TDT4200-PS3 – Optimization and BLAS

Jacob Odgård Tørring and Zawadi Berg Svela

Deadline: Wed October 13, 2020 at 22:00 on BlackBoard
Evaluation: Pass/Fail

- All problem sets are to be handed in INDIVIDUALLY. You may discuss ideas with max. 1-2 collaborators and note them in the comments on the top of your code file, but code sharing is strongly discouraged. Preferably seek help from TAs rather than get unclear/confusing advice from co-students!
- Code should not use other external dependencies aside from the ones we specified.

This assignment is based on the implementation of a general ANN (Artificial Neural Network), called genann.c at https://github.com/codeplea/genann

We have helped you along by in-lining some comments in genann.c, regarding what we would like you to optimize. You should do this optimization primarily by replacing some code with calls to a BLAS library (Basic Linear Algebra Subprogram), you will do some profiling of the original code to verify where the implementations of the program spends most of its time, and optimize also other parts of the code.

You should test your optimizations of genann.c by testing it with example 4, provided. To pass this task, you need to replace the double for-loop in "TODO 1" and "TODO 2" genann.c with BLAS calls as commented in the code.

1 Introduction

About BLAS

BLAS is a library of functions for linear algebra operations. Or rather, BLAS is a specification, and there are several implementations, including OpenBLAS and a version that is baked into Intel's Math Kernel Library (a.k.a. MKL). OpenBLAS is installed on all the lab computers and is the one we will be using. We will also be using the part of BLAS called CBLAS, which contains functions that allow for matrices to be row-major, rather than column-major as the rest of BLAS assumes.

BLAS functions look quite cryptic at first glance, so be sure to check out the documentation when using them. Documentation can be found on Intel's OneMKL website, `https://software.intel.com/content/www/us/en/`

`develop/documentation/onemkl-developer-reference-c/top/blas-and-sparse-blas-routines/blas-routines/blas-level-2-routines/cblas-gemv.html` or the netlib website, `http://www.netlib.org/blas/`. For part 1 and part 2 you will be needing the dgemv_cblas() (Double GEneral Matrix Vector multiplication) call to compute a matrix vector product.

About Profiling

In order to evaluate which part of the code we should optimize, we need to profile it. Optimizing code blindly is rarely a good idea, best case it won't hurt, worst case you will waste time making your program unreadable, and no faster than it was originally.

The main profiling tool we will be using in this assignment is from a collection of tools called Valgrind, specifically Callgrind. This is a tool that runs your code and analyses how much time is spent in different function calls. While you could have done this manually by inserting timing functionality like we did in PS2, part 1, automating this process saves us a lot of work.

Running a program through Valgrind is done similarly to how we ran programs in an MPI environment:

```
valgrind --tool=callgrind ./program
```

This generates a callgrind report named something like callgrind.out.123456, the number at the tail being the process ID. If you are generating multiple reports it might be beneficial to rename them. You can also use `--callgrind-out-file=<file>` to name your file as you generate the report, e.g.

```
valgrind --tool=callgrind --callgrind-out-file=example4-prof-1 ./example4
```

Valgrind also has other tools you may want to look into. Memcheck helps you track down memory leaks and other memory errors (like segmentation faults) and Cachegrind helps you optimize cache performance. The execution of the program slows down drastically when profiling. This means that it will take a while to profile example 4 (up to several minutes).

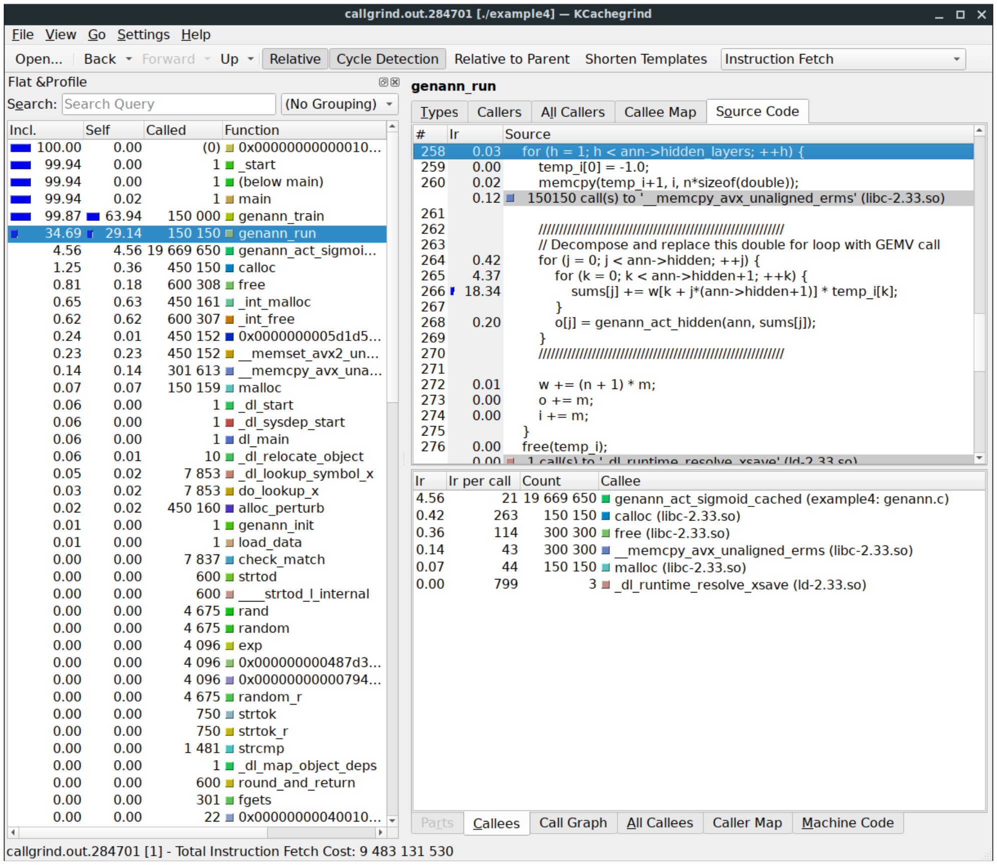
The **example4** program should be run using the **time** command: **time ./example4** to measure the total runtime for the measurement textfile deliverable.

Visualization: Kcachegrind/Qcachegrind

In order to view the report you should use kcachegrind/qcachegrind: **kcachegrind callgrind.out.123456**. The Source Code view will show how much time was spent for each line. The time distribution between each function can be seen on the left. See Figure 1 for an example.

It is run using the following commands:

- **Windows/WSL/Linux**: `kcachegrind callgrind.out.123456`


callgrind.out.284701 [./example4] — KCachegrind
File View Go Settings Help
Open... Back Forward Up Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch
Flat &Profile
Search: Search Query (No Grouping)
genann_run
Types Callers All Callers Callee Map Source Code
Parts Callees Call Graph All Callees Caller Map Machine Code

Incl.	Self	Called	Function
100.00	0.00	(0)	0x00000000000010...
99.94	0.00	1	_start
99.94	0.00	1	(below main)
99.94	0.02	1	main
99.87	63.94	150 000	genann_train
34.69	29.14	150 150	genann_run
4.56	4.56	19 669 650	genann_act_sigmoi...
1.25	0.36	450 150	calloc
0.81	0.18	600 308	free
0.65	0.63	450 161	_int_malloc
0.62	0.62	600 307	_int_free
0.24	0.01	450 152	0x0000000005d1d5...
0.23	0.23	450 152	__memset_avx2_un...
0.14	0.14	301 613	__memcpy_avx_una...
0.07	0.07	150 159	malloc
0.06	0.00	1	_dl_start
0.06	0.00	1	_dl_sysdep_start
0.06	0.00	1	dl_main
0.06	0.01	10	_dl_relocate_object
0.05	0.02	7 853	_dl_lookup_symbol_x
0.03	0.02	7 853	do_lookup_x
0.02	0.02	450 160	alloc_perturb
0.01	0.00	1	genann_init
0.01	0.00	1	load_data
0.00	0.00	7 837	check_match
0.00	0.00	600	strtod
0.00	0.00	600	____strtod_l_internal
0.00	0.00	4 675	rand
0.00	0.00	4 675	random
0.00	0.00	4 096	exp
0.00	0.00	4 096	0x000000000487d3...
0.00	0.00	4 096	0x00000000000794...
0.00	0.00	4 675	random_r
0.00	0.00	750	strtok
0.00	0.00	750	strtok_r
0.00	0.00	1 481	strcmp
0.00	0.00	1	_dl_map_object_deps
0.00	0.00	600	round_and_return
0.00	0.00	301	fgets
0.00	0.00	22	0x00000000040010...

```
#    Ir     Source
258  0.03    for (h = 1; h < ann->hidden_layers; ++h) {
259  0.00        temp_i[0] = -1.0;
260  0.02        memcpy(temp_i+1, i, n*sizeof(double));
     0.12    150150 call(s) to '__memcpy_avx_unaligned_erms' (libc-2.33.so)
261
262              ////////////////////////////////////////////////////////////
263              // Decompose and replace this double for loop with GEMV call
264  0.42        for (j = 0; j < ann->hidden; ++j) {
265  4.37            for (k = 0; k < ann->hidden+1; ++k) {
266 18.34                sums[j] += w[k + j*(ann->hidden+1)] * temp_i[k];
267                  }
268  0.20            o[j] = genann_act_hidden(ann, sums[j]);
269              }
270              ////////////////////////////////////////////////////////////
271
272  0.01        w += (n + 1) * m;
273  0.00        o += m;
274  0.00        i += m;
275          }
276  0.00    free(temp_i);
     0.00    1 call(s) to '_dl_runtime_resolve_xsave' (ld-2.33.so)
```

Ir	Ir per call	Count	Callee
4.56	21	19 669 650	genann_act_sigmoid_cached (example4: genann.c)
0.42	263	150 150	calloc (libc-2.33.so)
0.36	114	300 300	free (libc-2.33.so)
0.14	43	300 300	__memcpy_avx_unaligned_erms (libc-2.33.so)
0.07	44	150 150	malloc (libc-2.33.so)
0.00	799	3	_dl_runtime_resolve_xsave (ld-2.33.so)

callgrind.out.284701 [1] - Total Instruction Fetch Cost: 9 483 131 530

Figure 1: KCachegrind example on the baseline implementation

- **Mac**: `qcachegrind callgrind.out.123456`

Replace "callgrind.output.123456" with another filename as necessary.

Installing Valgrind and GUI

Linux: Valgrind and kcachegrind can be installed using apt on Ubuntu: `sudo apt install valgrind kcachegrind`.
Windows/WSL: Valgrind via works correctly on WSL 2, but no on WSL 1. So make sure that you have upgraded to WSL 2. To install valgrind simply run `sudo apt install valgrind`. QCachegrind[1] can be used to view the result of the profiling on Windows.
Mac: Running Valgrind locally on Mac is possible, but complicated, so instead we recommend utilizing the Snotra cluster. You can still use Qcachegrind locally to visualize the Callgrind output files. See the server guide under Course information for how to transfer files to/from the server.

You can install Homebrew[2] and use it to install Qcachegrind: `brew install qcachegrind`.

Prerequisites

You will also need to install OpenBLAS for this assignment, for Ubuntu systems OpenBLAS can be installed by **sudo apt install libopenblas-dev**. For Mac users openblas can be installed using brew: **brew install openblas**. Mac users will probably also need to make a soft link from the OpenBLAS path to the default library path:
ln -s /usr/local/opt/openblas/include/*.h /usr/local/include.

For profiling You will need valgrind and kcachegrind (for Linux users) or qcachegrind for Mac OS and Windows users. Valgrind will generate a report that can be analyzed using these GUI applications.

The program is compiled and tested for correctness by running "make" in the folder with the Makefile. The compile and linker flags are specified in this Makefile as well, where we link openblas.

2 Provided Files

- Makefile (run **make** to compile and test the project)
- example[1-4].c (test examples to run)
- genann.c (the file to edit for this problem set)
- Other files related to the project that can be ignored.

[1] https://sourceforge.net/projects/qcachegrindwin/

[2] https://brew.sh/

3 Problem description

This problem set consists of three parts, where the third part is optional. Each part is surrounded by a comment block. You should only need to change code inside of these blocks.

Requirements

You are not allowed to change the configuration of the neural network or the number of training steps for your optimizations. A significant reduction(more than 10%) in the test accuracy of the neural network for example 4 will not be accepted. The implementations must give a positive speedup compared with the previous step and baseline.

3.1 Part 1

- Profile the baseline implementation using valgrind and use kcachegrind/q-cachegrind to find the largest hotspots. Save the file as example4-prof-1.
- Time the program using the ”time” command ”time ./example4”. Report the result in a text file.
- Add a comment with ”// HOTSPOT Part 1 Before - your percentage value” at the top two places where the code spends the most time, with the value being how many percent of the total execution time is spent here.
- Go to TODO 1 and make a copy of this section and then block comment[3] out the original code for this section. The hotspot comment should mark where in the original code the hotspot was found. You can then implement the BLAS call for the copy of the code section that is not commented out. This section should be one of the largest hotspots from the previous step.
- After implementing the solution correctly, profile the program again and ensure that you get a speedup. Save the file as example4-prof-2. What is the new execution time and what are the two largest hotspots in the code now? Add a comment with ”// HOTSPOT Part 1 After - your percentage value” for each of the two hotspots.

3.2 Part 2

- Go to TODO 2 and copy and block comment out the original code for this section, and implement the BLAS call. This should be one of the largest hotspots from the previous part.

[3] use /* to start a block comment and */ to end a block comment.

- After implementing the solution correctly, profile the program again and ensure that you get a speedup. Save the file as example4-prof-3. What is the new execution time and what are the two largest hotspots in the code now? Add a comment with "// HOTSPOT Part 2 After - your percentage value" for each of the two hotspots.

3.3 Part 3 (Optional)

- Continue to optimize the program however you see fit.
- Make a copy of your code after finishing Part 2 if you make major revisions to the code for Part 3. Submit both of the versions in this case.
- TODO 3 is one suggestion to a loop to try to optimize.

4 Deliverables

- The genann.c file using BLAS
- A textfile containing:

  - The execution time before and after each task.
- The callgrind output files for before and after each task.