CSCI-GA.3033-023 - Lab1

This lab is intended to be performed **individually**, great care will be taken in verifying that students are authors of their own submission.

Theoretical questions are identified by **Q<number>** while coding exercises are identified by **C<number>**.

Inferencing in C and Python

In this part of the lab we explore the performance of an algorithm that does inferencing of a single sample using the first two layers of a fully connected neural network.

For each layer $l \in [0,1]$, denote with W_l the matrix of the weights, with x_l the vector of the input and with z_l the vector of the output (after the activation function has been computed). Assume the bias to be a vector of zeroes that can be ignored.

The dimension of the network are as follows:

1) The first layer takes as input an image of size 224 x 224 pixel (`50176 features`) and is composed of 4,000 neurons with a ReLU activation function.
2) The second layer takes as input the 4,000 outputs of the first layer and is composed of 1000 neurons with a ReLU activation function.

Q1:

- What are the dimensions of W_l, x_l, and z_l for each layer l?

Q2:

- Assuming double precision operations, what are W_l, x_l, and z_l sizes in memory for each layer l?

[Hint: refer to a C implementation of the code.]

C1: Memory bandwidth measurement:

Write a single-threaded (serial) microbenchmark to estimate the peak memory bandwidth from actual experimental measurements. The size of this array must be large enough to obtain peak bandwidth (not related to the size of inference network above).

- Report the best memory bandwidth reached across 16 measurements

[Hints: sum all elements of an array; be careful with the memory size, we want to avoid the speed-up of data residing in the cache. Use the makefile specified in the appendix. Make sure to avoid compiler optimizations that could invalidate the measurement, you can do that printing the value of the sum.]

Q3: Compare to Intel specifications

- Report what percentage of the peak memory bandwidth has been achieved by the microbenchmark in **C1** using the processor specifications.
- The memory bandwidth achieved in **C1** is lower that the processor peak, what is the main reason and how could be improved?

[Hints: use the command *cat /proc/cpuinfo* to obtain the model of the processor. Make sure to read the cpu model from the reserved cpu node and not from other nodes in the cluster. Find the specification on the Intel website https://ark.intel.com.]

C2: Python code:

Write a reference implementation of the inferencing for the network described above in Python. Use the following initialization function for vectors or matrices x, z or W:

x[i, j] = 0.5 + ((i+j)%50-30)/50.0

Then:

- Report the execution time excluding the initialization of the vectors and matrices.
- Report the checksum value *S* as the sum of the elements in the output vector of the last layer (z_1)

C3: Numpy code:

Use Numpy to speed up the dense computation of the output of each layer.

- Report the execution time excluding the initialization of the vectors and matrices.
- Report the speedup with respect to the execution time of **C2.**
- Verify that the checksum value *S* is the same as in **C2** (excluding the floating point rounding error)

[Hint: use numpy.clip() and numpy.dot().]

C4: C implementation

Write the equivalent of the reference implementation in C and compare the performance and the results with the Python versions. Initialize all the elements of the matrices W_l and the input vector x_0 with the same function described in C2.

- Report the speedup with respect to the execution time of **C2**
- Verify that the checksum value *S* is the same as in **C2** (excluding the floating point rounding error)

C5: Use the Intel MKL library to speed up the linear part of the computation.

- Report the execution time excluding the initialization of the vectors and matrices.
- Report the speedup with respect the execution time of **C2**
- Verify that the checksum value *S* is the same as in **C2** (excluding the floating point rounding error)

[Hint: https://software.intel.com/en-us/mkl-developer-reference-c-cblas-gemv]

Training in PyTorch

In this part of the lab we create a Neural Network in PyTorch to classify a dataset of images. The dataset is stored in the folder */scratch/am9031/CSCI-GA.3033-023/kaggleamazon/* and is composed by a training and a validation set of respectively 20000 and 1479 images, classified in 17 labels. In the files *train.csv* and *test.csv* you will find the filename of each image and the related label index. All the images are physically stored in the sub-folder *train-jpg/*

C6:
Create a PyTorch program with a DataLoader that loads the images and the related labels from the filesystem. During the creation of the dataset, make a two steps data-augmentation (pre-processing), using the *torchvision* package, with the following sequence of transformations:

1. Resize the images to squares of size 32x32 (*transforms.Resize()*)
2. Transform the images to tensor (*transform.ToTensor())*

The DataLoader for the training set as well for the validation set uses a minibatch size of 100 and one single worker.

Create a Neural Network composed by 3 fully connected layers.

1. The first layer has an input size equal to 32x32X3=3072 and an output size of 1024
2. The output of second layer is 256
3. The output of the third layer must be the number of the labels, ie 17

Define the forward function of the NN with the three created layers, using a ReLU activation function for the first two layers and a Softmax activation for the last one. Be careful in the input size, the minibatches of images will create a tensor of shape (mini_batch_size, 3, 32, 32), that is ok if we were using convolutional layers, but not for only fully connected, so you need to reshape the input Variable to be (mini_batch_size, 3072), using the *view()* function.
Use as optimizer an SGD algorithm with learning rate 0.01 and momentum 0.9.
Use *nn.CrossEntropyLoss()* as loss computation criterion.
Create a main function that creates the DataLoaders for the training and validation sets. Then do a cycle of 5 epochs with a complete training phase on all the minibatches of the training set.

C7: Time measurement of code in C6
Using the code from exercise C6 report the real time using *time.monotonic()* for the following sections of the code:

- Aggregate DataLoader I/O time
- Aggregate DataLoader Data-augmentation (Preprocessing) time
- Aggregate time spent waiting to load the batch (data and target) from the DataLoader during the training
- Time to perform 5 epochs

C8: I/O optimization starting from code in C7

- Report the total time spent waiting for the Dataloader varying the number of workers starting from zero and increment the number of workers until the time doesn't decrease anymore.
- Report how many workers are needed for best performance

C9: Training optimization starting from the code in C8

- Report the average epoch time over 10 epochs using the CPU vs using the GPU
- With the GPU-enabled code, report the average epoch time over 10 epochs using these Optimizer algorithms:
    - SGD
    - SGD with momentum
    - Adam

Grading

The grade will be a total of 50 points distributed as follows:

EXERCISE	DESCRIPTION	POINTS
Q1	theoretical question	2
Q2	size question	1
C1	memory bw (C)	5
Q3	cpu peak bw	2
C2	inference (Python)	8
C3	inference (Python + Numpy)	2
C4	inference (C)	8
C5	inference (C+ MKL)	2
C6	training (PyTorch)	8
C7	code timing (PyTorch)	5
C8	I/O optimization (PyTorch)	2
C9	GPU and Opmitizer optimization (PyTorch)	5

Other grading rules:

- Late submission is -3 points for every day, maximum 3 days.
- Non-compiling code: 0 points

Appendix – Submission instructions

Submission through NYUClasses.

Please submit a targz archive with file name *<your-netID>.tgz* with a folder named as your netID (example: am9031/) containing the following files:

- A pdf file with the answers to all exercises (measurements and textual answers)
- A file named lab1.c containing exercises C1, C4 and C5 (executed in sequence)
- A file named lab1.py containing exercises C2 and C3 (executed in sequence)
- A file named lab1.pytorch containing exercise C6, C7, C8, C9 (leave the code for C6, C7, C8 uncommented, while commenting the code for C9)

Failing to follow the right directory/file name specification is -1 point. Failing to have programs executed in sequence is -1 point.

Appendix – How to Run Experiments

To test your code while developing you can use the Prince HPC cluster.

For this lab you will be using 4 types of nodes:

- CPU-only nodes:
  - Development (various nodes)
  - Reserved for the course (2 nodes)
- GPU nodes
  - Development (various nodes)
  - Reserved for the course (1 nodes)

You must follow these rules to run jobs:

1. All jobs that do not require a GPU need to be executed on the **CPU-only nodes**. There are only a few **GPU nodes**
2. You can run on the **reserved nodes** to get performance numbers only when your code is **functional and it returns correct results**, for all other jobs you must run on the **development nodes**

The folder */scratch/am9031/CSCI-GA.3033-023* contains:

- Makefile: the makefile to use for the C exercises
- Job launch scripts:
  - launch_cpu_develop.s: job script to use when launching a standard development cpu-only job

    - launch_cpu_reserv.s: job script to use when launching a “performance measurement” job, run on these nodes only when you are confident that your code functions properly and you are ready to take the measures
    - launch_gpu_develop.s: job script to use when launching a standard development job that requires a gpu
    - launch_gpu_reserv.s: job script to use when launching a “performance measurement” job that requires a gpu, run on this node only when you are confident that your code functions properly and you are ready to take the measures
- Dataset for the PyTorch exercise in */scratch/am9031/CSCI-GA.3033-023/kaggleamazon/*