ECE-GY 9143 - High Performance Machine Learning

Homework Assignment 4

Parijat Dube and Kaoutar El Maghraoui

Due Date: April 13, 2024 | Spring 2024 | Max Points: 100

Instructions:
This lab is intended to be performed **individually**, great care will be taken in verifying that students are authors of their own submission and that the exam is different from previous courses.
This lab has three parts. Part-A on vector addition and matrix multiplication in CUDA, Part-B on CUDA Unified Memory, and Part-C on convolution in CUDA. Please perform this lab on a GCP GPU instance (setup instructions are at the end of the document). As we get closer to the deadline, we might experience GPU availability issues. Start early.

The submission template for this assignment is:
Submit a 'UNI.zip' file with the following contents

- Part-A (directory)
  - Makefile
  - vecadd.cu
  - vecaddKernel.h
  - vecaddKernel00.cu
  - vecaddKernel01.cu
  - matmultKernel00.cu
  - matmultKernel01.cu
  - timer.h
  - timer.cu
- Part-B (directory)
  - q1.cpp
  - q2.cu
  - q3.cu
  - q4_with_unified.jpg
  - q4_without_unified.jpg
  - q4.py (the code to plot these results in python or any other language of your choice)
- Part-C (directory)
  - c1.cu
  - c2.cu
  - c3.cu

  - program_output.csv (The program output described in Part-C; can be generated manually)

- report.pdf (The report with outputs from all the parts and answers to questions wherever requested)

Your report must contain screenshots of outputs and execution times from the GCP instance for all programming questions. Paste screenshots for all values of K, threads, and blocks in Part-B Q1, Q2, and Q3. An easy way to time a program is to use the **time** utility in bash. We expect you to maintain academic honesty when submitting your screenshots. Submitting dishonest outputs will invite a strict penalty.

Part-A: CUDA Matrix Operations (40 points)

The purpose of this exercise is for you to learn how to write programs using the CUDA programming interface, how to run such programs using an NVIDIA graphics processor, and how to think about the factors that govern the performance of programs running within the CUDA environment. For this assignment, you will modify two provided CUDA kernels. The first is part of a program that performs vector addition. The second is part of a program to perform matrix multiplication. The provided vector addition program does not coalesce memory accesses. You will modify it to coalesce memory access. You will modify the matrix multiplication program to investigate its performance and how it can be optimized by changing how the task is parallelized. You will create an improved version of the matrix multiplication program and empirically test the time it takes to run. You will analyze the results of your timing tests.

You will turn in all code you have written and used for this assignment, a makefile that will compile your code, and a standard lab report documenting your experiments and results. We need to be able to compile your code by executing "make".

Note: The GCP setup instructions for this part have been provided at the end of this document under Appendix.

Problem-1: Vector add and coalescing memory access

You are provided with `vecadd`, a micro benchmark to determine the effectiveness of coalescing. The provided code is a non-coalescing version, and it is your job to create a coalescing version, and to measure the difference in performance of the two codes.

Here is a set of files provided in `CUDA1.zip` for this problem

- `Makefile`
- `vecadd.cu`
- `vecaddKernel.h`
- `vecaddKernel00.cu`
- `timer.h`
- `timer.cu`

Q1 **5 points**

Compile and run the provided program as `vecadd00` (using the kernel `vecaddKernel00`) and collect data on the time the program takes with the following number of values per thread: 500, 1000, 2000.

Include a short comment in your make file describing what your various programs do, and fully describe and analyse the experiments you performed in your lab report.

Q2 **10 points**

In `vecadd01` you will use a new device kernel, called `vecAddKernel01.cu`, to perform vector addition using coalesced memory reads. Change the `Makefile` to compile a program called `vecadd01` using your kernel rather than the provided kernel. Modify the `Makefile` as appropriate so the clean and tar commands will deal with any files you have added.

Test your new program and compare the time it takes to perform the same work performed by the original. Note and analyse your results, and document your observations in the report.

Problem-2: Shared CUDA Matrix Multiply

For the next part of this assignment, you will use `matmultKernel00` as a basis for another kernel with which you will investigate the performance of matrix multiplication using a GPU and the CUDA programming environment. Your new kernel should be called `matmultKernel01.cu`. Your code files must include internal documentation on the nature of each program, and your `README` file must include notes on the nature of the experiment. You can have intermediate stages of this new kernel, eg without and with unrolling, but we will grade you on your final improved version of the code and its documentation.

Here is a set of provided files in `CUDA1.zip` for this problem:

- `Makefile`
- `matmult.cu`
- `matmultKernel.h`
- `matmultKernel00.cu`
- `timer.h`
- `timer.cu`

In both the questions of this problem (Q3 and Q4 below) you should investigate how each of the following factors influences performance of matrix multiplication in `CUDA`:

1. The size of matrices to be multiplied
2. The size of the block computed by each thread block
3. Any other performance enhancing modifications you find

Q3 **8 points**

First you should time the initial code (with the provided kernel `matmultKernel00`) using square matrices of each of the following sizes: 256, 512, 1024.
When run with a single parameter, the provided code multiplies that parameter by `FOOTPRINT_SIZE` (set to 16 in `matmult00`) and creates square matrices of the resulting size. This was done to avoid nasty padding issues: you always have data blocks perfectly fitting the grid.

Q4 **12 points**

In your new kernel each thread computes four values in the resulting C block rather than one. So now the `FOOTPRINT_SIZE` becomes 32. (Notice that this is taken care of by the `Makefile`.) You will time the execution of this new program with matrices of the sizes listed above to document how your changes affect the performance of the program. Provide the speedup that your new kernel achieves over the provided kernel.

To get good performance, you will need to be sure that the new program coalesces its reads and writes from global memory. You will also need to unroll any loops that you might be inclined to insert into your code.

Q5 **5 points**

Can you formulate any rules of thumb for obtaining good performance when using CUDA? Answer this question in your conclusions.

Part-B: CUDA Unified Memory (20 points)

In this problem we will compare vector operations executed on host vs on GPU to quantify the speed-up.

Q1 4 points

Write a C++ program that adds the elements of two arrays with a K million elements each. Here K is a command-line parameter of the program. Profile and get the time to execute this program for K=1,5,10,50,100. Use free() to free the memory at the end of the program.

Q2 6 points

Next using CUDA execute the add operation as a kernel on GPU. Use cudaMalloc() to allocate memory on GPU for storing the arrays and cudaMemcpy() to copy data to and from the GPUs. Note that host and GPU memory is not shared in this case. You will be running three scenarios for this part:

1. Using one block with 1 thread
2. Using one block with 256 threads

3. Using multiple blocks with 256 threads per block with the total number of threads across all the blocks equal to the size of arrays.

Again, profile and get the time to execute this program for $K = 1, 5, 10, 50, 100$. So for each of the three scenarios above you will get profile time for five different values of K Use cudaFree() and free() to free the memory on the device and host at the end of the program.

Q3 6 points

This time you will repeat Step 2 using CUDA Unified Memory and providing a single memory space accessible by all GPUs and CPUs in your system. Instead of cudaMalloc() you will use cudaMallocManaged() to allocate data in unified memory, which returns a pointer that you can access from host (CPU) code or device (GPU) code. To free the data, just pass the pointer to cudaFree(). Again, profile and get the time to execute this program for $K = 1, 5, 10, 50, 100$ for each of the three scenarios.

Q4 4 points

Plot two charts one for Step 2 (without Unified Memory) and one for Step 3 (with Unified Memory). The x-axis is the value of K (in million) and y-axis is the time to execute the program (one chart for each of the three scenarios). In both the charts also plot the time to execute on CPU only. Since the execution time scale on CPU and GPU may be orders of magnitude different, you may want to use log-log scale for y-axis when plotting.

For this problem you may want to refer to slide deck **HPML-10-CUDA-basics-Fall2022** which has a similar example and the following blogpost:
An Even Easier Introduction to CUDA, available at https://developer.nvidia.com/blog/even-easier-introduction-cuda/

Part-C: Convolution in CUDA (40 points)

C1: Convolution in CUDA 15 points

In this part of the lab we implement a convolution of an image using a set of filters. Consider the following:

- An input tensor I with dimensions: C, H, W. Each element of I is generated as follows:

$$I[c, x, y] = c \cdot (x + y)$$

- A set of convolution filters with dimensions: K, C, FH, FW. Each element of the filter F is generated as follows:

$$F[k, c, i, j] = (c + k) \cdot (i + j)$$

- Dimensions are: $H = 1024$, $W = 1024$, $C = 3$, $FW = 3$, $FH = 3$, $K = 64$.

- All tensors have double elements (double precision)
- A tensor I_0 of sizes C, W+2P, H+2P where P=1. I_0 is obtained from the tensor I adding the padding rows and columns with the elements set to zero.
- The output tensor O with dimensions: K, W, H. Each pixel of the output tensor $O[k, x, y]$ is obtained as:

$$O[k, x, y] = \sum_{c=0}^{C-1} \sum_{j=0}^{FH-1} \sum_{i=0}^{FW-1} F[k, c, FW - 1 - i, FH - 1 - j] \cdot I_0[c, x + i, y + j]$$

Note that we need to use the transpose of the filter in order to compute a convolution rather than a cross-correlation. Implement a simple convolution algorithm in CUDA without tiling and without shared memory.

- Print the checksum as the total sum of the elements of O along all its dimensions.
- Report the time to execute the CUDA kernel with the convolution, without including the time to generate the data and to copy it into CUDA memory (only the kernel execution time – you need to use a synchronization primitive).

C2: Tiled Convolution with CUDA 15 points

Implement the convolution at exercise C1 using shared memory and tiling.

- Print the checksum as the sum of the elements of O along all its dimensions. The checksum is expected to be the same as C1.
- Report the time to execute the CUDA kernel with the convolution, without including the time to generate the data and to copy it into CUDA memory (only the kernel execution time – you need to use a thread synchronization primitive).

C3: Convolution with cuDNN 10 points

Implement the convolution at exercise C1 using cuDNN with the *CUDNN_CONVOLUTION_FWD_PREFER_FASTEST* algorithm selection.

- Print the checksum as the sum of the elements of O along all its dimensions. The checksum is expected to be the same as C1.
- Report the time to execute the CUDA kernel with the convolution, without including the time to generate the data and to copy it into CUDA memory (only the kernel execution time).

Program Output

```
C1_checksum,C1_execution_time
C2_checksum,C2_execution_time
C3_checksum,C3_execution_time
```

Where each string contains the checksum and the execution time of the exercises C1, C2 and C3 respectively. Execution times should be printed in milliseconds with 3 decimals. Failing to respect this format is -3 points.

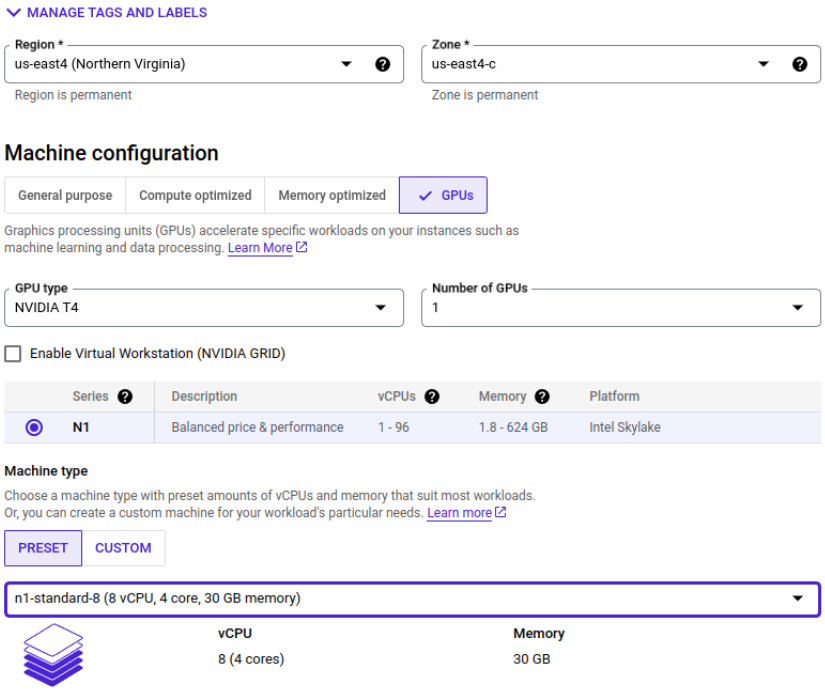

Figure 1: Caption

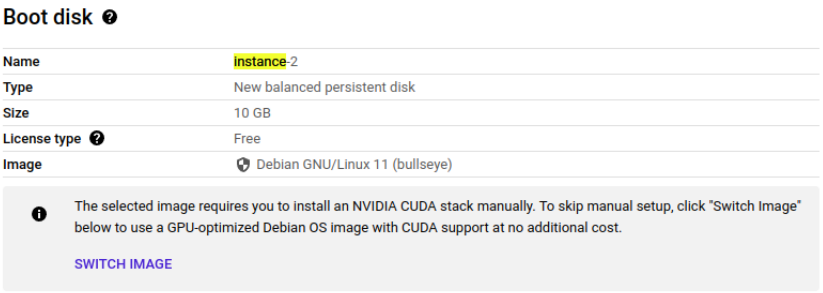

Figure 2: Caption

Appendix

Setup for Part-A on Google Cloud Platform

Setting up your GPU instance on GCP

1. Make sure your GPU resource quota supports at least one GPU. To verify and increase quota, check: https://cloud.google.com/compute/resource-usage.

2. Create a new instance on GCP with the configuration shown in figure 1. Scroll down to the "Boot disk" option and choose "SWITCH IMAGE". Choose the following image: " Deep Learning VM with CUDA 11.3 M112" (See figure 2).

3. SSH into your instance using the web terminal and when it prompts you to install the NVIDIA CUDA driver, press "y" and the return key. This will automatically install the correct driver on your system. The installer should end with a "Nvidia driver installed" message on your system.

Verifying that Part-A works on your VM instance

1. Upload the 'CUDA1.zip' file to your VM instance using the UPLOAD FILE option at the top in your web terminal.

2. Unzip CUDA1.zip using 'unzip CUDA1.zip' and remove any pre-existing output files using 'rm *.o'.

3. Use 'nano Makefile' or any other editor of your choice. Update the CUDA SDK Install Path (SDK_INSTALL_PATH) to point to your current installation - '/usr/local/cuda'.

4. Now, run 'make' command to build the files.

5. Your installation works correctly if the './vecadd00 500' command gives you 'Test PASSED'.

What to do if my installation fails unexpectedly or if I am stuck?

Let one of the TAs know immediately through ED and visit OHs regularly. This homework is particularly long and every single day counts. **Please do not delay letting us know if you are stuck in the setup phase**.