CS 4473/5473: PDN Programming

Dr. Richard M. Veras

PG: Strided Access (Working with Dense Data) v3

Goal: In this assignment you will explore the relationship between dense data (vectors, matrices and tensors) and the memory hierarchy. Much of the parallel code you will write will depend on the careful orchestration of data through the memory sub-system or your target hardware. Most interesting data is high-dimensional, similarly physical memory is also high-dimensional, but efficiently mapping between the two relies on a flat one-dimensional representation of logical memory. These tasks will give you the tools to thread that needle.

Programming Tasks: These problems are meant to get you comfortable with accessing dense data (matrices and tensors) with arbitrary strides. In addition to providing correct code, you will also need to provide test cases that give you 100% coverage with **gcov** and 0 memory leaks reported under **valgrind**. The instructions for running gcov and valgrind will be provided at the end of the document. You may use outside code, but you must document its use. Your submission to canvas will be a tarball (submission.tar.gz) of the problems with the following directory structures:

./Problems

./Problems/FileReader/

./Problems/PrettyPrinter/

./Problems/PrettyBlockedPrinter/

Each directory will contain a prog.c, Makefile and the test cases that you use.

Problems: For each problem create the specified directory and include a **Makefile** and your created test files.

1. **./Problems/FileReader/prog.c**: On the command line read the name of the input file and the output file. The input file will start with a sequence of integers for number of rows 'm,' number of columns 'n,' the row stride 'rs,' the column stride 'cs,' and the number of single character elements to follow along with those characters. The output file will contain the features of the matrix, along with an ASCII graphic that indicates the starting elements of each row and column. Note: column stride means the number of elements in memory between two adjacent elements in a row in a matrix. Similarly, the row stride is the number of elements in memory between two adjacent elements in the same column.

```
./prog.x infilename outfilename
```

Contents of **infilename**	Contents of **outfilename** after running prog.x
2 3 3 1 6 A B C D E F	M: 2 N: 3 RS: 3 CS: 1 MEMORY[6] = {'A','B','C','D','E','F'} ROW_STARTS: V V MEMORY: A, B, C, D, E, F COL_STARTS: ^ ^ ^
2 3 1 2 6 A B C D E F	M: 2 N: 3 RS: 1 CS: 2 MEMORY[6] = {'A','B','C','D','E','F'} ROW_STARTS: V V MEMORY: A, B, C, D, E, F COL_STARTS: ^ ^ ^
2 2 2 4 8 A B C D E F G H	M: 2 N: 2 RS: 2 CS: 4 MEMORY[8] = {'A','B','C','D','E','F','G','H'} ROW_STARTS: V V MEMORY: A, B, C, D, E, F, G, H COL_STARTS: ^ ^

2. **./Problems/PrettyPrinter/prog.c:** On the command line read the name of the input file and the output file. The input file will start with a sequence of integers for number of rows 'm,' number of columns 'n,' the row stride 'rs,' the column stride 'cs,' and the number of single character elements to follow along with those characters. The output file will be the "pretty-printed" representation of that matrix.

./prog.x infilename outfilename

Contents of **infilename**	Contents of **outfilename** after running prog.x
2 3 3 1 6 A B C D E F	m\n (j-->) (i) \ 00_01_02 | 00| A B C V 01| D E F
2 3 1 2 6 A B C D E F	m\n (j-->) (i) \ 00_01_02 | 00| A C E V 01| B D F
3 2 1 3 6 A B C D E F	m\n (j-->) (i) \ 00_01 | 00| A D V 01| B E 02| C F

3. **./Problems/PrettyBlockedPrinter/prog.c:** On the command line read the name of the input file and the output file. The input file will start with a sequence of integers for number of rows 'm,' number of columns 'n,' the row stride 'rs,' the column 'cs,' the number of blocked rows 'mb,' the number of blocked columns 'nb,' the starting row 'i' of the block, the starting column 'j' of the block, and the number of single character elements to follow along with those characters. The output file will be the "pretty-printed" representation of sub-block from the origin. Hint: Run these values (without i, j, mb and nb) in your previous program. What I want to see is the block that starts at (i,j).

```
./prog.x infilename outfilename
```

Contents of **infilename**	Contents of **outfilename** after running prog.x
2 3 3 1 2 2 0 1 6 A B C D E F	mb\nb (j-->) (i) \ 00_01 | 00| B C V 01| E F
3 2 1 3 1 2 1 0 6 A B C D E F	mb\nb (j-->) (i) \ 00_01 | 00| B E V

Additional Resources: In case you are all out of bubble gum.

1. "What Every Programmer Should Know About Memory." Ulrich Drepper. 2007 https://people.freebsd.org/~lstewart/articles/cpumemory.pdf still the best starting point.
2. Cachegrind: https://valgrind.org/docs/manual/cg-manual.html this will simulate memory access of your code and give you a pretty good detailing of cache misses.

Hints: Everything you need to do this assignment can be found in these hints.

1. Makefiles explained: Here and Here

2. GDB is a powerful tool that is worth learning for these problems (run it using the terminal interface flag **gdb -tui**): Here, Here, and Here
3. How to use valgrind to find memory leaks: Here and Here
4. How to use gcov for code coverage: Here and Here (lcov is not necessary)
5. Pointers in C: Here, Here, and Here
6. Reading command line arguments in C: Here
7. Reading and writing a file in C: Here
8. Reading an integer in C: Here
9. Reading a float in C: Here
10. Using the math library in C: Here
11. Allocating an array in C: Here
12. Dynamically allocate a 2D array in C. Here
13. Creating a tarball: Here
14. Play with gcov and valgrind. Then write the Makefile. Below is an example.

example.c

```
/*
  This is a small program to demonstrate gcov. It
  can take no arguments or 1 argument. It prints
  a message if more than one is given. If the argument's
  value is odd it prints out odd, otherwise even.

  The goal is to show you need multiple test cases
  to cover all of the code.
  -richard.m.veras@ou.edu
*/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
// No arguments passed
// note: argv[0] is the name of the
     // program, so we always have at least
     // 1 value in argv.
if (argc == 1+0)
```

```
      printf("I need more arguments!\n");
// If we have 1 argument
else if( argc == 1+1 )
{
// convert the argument to a value
int val = atoi(argv[1]);
if ( val % 2 == 0)
printf("Even!\n");
else
printf("Odd\n");
}
// more than 1 argument
else
printf("I need less arguments!\n");
return 0;
}
```

Makefile

```
# Note: the large spaces are tabs, this is important
# Run GCOV, requires that we build the code
# All of these runs of ./example are a test case
# We want to cover 100% of example through these tests
run-tests-coverage: build
./example
./example 3
./example 8
./example 5 4
gcov example.c

# Run Valgrind to find memory leaks, requires that we build the code
run-tests-leak: build
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-
callers=20 --track-fds=yes ./example
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-
callers=20 --track-fds=yes ./example 3
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-
callers=20 --track-fds=yes ./example 8
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-
callers=20 --track-fds=yes ./example 5 4

# Compile the code, requires that we clean up some files first.
```

```
# Add -lm if your code needs the math library.
build: clean
gcc -fprofile-arcs -ftest-coverage -g example.c -o example

clean:
rm -f *~
rm -f example
rm -f *.gcda *.gcno *.gcov
```