CHALMERS
UNIVERSITY OF TECHNOLOGY

DAT400 Lab 1: Introduction to tools and application.

Sonia Rani Gupta, Hari Abram, Miquel Pericàs

September 1, 2025

The Generalized Matrix-Matrix Multiplication (GEMM) is one of the traditional Basic Linear Algebra Subprograms (BLAS), but it still lives at the heart of modern Artificial Intelligence (AI) and Deep Learning applications, such as self-driving cars, pattern matching, etc. In image recognition, for example, Convolutional Neural Networks [Fig. 1] (CNN) transform the images into batches of 2D matrices. GEMM is then used for carrying out the kernel updates of weights and biases. GEMM is also heavily used in numerical science as it aids in solving linear systems ($AX = b$) arising from Partial Differential Equations (PDEs). Last but not least, it is essentially the benchmark for measuring the sustainable floating-point performance of the top 500 supercomputers in the world.

In this experiment, you are given a code that trains a 3-layer network for recognizing handwritten Arabic numerals by training a multiple-layer network.
Read more on the convolution and training phase: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf
Read more on matrix multiplication here. https://en.wikipedia.org/wiki/Matrix_multiplication

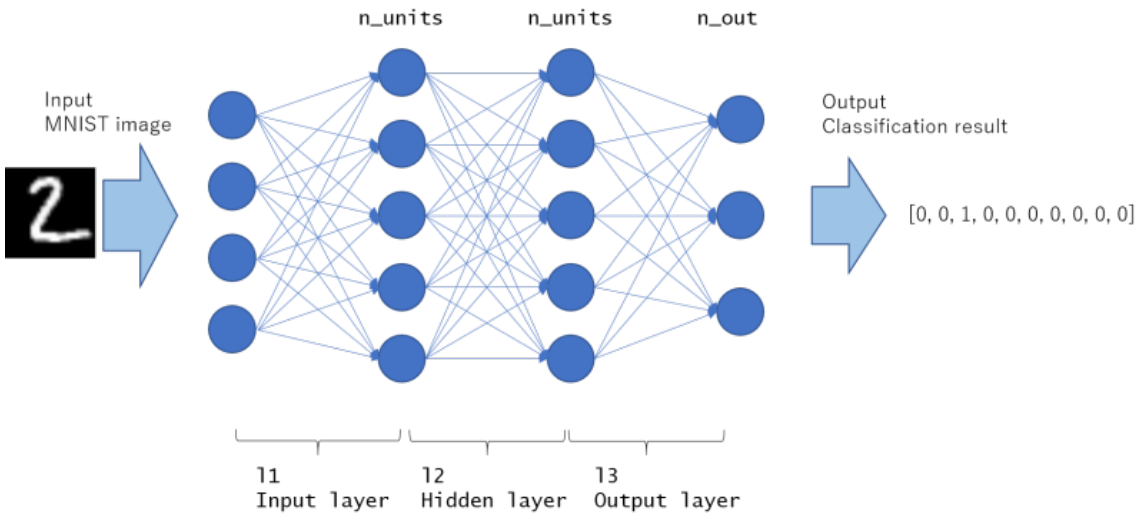

Figure 1: The neural network for the MNIST dataset.

The task of the DAT400 labs is to train the network faster, by optimizing the most computationally expensive part. Lab 1 provides an introduction to the `make`, `gdb`, and `perf` tools.

General Instructions:

a) Open a terminal session by hitting the menu button and searching for `terminal`.
b) Download the lab exercise from the Canvas portal.
c) Un-compress the folder using *tar -xvf TAR_FILE_NAME*.
d) Compile the code: *make*
e) Run code: *make run_serial*

The execution may take a while. When finished, it should print the predictions and ground truth and then get a segmentation fault and crash.

```
[pirah@tegra-ubuntu:~/DAT400/cpu$ make run_serial
./nnetwork
Loading data ...
Training the model ...
Predictions:
0.124327 0.0738499 0.208982 0.0928889 0.0538223 0.115591 0.105101 0.0957345 0.051625 0.0780778
0.130716 0.0572842 0.296196 0.0826492 0.0349954 0.11622 0.0998998 0.0864578 0.0326912 0.0628909
0.127809 0.0675256 0.239881 0.0894578 0.0462237 0.11629 0.103776 0.0927333 0.0437267 0.0725781
0.128875 0.0367222 0.43358 0.0638592 0.0177907 0.106697 0.0854819 0.0685234 0.0159648 0.0425064
0.131108 0.0439915 0.380328 0.0712509 0.023227 0.112019 0.0916783 0.0754228 0.0211341 0.0498401
0.131499 0.048938 0.346889 0.075956 0.0274653 0.113532 0.095123 0.080424 0.0252585 0.0549148
0.131728 0.0487348 0.348157 0.07559 0.0271559 0.114175 0.0949726 0.079866 0.0249916 0.0546286
0.128746 0.0653159 0.25064 0.0881251 0.0439561 0.116622 0.103156 0.0915485 0.0414592 0.0704309
0.128338 0.0356545 0.441746 0.0627059 0.0170647 0.105817 0.0846082 0.0673843 0.0152792 0.0414028
0.131314 0.0550064 0.307785 0.0810613 0.0331526 0.116055 0.0989145 0.0850083 0.030777 0.0609262

Ground truth:
0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0 0 0
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0

Iteration #: 99
Iteration Time: 0.547157s
Loss: 0.963032
*****************************************
```

Figure 2: Example output without a segmentation fault.

Some basic commands of Linux terminal:

- **cd**: Change directory, e.g., *cd cpu*
- **ls**: List the items inside a directory, e.g., *ls -l* gives more details of items inside the directory.
- **pwd**: Print the absolute path of current directory.
- **cp**: Copy files, e.g., *cp [source] [destination]*
- **rm**: Remove file or folder, e.g., *rm foo.h* or *rm -r foo/* - if foo is a folder.

1 Program Compilation using Makefile

The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. You need a file called a *makefile* to tell make what to do. A simple makefile consists of "rules" with the following shape:

target ... : prerequisites ...
recipe
...

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as 'clean'. A *prerequisite* is a file that is used as input to create the target. A target often depends on several files. A *recipe* is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. Note: put a **tab** character at the beginning of every recipe line. For example, in the lab1 code:

nnetwork_mpi.o:
g++ -c -o $@ nnetwork.cxx ${CXXFLAGS} -DUSE_MPI
nnetwork.o:
g++ -c -o $@ nnetwork.cxx ${CXXFLAGS}
nnetwork_mpi: $(OBJ) nnetwork_mpi.o
g++ -o $@ $^ $(LIBS)
nnetwork: $(OBJ) nnetwork.o
g++ -o $@ $^ $(LIBS)

In the example, the targets are the executable files *nnetwork* and *nnetwork_mpi*, and the object files *nnetwork.o* and *nnetwork_mpi.o*. In the makefile, we name the target *nnetwork* to *run_serial*. To create an executable file called *nnetwork* using the makefile given in the lab, type:

make

To run the executable file created using makefile, type:

make run_serial

To use this makefile to delete the executable file and all the object files from the directory, type:

make clean

The variable *$@* represents the name of the file being created (i.e. the target) and *$^* represents all the prerequisites required to create the output file.

In terms of compiler flags, $(CC) is used for compiling C programs, $(CXX) is used for compiling C++ programs. Similarly $(CFLAGS) is used for C programs, $(CXXFLAGS) is used for compiling C++. Another important flag in the makefile is called the linker flag. For example, you could add -fopenmp at the end of CXXFLAG or create a new rule, CXXFLAG += -fopenmp to enable OpenMP in the program.

1.1 Task 1

- Add a compiler variable, which has been commented out as *#CXX=* in Makefile, replace g++ with the new variable in all rules.
- Currently object files *deep_core.o* and *vector_ops.o* are generated separately. You task is to create a new single rule to generate both by using % (means for all in make) sign. In other words, if you have any C++ source file *example.cpp* the rule should be able to compile it into *example.o*.

2 GDB Debugging

GDB stands for GNU Project Debugger. It operates on executable files which are binary files produced by the compilation process.

- To enable debugging through gdb, add -g flag while compilation.
- Go to the terminal and type gdb ./binary_file_name.
- Now you are inside the gdb environment type **run or r** and press enter to execute the program.

Some important commands to work around:

- **bt**: Print a backtrace of the entire stack.
- **break or b**: Sets break-point on a particular line.
- **c**: Continue the execution until next break-point.
- **next or n**: Executes next line of code, but don't dive into functions.
- **print or p**: Used to display the stored value.
- **quit or q**: Exit.

2.1 Task 2

- Change Makefile to enable debug mode.
- As the code has a segmentation fault, first, locate the line and fix the error in the code using the gdb commands described above.
- Set a break-point on relu function, and print the value of the following variables:
  - What is the value of the size variable?
  - What is z here?
  - What is the value of z[i] when i=3?
- Summarize your answer in the report.

If you are receiving strange errors from gdb not finding the file when trying to run, see if the SHELL variable is empty in the terminal environment by running *echo SHELL*. If it is, try running *export SHELL="/bin/sh"* and try again.

3 Perf: A performance monitoring and analysing tool

The *perf* command in Linux gives access to various tools integrated into the Linux kernel. These tools can help you collect and analyze the performance data about the program or system. We will use perf to analyze the hot spots of our program and to monitor the cache usage. Some important perf tools are as follows:

- **perf record:** The perf record command is used to sample the data of events to a file. Use the perf report command to view the monitoring data.
- **perf list:** The list command shows the list of events that can be traced by the perf command
- **perf stat:** The perf stat command can be used to count the events related to a particular process or command. The stat command gives you the option to select only specific events for reporting. To select the events, run the stat command with the -e option followed by a comma-separated list of events to report. For example:

  perf stat -e cycles,page-faults ./binary_file_name

  This command provides statistics about the events named cycles and page-faults for all the cores on the system.

A popular way of visualizing the perf profiling data and the most frequent code-paths in the program is by using ***flame graphs***. It allows us to see which function calls take the biggest portion of execution time. Figure 3 shows the example of a flame graph for a benchmark. From the mentioned flame graph, we can see that the path that takes the most amount of execution time is x264 → threadpool_thread_internal → slices_write → slice_write → x264_8_macroblock_analyse. The original output is interactive and allows us to zoom into a particular code path. The flame graph was generated with open-source scripts developed by Brendan Gregg. As the first step, clone the repository to your lab machine as follows:

git clone https://github.com/brendangregg/FlameGraph.git

See more detailed instructions of how to generate the flame graph at
https://github.com/brendangregg/FlameGraph.

3.1 Task 3

- Create a rule for enabling perf record in Makefile, (e.g. make run_perf). Write down the rule for make run_perf in the report.
- Use perf record to discover where most of the execution time was spent. Summarize your observations in the report.
- Use perf list to find the event for LLC (Last-Level Cache) misses, and run the program with it enabled. Report the LLC misses. A miss in the LLC means that the machine must read the data from the main memory, which is a very slow operation. This is why we are interested in the LLC misses.
- Generate your flame graph of nnetwork and summarize your observation in the report. Remember that to generate a flame graph with *perf.data*, you need to add flags *-g –call-graph fp* while recording for perf data. Also, it would help if you re-compile your *nnetwork* binary with *-fno-omit-frame-pointer* flag to resolve all the user and kernel level calls.

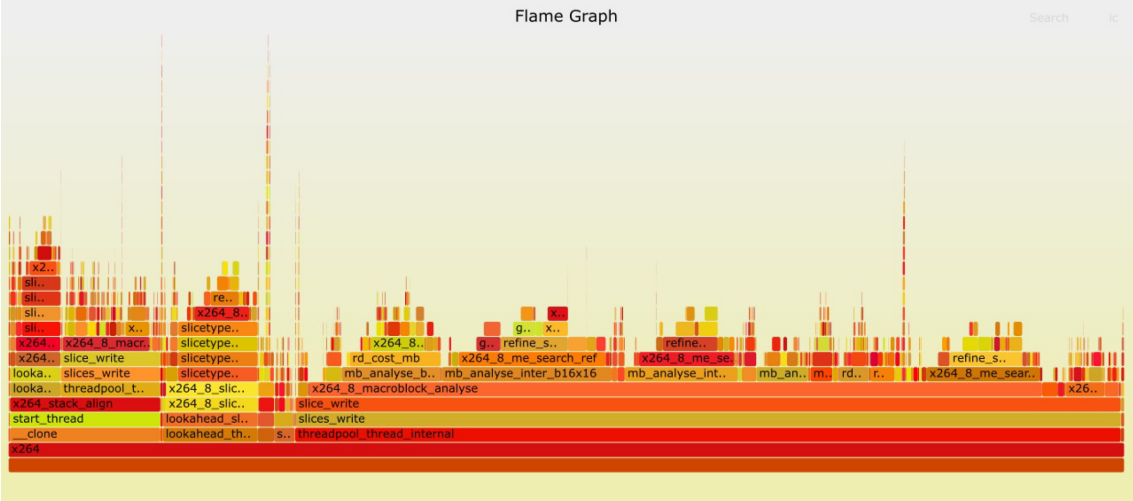

Figure 3: A flame graph example.

4 Compiler Reports

Compilers play a very important role in speeding up applications. Usually, developers leave this job to compilers, interfering only when they see an opportunity to improve something compilers cannot accomplish. compilers provide optimization reports which are used for performance analysis. Sometimes you want to know if some function was inlined or loop was vectorized, unrolled, etc. If it was unrolled, what is the unroll factor?

We will now investigate what optimizations can be done by g++ compiler and what optimizations are implemented by the compiler on our test application.

4.1 Task 4

- In the terminal type *man g++*, scroll to the section "Optimization Options". Try at least 3 optimization flags and explain what is their impact. Edit Makefile to add optimization options as CXXFLAGS.
- Generate compiler report for specific optimization. Add *-fopt-info-loop-optimized=loop.opt* in CXXFLAGS. You must add an optimization flag to generate the report. Which compiler flags did you use to generate the report?
- Analyze the generated report and describe which part of the code is optimized and what optimizations are done during compilation.

Lab 1 Report Submission:

- Submit your report on Canvas as a group. Write down the group number and CIDs of each partner.

- You will get a PASS only if all the tasks are properly addressed in the report.
- The reports should be submitted by 15th September.