

CMSE/CSE 822 –Parallel Computing
Fall 2019, Homework 2
Due 11:59 pm, Friday, Sep 27th

Important note: Please use a word processing software (e.g., MS Word, Mac Pages, Latex, etc.) to type your homework. Follow the submission instructions at the end to turn in an electronic copy of your work.

1) [25 pts] Performance Modeling

Assume you are evaluating a polynomial of the form,

$x = y^2 + z^3 + yz$, which can be implemented as follows:

```
float x[N], y[N], z[N];  
  
for (i=0; i < N; ++i)  
    x[i] = y[i]*y[i] + z[i]*z[i]*z[i] + y[i]*z[i];
```

Here i is an integer and x,y and z are single precision floating point arrays.

- a. [10 pts] What is the arithmetic intensity of this kernel?
- b. [10 pts] Assume this kernel is to be executed on a processor that has 30 GB/sec of memory bandwidth. Under what conditions will this kernel's performance be memory-bound, and under what conditions will it be compute-bound?
- c. [5 pts] Develop a roofline model for this processor, assuming it has a peak computational throughput of 95 GFLOP/sec. Mark the arithmetic intensity of the kernel on your plot and determine its expected performance.

2) [75 pts] Cache optimization: Matrix Vector Block Multiplication

In this problem, you will optimize the multiplication of an $N \times M$ matrix with block of vectors (whose width for purposes of this assignment is assumed to be constant at 16). This routine takes as input a vector block of size $M \times 16$ and produces an output vector block of size $N \times 16$. A simple nested loop implementation is given below.

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        for (k = 0; k < 16; k++) {
            output[i][k] += input_matrix[i][j]*input_vector[j][k];
        }
    }
}
```

However, as discussed in class, this simple implementation cannot make efficient use of the memory hierarchy. In particular, assuming row-major storage, while the accesses to the input matrix can achieve ideal cache miss rates, accesses to the output matrix may result in a high number of cache misses.

- a. [25 pts] *Implementation*: Develop a cache optimized version of the matrix vector multiplication kernel. For this purpose, you will use the source file provided which already includes the data structures you will use, an implementation of the simple multiplication algorithm, as well as time measurement mechanisms. Your task is to fill in the `optMultiplication` function. While there potentially are better ways to optimize this computation (for instance, by using the blocked matrix-matrix multiplication algorithm), you are expected to use the multi-pass matrix-vector multiplication idea discussed in class.
- b. [20 pts] *Performance Analysis*: Test the performance of your implementation for a set of different N (number of rows), M (number of columns) and B (blocking factor) values. Identify the different regimes as indicated by the relative speed up of the optimized implementation over the naïve one, and outline the conditions (in terms of matrix sizes, shapes, etc.) under which you observe these regimes. **Hint**: Experiment with different matrix sizes (small to large) and input matrix shapes (short&wide rectangle, square, tall&skinny rectangle) while using a reasonable blocking factor.
- c. [20 pts] *Cache Performance Measurement*: Using a cache performance measurement tool (see instructions below on TAU) and your knowledge about how caches work, explain the root causes for the different regimes you observe in part b. For example, you can point to the ratio of cache misses you observe at various levels (L1, L2 or L3) using the naïve vs. optimized implementations to explain the relative performance differences. Can you make an estimation on the relative L1 and L2 latencies based on the performance and cache misses data you obtained?
- d. [15 pts] *Inference about the Memory Hierarchy*: Experiment with different blocking factors B to make inferences about the memory hierarchy. In particular, can you estimate the size of the L1 and L2 caches based on the performance and cache misses data you obtained? Compare your estimations with the hardware specifications.

Instructions:

- **Using the intel16 system:** You should use the intel16 system for all of your runs. Whenever you login to your hpcc you are in gateway nodes and you should never run your programs on the gateway node. After logging into HPCC, run the following command to switch to the intel16 cluster (and then proceed with loading the necessary modules, if applicable)

```
ssh dev-intel16
```

- **Compiling your programs:** While you can use any compiler (different versions of GCC or Intel compiler), throughout this assignment please use GNU/6.4.0-2.28. GNU/6.4.0-2.28 together with OpenMPI 2.1.2 are necessary for compiling and running with TAU, a performance analysis tool that we will be using. So whenever you login to HPCC or in your (batch or interactive) queue runs, make sure to execute the following commands:

```
module unload GNU OpenMPI
module load GNU/6.4.0-2.28 OpenMPI/2.1.2
export TAU_SET_NODE=0
```

After loading the proper modules, you can compile the provided source file in one of the two ways below.

```
make multiplication
make multiplication-tau
```

Running the executables: After compilation, the provided source can be executed either in the "test" or in the "perf" modes, conveniently on any of the dev-nodes:

```
./multiplication.x test inputfile
./multiplication.x perf N M B
```

where the *inputfile* contains a set of test runs. It starts with a line indicating the number of tests, followed by the tests themselves, each on a separate line. Each test is specified with N M B values.

N: number of matrix rows
M: number of matrix columns
B: blocking factor.

- We provide you a sample test file, which is the exact file that we will be using to test the accuracy (not performance) of your submission.
- “Test” mode is meant for accuracy testing only. You are expected to run your resulting implementation in the “perf” mode for timing and cache miss comparisons.
- **Measuring cache misses on HPCC:** For this purpose, we will be using the TAU performance analysis tool. Using hardware counters available through PAPI, TAU allows the measurement of an extensive list of CPU events. To compile your programs with TAU, follow the instructions below:
 - i. Append the following lines to your Bash profile by editing the .bashrc file, which is a hidden file in your home directory (you can open it using vim or emacs, e.g. vim ~/.bashrc)

```
# PAPI
export PATH=/mnt/home/afibuzza/.local/papi/5.6.0/bin:$PATH
export INCLUDE_PATH=/mnt/home/afibuzza/.local/papi/5.6.0/include:$INCLUDE_PATH
export LD_LIBRARY_PATH=/mnt/home/afibuzza/.local/papi/5.6.0/lib:$LD_LIBRARY_PATH
```

```
export MANPATH=/mnt/home/afibuzza/.local/papi/5.6.0/man:$MANPATH

# TAU
export PATH=/mnt/home/afibuzza/.local/tau/2.28/x86_64/bin:$PATH
export INCLUDE_PATH=/mnt/home/afibuzza/.local/tau/2.28/include:$INCLUDE_PATH
export LD_LIBRARY_PATH=/mnt/home/afibuzza/.local/tau/2.28/x86_64/lib:$LD_LIBRARY_PATH
export TAU_MAKEFILE=/mnt/home/afibuzza/.local/tau/2.28/x86_64/lib/Makefile.tau-papi-mpi
export TAU_OPTIONS=-optCompInst
export TAU_METRICS=P_WALL_CLOCK_TIME:PAPI_L1_DCM:PAPI_L2_TCM:PAPI_L3_TCM
```

- ii. To see the complete list of available hardware counters, run on a dev-node

```
papi_avail
```

where supported counters have "Yes" in the third and fourth columns.

- iii. Now, compile the source file using the Tau MPI wrapper:

```
make multiplication-tau
```

- iv. You can now run the executable *instrumented* with TAU on a dev-node or using the queue system:

```
./multiplication-tau.x perf 1000 1000 200
```

- v. The performance data collected by TAU is saved automatically into default directories/files. Below are the directories/files you will see (as a result of the TAU_METRICS options specified):

```
MULTI_P_WALL_CLOCK_TIME/profile.0.0.0
MULTI_PAPI_L1_DCM/profile.0.0.0
MULTI_PAPI_L2_TCM/profile.0.0.0
MULTI_PAPI_L3_TCM/profile.0.0.0
```

- vi. You can go to respective directories and then run the following command:

```
pprof > tauprofile
```

This will generate a tauprofile file in which you can see the cache misses in L1, L2 and L3 in a tabular view. Record these data.

- vii. You can simply view the profile.0.0.0 files for each event using the cat command or a text editor (vim or emacs). The contents are simple enough to be understood in this way.
- viii. Note that, all the files generated by TAU are not overwritten, so make sure to record the cache miss count after each run and then remove/delete those folders before the next run to get the actual cache miss counts of a specific run.

- **Measuring your execution time and cache misses properly:** The wall-clock time measurement mechanism (based on the gettimeofday() function) implemented in the provided main.c file will allow you to measure the timings for a particular part of your program (see the skeleton code) precisely.

However, on the dev-nodes there will be several other programs running simultaneously, and your measurements may not be very accurate due to the “background noise”. After making sure that your program is bug-free and executes correctly (this can be done on the dev-intel16 node, for instance), a good way of getting reliable performance data for various input sizes is to use the interactive queue.

Please use the intel16 cluster for all your performance measurement runs!

You can submit an interactive job request that will you a dedicated node as follows:

```
salloc -N 1 -c 28 -C "intel16" --time=00:30:00 --mem=60G
```

This will give you exclusive access to an intel16 node for 30 minutes. If you ask for a long job, your job may get delayed. **The default memory limit is set to be 750 MBs per job on HPCC systems, so it is very important that you ask for more as above.**

Once you are granted an interactive job, make sure to run your jobs one after the other (i.e., do not run them as background jobs, and do not worry about it if you do not know what background jobs mean). This is important because having multiple background jobs running simultaneously may create “noise” in the data you obtain.

Make sure to load the modules and making the binaries once you get an interactive job. Sometimes the interactive nodes will have a different module loaded which might not find the correct linker libraries.

- **Batch jobs:** Interactive jobs may sometimes be delayed significantly, and therefore insisting on this option may be counter-productive. For your convenience, we have provided a sample batch script named **job_script.sb**. Then submit the job using **sbatch job_script.sb**.
- **Obtaining files from the git repo & submission:** You will clone the skeleton code and the testing input file through the instructor repository on the Gitlab server. Assuming that you have already cloned the instructor repository, you will need to pull the most recently committed files for HW2 and move them over into your individual repository:

```
cd cmse822_fs19-instructor
git pull
cd ../cmse822_fs19-username/homework
cp -r ../../cmse822_fs19-instructor/homework/2 .
```

Then complete the homework and submit it using your own personal repository. Your submission will include exactly two files:

- Your final optMultiplication.c file
- A pdf file named “HW2_yourMSUNetID.pdf”, which contains your answers to the non-implementation questions of the assignment.