memory and create a reference result in CPU memory*
- *Copy from host memory to device memory*
- *Initialize thread block and kernel grid dimensions*
- *Invoke CUDA kernel*
- *Copy results from GPU to CPU*
- *Compare the results with the reference result*
- *Free host and device memory*
- *Write the CUDA kernel*

Questions to answer in the report

1. Name three applications domains of matrix multiplication.
2. How many floating operations are being performed in your matrix multiply kernel?
3. How many global memory reads are being performed by your kernel?
4. For a matrix A of (128x128) and B of (128x128):
 1. Explain how many CUDA threads and thread blocks you used.
 2. Profile your program with Nvidia Nsight. What **Achieved Occupancy** did you get?
5. For a matrix A of (511x1023) and B of (1023x4094):
 1. Did your program still work? If not, what changes did you make?
 2. Explain how many CUDA threads and thread blocks you used.
 3. Profile your program with Nvidia Nsight. What **Achieved Occupancy** do you get now?
6. Further increase the size of matrix A and B, plot a stacked bar chart showing the breakdown of time including (1) data copy from host to device (2) the CUDA kernel (3) data copy from device to host. For this, you will need to add simple CPU timers to your code regions. Explain what you observe.
7. Now, change DataType from double to float, re-plot the a stacked bar chart showing the time breakdown. Explain what you observe.

Exercise 3 - Histogram and Atomics

This exercise will practice a parallel computation pattern called histogram using shared memory and atomics in CUDA kernels. You will implement an efficient histogramming algorithm for an input array of integers within a given range. Each integer will map into a single bin, so the values will range from 0 to (NUM_BINS - 1). The histogram bins will use unsigned 32-bit counters that must be saturated at 127 (i.e. the maximum count of a bin is 127 even if its actual count is larger than 127). This can be split into two kernels: one that does a histogram without saturation, and a final kernel that cleans up the bins if they are too large. These two stages can also be combined into a single kernel, but then they will share the same CUDA block and grid configuration for execution.

The input length can be assumed to be less than 2^{32} . NUM_BINS is fixed at 4096 for this lab. Pay attention to synchronization and the definition of shared memory capacity in a CUDA kernel. Please complete the following main steps in your code. You can create your own code, or, use the following code template (**Download Code Template Here** [lab3_ex3_template.cu](#))

(<https://canvas.kth.se/courses/36161/files/6009987?wrap=1>)_ ↓ (https://canvas.kth.se/courses/36161/files/6009987/download?download_frd=1))and edit code parts demarcated by the `//@@` comment lines.

- *Allocate host and device memory*
- *Initialize host memory and create a reference result in CPU memory*
- *Copy from host memory to device memory*
- *Initialize thread block and kernel grid dimensions*
- *Invoke CUDA kernel*
- *Copy results from GPU to CPU*
- *Compare the results with the reference result*
- *Free host and device memory*
- *Write the CUDA kernel*

Questions to answer in the report

1. Describe all optimizations you tried regardless of whether you committed to them or abandoned them and whether they improved or hurt performance.
2. Which optimizations you chose in the end and why?
3. How many global memory reads are being performed by your kernel? Explain
4. How many atomic operations are being performed by your kernel? Explain
5. How much shared memory is used in your code? Explain
6. How would the value distribution of the input array affect the contention among threads? For instance, what contentions would you expect if every element in the array has the same value?
7. Plot a histogram generated by your code and specify your input length, thread block and grid.
8. For a input array of 1024 elements, profile with Nvidia Nsight and report **Shared Memory Configuration Size** and **Achieved Occupancy**. Did Nvsight report any potential performance issues?

Exercise 4 - A Particle Simulation Application

This exercise will look into a simplified version of a particle-in-cell simulation code and focus on its particle mover that calculates and updates the position of particles. You can access the code at <https://github.com/KTH-HPC/sputniPIC-DD2360.git>. The particle mover is contained in `/src/Particles.cu` file, which contains two main computation functions: `mover_PC()` and `interpP2G()` and two main memory management functions: `particle_allocate()` and `particle_deallocate()`.

In this exercise, you will compile and run the current version of the code, which still runs on CPU, and get familiar with the `particle mover_PC()` function in `Particles.cu`. Some background information about the code and how to run a simple simulation can be found in [Introduction of spunitGPU.pdf](https://canvas.kth.se/courses/36161/files/5964916?wrap=1) (<https://canvas.kth.se/courses/36161/files/5964916?wrap=1>)_ ↓ (https://canvas.kth.se/courses/36161/files/5964916/download?download_frd=1) . For this exercise, use the `GEM_2D.inp` input file.

You will then implement a GPU version of `mover_PC()` called `mover_PC_gpu()`. For this exercise,

it is ok to include GPU memory allocation routines inside `mover_PC_gpu()` so that you allocate GPU memory for particles, copy particles to GPU memory, invoke computation, and deallocate GPU memory. Of course, you can create a more optimized implementation that will modify a larger scope of the code. Finish the following steps:

Questions to answer in the report

1. Describe the environment you used, what changes you made to the Makefile, and how you ran the simulation.
2. Describe your design of the GPU implementation of `mover_PC()` briefly.
3. Compare the output of both CPU and GPU implementation to guarantee that your GPU implementations produce correct answers.
4. Compare the execution time of your GPU implementation with its CPU version.