CS351 Lab #6

Benchmarking CPU & Memory Hierarchy

Instructions:

- *Assigned date: Thursday April 14th, 2022*
- *Due date: 11:59PM on Friday April 29th, 2022 [firm deadline, no extensions; no submissions after May 6th, 2022 will be accepted]*
- *Maximum Points: 60 points*
- *Extra Credit: 20 points*
- *This lab must be done individually*
- *Please post your questions to BB*
- *Only a softcopy submission is required; it will automatically be collected through GIT at the deadline; email confirmation will be sent to your HAWK email address; late submission will be penalized at 10% per day; your submission will not be graded until you have submitted a single page PDF file with your name, email, A#, assignment number, and URL to the gitlab repo*

1 Your Assignment

This project aims to teach you how to benchmark the processor. You can be creative with this project. You are free to use any of the following programming languages (C) and abstractions (PThreads) that might be needed. Libraries such as STL or Boost cannot be used. You can use any Linux system for your development, but you must make sure it compiles and runs correctly on fourier. The performance evaluation should be done on your own computer in a Linux environment (e.g. think back at Lab #0). Given the large class size, we encourage all students to avoid using fourier for anything but testing functionality on small datasets.

In this project, you need to design a benchmarking program that computes matrix multiplication of square matrices of arbitrary size (within the constraints of available memory). You will perform strong scaling studies, unless otherwise noted; this means you will set the amount of work (e.g. the number of instructions or the amount of data to evaluate in your benchmark), and reduce the amount of work per thread as you increase the number of threads. The TAs will compile (with the help of make) and test your code on fourier.

Your cpubench program should support the following benchmarks:

1. Mode:
    a. Flops: Measure integer (int) and floating point (double) instructions per second your processor can do
    b. Matrix: Implement matrix multiplication (https://en.wikipedia.org/wiki/Matrix_multiplication) supporting square matrices, of arbitrary size. For example, you should be able to support matrices of size 2, where 2x2 matrix is multiplied by another 2x2 matrix, and result in a 2x2 matrix. You need to implement the matrix multiplication to support multiple cores, as you will evaluate its performance in a multi-core environment.
2. Type:
    a. Single: single precision 4-byte int data types

   b. Double: double precision 8-byte double data types
3. Size:
   a. Small: for flops, this should be set to 10 billion operations (10^10); for matrix, this should be set to 1024 (e.g. 1024x1024 matrix) requiring 12MB to 32MB depending on the algorithm and data type
   b. Medium: for flops, this should be set to 100 billion operations (10^11); for matrix, this should be set to 4096 (e.g. 4096x4096 matrix) requiring 0.2GB to 0.6GB depending on the algorithm and data type
   c. Large: for flops, this should be set to 1000 billion operations (10^12); for matrix, this should be set to 16384 (e.g. 16384x16384 matrix) requiring 3GB to 8GB depending on the algorithm and data type
4. Threads:
   a. 1: Use 1 thread to execute your benchmark on the specified size dataset/operations
   b. 2: Use 2 threads to execute your benchmark in parallel on the specified size/2 dataset/operations per thread
   c. 4: Use 4 threads to execute your benchmark in parallel on the specified size/4 dataset/operations per thread

You are to measure the processor speed, in terms of operations per second; report data in GFlops, giga operations (10^9) per second. Here is the sample output of 4 different runs on a dual socket AMD Epyc 7501 processor with 64-cores at 2.0 GHz:

```
(base) iraicu@linux:~/cpubench$ ./cpubench flops double 1000 1
mode=flops type=double size=1000 threads=1 time=221.60366 throughput=4.5125

(base) iraicu@linux:~/cpubench$ ./cpubench flops double 1000 64
mode=flops type=double size=1000 threads=64 time=3.49288 throughput=286.2971


(base) iraicu@linux:~/cpubench$ ./cpubench matrix double 16384 1
mode=matrix type=double size=16384 threads=1 time=6601.354944 throughput=0.6205

(base) iraicu@linux:~/cpubench$ ./cpubench matrix double 16384 64
mode=matrix type=double size=16384 threads=64 time=103.862824 throughput=39.4366
```

You can compute the theoretical flops/sec with the following formula:

FLOPS = (sockets) x (cores per socket) x (cycles per second) x (FLOPS per cycle)

For the system I tested this on, I can plug in the following numbers:

1024.0 GFlops/sec = 2 x 32 x 2.0GigaCycles x 8

The cycle per second is determined by the speed of the processor, and the flops per cycle is determined by the processor architecture and is typically a number between 1 and 16. AMD Zen architecture has 8 flops per cycle, while Intel CPUs from the last several years have 16 flops per cycle. Older processors could have as few as 1 or 2 flops per cycle. Don't guess what it is for your process, look it up under the processor specifications, and cite your result with where you found this information.

Fill in the table 1 below for the flops mode of the processor performance:

Mode	Type	Size	Threads	Measured Time	Measured Throughput	Theoretical Throughput	Efficiency
flops	single	small	1				
flops	single	small	2				
flops	single	small	4				
flops	single	medium	1				
flops	single	medium	2				
flops	single	medium	4				
flops	single	large	1				
flops	single	large	2				
flops	single	large	4				
flops	double	small	1				
flops	double	small	2				
flops	double	small	4				
flops	double	medium	1				
flops	double	medium	2				
flops	double	medium	4				
flops	double	large	1				
flops	double	large	2				
flops	double	large	4				

Fill in the table 2 below for the matrix mode of the processor performance:

Mode	Type	Size	Threads	Measured Time	Measured Throughput	Theoretical Throughput	Efficiency
matrix	single	small	1				
matrix	single	small	2				
matrix	single	small	4				
matrix	single	medium	1				
matrix	single	medium	2				
matrix	single	medium	4				
matrix	single	large	1				
matrix	single	large	2				
matrix	single	large	4				
matrix	double	small	1				
matrix	double	small	2				
matrix	double	small	4				
matrix	double	medium	1				
matrix	double	medium	2				
matrix	double	medium	4				
matrix	double	large	1				
matrix	double	large	2				
matrix	double	large	4				

Other requirements:

- You must write all benchmarks from scratch. You can use well known benchmarking software to verify your results, but you must implement your own benchmarks. Do not use code you find online, as you will get 0 credit for this assignment.
- All of the benchmarks will have to evaluate concurrency performance; concurrency can be achieved using threads and processes. For this assignment, you must use multi-threading. Use strong scaling in all experiments, unless it is not possible, in which case you need to explain why a strong scaling experiment was not done. Be aware of the thread synchronizing issues to avoid inconsistency or deadlock in your system.
- All your benchmarks can be run on a single machine. You should run these benchmarks on your own system under Linux. Make sure your work compiles and can run the small workloads on fourier.
- Not all timing functions have the same accuracy; you must find one that has at least 1ms accuracy or better, assuming you are running the benchmarks for at least seconds at a time.
- Since there are many experiments to run, find ways (e.g. scripts) to automate the performance evaluation.
- If you don't have enough memory to run the large size benchmark for matrix, run the largest size possible given the memory you have.
- If the execution of your benchmarks is taking too long for you to complete the assignment on time, significantly longer than the examples shown in this writeup, then it's possible you are doing something wrong and are not getting a good enough and efficient use of your hardware. Trivial solutions to the proposed benchmarks will not run efficiently and could take on the order of 10 to 20 hours for a single large run of matrix computation.
- There are likely some compiler flags that will help you execute your benchmarks faster.
- Achieving 100% efficiency is likely impossible. Your initial versions of your benchmarks might be less than 1% efficient, for which you will receive some credit, but not all. Once you have a functional benchmark, work on fine tuning it to improve your efficiency.
- For insights on tunning your matrix multiplication benchmark, read this detailed website: https://gist.github.com/nadavrot/5b35d44e8ba3dd718e595e40184d03f0
- For some insights into instruction level parallelism, read this short Wikipedia article: https://en.wikipedia.org/wiki/Instruction-level_parallelism
- You are likely going to need to use dynamic memory allocation using malloc().
- Remember that signed integers can only hold values up to 2 billion; be careful when running the trillion operation flops benchmark.
- You can learn more about floating point operations per second (flops) in this Wikipedia article: https://en.wikipedia.org/wiki/FLOPS
- **EXTRA CREDIT:** A detailed document on installing and running HPL can be found at: http://www.crc.nd.edu/~rich/CRC_EPYC_Cluster_Build_Feb_2018/Installing%20and%20running%20HPL%20on%20AMD%20EPYC%20v2.pdf
- No GUIs are required. Simple command line interfaces are expected.

EXTRA CREDIT

For 20 extra credit points, install and run the HPL benchmark from the Linpack suite (http://en.wikipedia.org/wiki/LINPACK) and report the best performance achieved using double precision floating point; make sure to run Linpack across all cores. You should use the same problem size as in the matrix multiplication you implemented and evaluated earlier. You can download the Modified HPL Project from http://www.nics.tennessee.edu/sites/default/files/HPL-site/home.html (you need to download it, and compile it). You are going to run shpl (single precision) and xhpl (double precision). If you have trouble

with the Modified HPL download, you can try the latest HPL benchmark from http://www.netlib.org/benchmark/hpl/. Make sure to tune the HPL benchmark in HPL.dat (see http://www.netlib.org/benchmark/hpl/tuning.html for more information on how to interpret the HPL.dat configuration). You will need a working MPI installation; you can install MPICH or OpemMPI with your package manager. You will need to install a BLAS implementation (shared library, not static) using the package manager (e.g. BLAS, LAPACK, or ATLAS). You will need to update the options file with the path to the BLAS shared library. Getting HPL install, compiled, and run is not trivial. But it's something that can be done in 10 to 20 minutes.

Here is the sample output of 2 different runs on a dual socket AMD Epyc 7501 processor with 64-cores at 2.0 GHz (there is a HPL.dat that must contain many configuration parameters):

```
(base) iraicu@linux:~/cpubench$ mpirun -np 1 xhpl
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR12R2R4       16384   232     1     1             326.86             1.253e+01

(base) iraicu@linux:~/cpubench$ mpirun -np 64 xhpl
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR12R2R4       16384   232     8     8               5.15             7.950e+02
```

Fill in the table 3 below for the HPL benchmark of the processor performance for extra credit:

Mode	Type	Size	Threads	Measured Time	Measured Throughput	Theoretical Throughput	Efficiency
shpl	single	1024	4				
shpl	single	4096	4				
shpl	single	16386	4				
xhpl	double	1024	4				
xhpl	double	4096	4				
xhpl	double	16386	4				

2 What you will submit

When you have finished implementing the complete assignment as described above, you should submit your solution to your private git repository. Each program must work correctly and be detailed in-line documented. You should hand in:

1. **Source code and compilation (70%):** All of the source code in C and Bash; in order to get full credit for the source code, your code must have in-line documents, must compile (with a Makefile), and must be able to run a variety of benchmarks through command line arguments. Must have working code that compiles and runs on fourier.

2. **Report / Performance (30%):** A separate (typed) design document (named lab6-report.pdf) describing the results in a table format. You must evaluate the performance of the various parameters outlined and fill in the 2 tables specified to showcase the results. You must summarize your findings and explain why you achieve the performance you achieve, and how the results compare between the various approaches.

To submit your work, simply commit all your changes to the Makefile, cpubench.c, runbench.sh, HPL.dat (EC), and push your work to Github. You can find a git cheat sheet here: https://www.git-tower.com/blog/git-cheat-sheet/. Your solution will be collected automatically at the deadline. If you want to submit your homework later, you will have to push your final version to your GIT repository and you will have let the TA know of it through email. There is no need to submit anything on BB for this assignment. If you cannot access your repository contact the TAs.

Grades for late programs will be lowered 10% per day late.